Neural Acceleration for GPU Throughput Processors

Amir Yazdanbakhsh Jongse Park Hardik Sharma
Pejman Lotfi-Kamran† Hadi Esmaeilzadeh
Alternative Computing Technologies (ACT) Lab
School of Computer Science, Georgia Institute of Technology
†School of Computer Science, Institute for Research in Fundamental Sciences (IPM)
{a.yazdanbakhsh, jspark, hsharma}@gatech.edu plotfi@ipm.ir hadi@cc.gatech.edu

ABSTRACT

Graphics Processing Units (GPUs) can accelerate diverse classes of applications, such as recognition, gaming, data analytics, weather prediction, and multimedia. Many of these applications are amenable to approximate execution. This application characteristic provides an opportunity to improve GPU performance and efficiency. Among approximation techniques, neural accelerators have been shown to provide significant performance and efficiency gains when augmenting CPU processors. However, the integration of neural accelerators within a GPU processor has remained unexplored. GPUs are, in a sense, many-core accelerators that exploit large degrees of data-level parallelism in the applications through the SIMT execution model. This paper aims to harmoniously bring neural and GPU accelerators together without hindering SIMT execution or adding excessive hardware overhead. We introduce a low overhead neurally accelerated architecture for GPUs, called NGPU, that enables scalable integration of neural accelerators for large number of GPU cores. This work also devises a mechanism that controls the tradeoff between the quality of results and the benefits from neural acceleration. Compared to the baseline GPU architecture, cycle-accurate simulation results for NGPU show a 2.4× average speedup and a 2.8× average energy reduction within 10% quality loss margin across a diverse set of benchmarks. The proposed quality control mechanism retains a 1.9× average speedup and a 2.1× energy reduction while reducing the degradation in the quality of results to 2.5%. These benefits are achieved by less than 1% area overhead.

Categories and Subject Descriptors

C.1 [**Processor Architecture**]: Neural nets

Keywords

Approximate computing, GPU, neural processing unit

DOI: http://dx.doi.org/10.1145/2830772.2830810

1 Introduction

The diminishing benefits from CMOS scaling [1–3] has coincided with an overwhelming increase in the rate of data generation. Expert analyses show that in 2011, the amount of generated data surpassed 1.8 trillion GB and by 2020, consumers will generate 50× this staggering figure [4]. To overcome these challenges, both the semiconductor industry and the research community are exploring new avenues in computer architecture design. Two of the promising approaches are acceleration and approximation. Among programmable accelerators, GPUs provide significant gains in performance and efficiency. GPUs that were originally designed to accelerate graphics functions, now are being used for a wide range of applications, including recognition, learning, gaming, data analytics, weather prediction, molecular dynamics, multimedia, scientific computing, and many more. The availability of programming models for GPUs and the advances in their microarchitecture have played a significant role in their widespread adoption. Many companies, such as Microsoft, Google, and Amazon use GPUs to accelerate their enterprise services. As GPUs play a major role in accelerating many classes of applications, improving their performance and efficiency is imperative to enable new capabilities and to cope with the ever-increasing rate of data generation.

Many of the applications that benefit from GPUs are also amenable to imprecise computation [6–9]. For these applications, some variation in output is acceptable and some degradation in the output quality is tolerable. This characteristic of many GPU applications provides a unique opportunity to devise approximation techniques that trade small losses in the quality of results for significant gains in performance and efficiency. Among approximation techniques, neural acceleration provides significant gains for CPUs [10–14] and may be a good candidate for GPUs. Neural acceleration relies on an automated algorithmic transformation that converts an approximable segment of code[1] to a neural network. This transformation is called the neural transformation [10]. The compiler automatically performs the neural transformation and replaces the approximable segment with an invocation of a neural hardware that

[1]Approximable code is a segment that if approximated will not lead to catastrophic failures in execution (e.g., segmentation fault) and its approximation may only lead to graceful degradation of the application output quality.

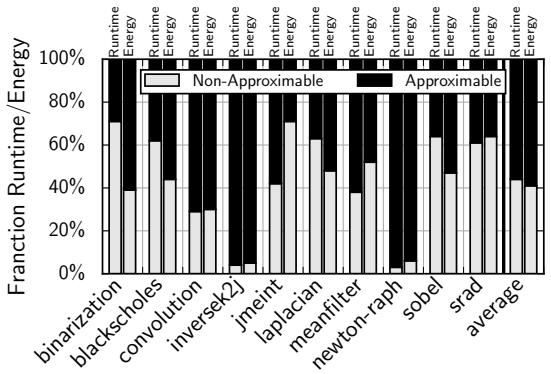


Figure 1: Runtime and energy breakdown between neurally approximable regions and the regions that cannot be approximated.

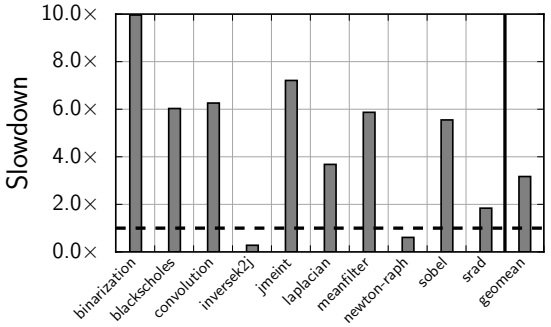


Figure 2: Slowdown with software-only neural transformation due to the lack of hardware support for neural acceleration.

accelerates the execution of that segment.

To examine the potential benefits of neural acceleration in GPUs, we first study[2] its applicability to a diverse set of representative CUDA applications. Figure 1 illustrates the results and shows the breakdown of application runtime and energy dissipation between neurally approximable regions and the regions that cannot be neurally approximated[3]. The neurally approximable segments are the ones that can be approximated by a neural network. On average, applications spend 56% of their runtime and 59% of their energy in neurally approximable regions. Some applications such as inversek2j and newton-raph spend more than 93% of their runtime and energy in neurally approximable regions. These encouraging results demonstrate the significant potential of neural acceleration for GPU processors.

Why hardware acceleration? As previous work [15] suggested, it is possible to apply neural transformation with no hardware modifications and replace the approximable region with an efficient software implementation of the neural network that mimics the region. We explored this possibility and the results are presented in Figure 2. On average, the applications suffer from 3.2× slowdown. Only inversek2j and newton-raph, which spend more than 93% of their time in the neurally approximable region, see 3.6× and 1.6× speedup, respectively. The slowdown with software implementation is due to (1) the overhead of fetching/decoding the instructions, (2) the cost of frequent accesses to the memory/register file, and (3) the overhead of executing the sigmoid function. The significant potential of neural transformation (Figure 1) and the slowdown with the software-only approach (Figure 2) necessities designing GPU architectures with integrated neural accelerators.

Why not reuse CPU neural accelerators? Previous work [10] proposes an efficient hardware neural accelerator for CPUs. One possibility is to use CPU Neural Processing Unit (NPU) in GPUs. However, compared to CPUs, GPUs contain (1) significantly larger number of cores (SIMD lanes) that are also (2) simpler. Augmenting each core with a NPU that harbors several parallel processing engines and buffers imposes significant area overhead. Area overhead of integrating NPUs to a GPU while reusing SIMD lanes' multiply-add units is 31.2%. Moreover, neural networks are structurally parallel. Hence, replacing a code segment with neural networks adds structured parallelism to the thread. In the CPU case, NPU's multiple multiply-add units exploit this added parallelism to reduce the thread execution latency. GPUs, on the other hand, already exploit data-level parallelism and leverage many-thread execution to hide thread latencies. One of the insights from this work is that the added parallelism is not the main source of benefits from neural acceleration in GPUs. Therefore, neural acceleration in GPUs leads to a significantly different hardware design as compared to CPUs.

Contributions. To this end, the following are the major contributions of this work.

- While this work is not the first to explore neural acceleration, it is the first to evaluate tight integration of neural acceleration within GPU cores. Integrating neural accelerators within GPUs is fundamentally different from doing so in a CPU because of the hardware constraints and the many-thread SIMT execution model in GPUs.
- We observe that, unlike CPUs, the added parallelism is not the main source of benefits from neural acceleration in GPUs. The gains in GPUs come from (1) eliminating the fetch/decode during neural execution, (2) reducing accesses to the memory/register file by storing the parameters and the partial results in small buffers within the SIMD lanes, and (3) implementing sigmoid as a lookup table. This insight leads to a low overhead integration of neural accelerators to SIMD lanes by limiting the number of ALUs in an accelerator to only the one that is already in a SIMD lane.
- Through a combination of cycle-accurate simulations and a diverse set of GPU applications from different domains (finance, machine learning, image processing, vision, medical imaging, robotics, 3D gaming, and numerical analysis), we rigorously evaluate the proposed NGPU design. Compared to the baseline GPU, NGPU achieves a 2.4× average speedup and a 2.8× average energy reduction within a 10% quality loss margin. These benefits are achieved with less than 1% area overhead.
- We also devise a mechanism that controls the trade-off between the quality loss and performance and efficiency gains. The quality control mechanism

[2]Section 6.1 presents our experimental methodology with the GPGPU-Sim cycle-accurate simulator.

[3]The annotation procedure is discussed in Section 2.

retains a 1.9× average speedup and a 2.1× energy reduction while reducing the quality loss to 2.5%.

2 Neural Transformation for GPUs

To enable the integration of neural accelerators within GPUs, the first step is to develop a compilation workflow that automatically performs the neural transformation on GPU code. We also need to develop a programming interface that enables developers to delineate approximable regions as candidates for the neural transformation.

2.1 Safe Programming Interface

Any practical approximation technique, including ours, needs to provide execution safety guarantees. That is, approximation should never lead to catastrophic failures such as out-of-bound memory accesses. In other words, approximation should never affect critical data and operations. The criticality of data and operations is a semantic property of the program and can only be identified by the programmer. The programming language must therefore provide a mechanism for programmers to specify where approximation is safe. This requirement is commensurate with prior work on safe approximate programming languages such as EnerJ [16], Rely [17], FlexJava [18], and Axilog [19]. To this end, we extend the CUDA programming language with a pair of `#pragma` annotations that mark the start and the end of a safe-to-approximate region of GPU code. The following example illustrates these annotations.

```
#pragma(begin_approx,"min_max")
mi = __min(r, __min(g, b));
ma = __max(r, __max(g, b));
result = ((ma + mi) > 127 * 2) ? 255 : 0;
#pragma(end_approx,"min_max")
```

This segment of the binarization benchmark is approximable and is marked as a candidate for transformation. The `#pragma(begin_approx,"min_max")` marks the segment's beginning and names it the `"min_max"` segment. The `#pragma(end_approx,"min_max")` marks the end of the segment that was named `"min_max"`.

2.2 Compilation Workflow

As discussed, the main idea of neural algorithmic transformation is to learn the behavior of a code segment using a neural network and then replace the segment with an invocation of an efficient neural hardware. To implement this algorithmic transformation, the compiler needs to (1) identify the inputs and outputs of the segment, (2) collect the training data by observing (logging) the inputs and outputs, (3) find and train a neural network that mimics the observed behavior, and finally (4) replace that region of code with instructions that configure and invoke the neural hardware. These steps are illustrated in Figure 3. Our compilation workflow is similar to the one described in prior work that targets neural acceleration in CPUs [10]. However, we specialize these steps for GPU applications and add the automatic *input/output identification step to the compilation workflow* to further automate the transformation.

❶ **Input/output identification.** To train a neural network that mimics a code segment, the compiler needs to collect the input-output pairs that represent the functionality of the region. The first step is identifying the inputs and outputs of the delineated segment. The compiler uses a combination of live variable analysis and Mod/Ref analysis [20] to automatically identify the inputs and outputs of the annotated segment. The inputs are the intersection of live variables at the location of `#pragma(begin_approx,...)` with the set of variables that are referenced within the segment. The outputs are the intersection of live variables at the location of `#pragma(end_approx,...)` with the set of variables that are modified within the segment. In the previous example, this analysis identifies `r`, `g`, and `b` as the inputs to the region and `result` as the output.

❷ **Code observation.** After identifying the inputs and outputs of the segment, the compiler instruments these inputs and outputs to log their values in a file as the program runs. The compiler then runs the program with a series of representative input datasets (such as the ones from a program test suite) and logs the pairs of input-output values. The collected set of input-output values constitutes the training data that captures the behavior of the segment.

❸ **Topology selection and training.** This step needs to both find a topology for the neural network and train it. In finding the topology, the objective is to strike a balance between the network's accuracy and its efficiency. Theoretically, a larger, more complex network offers better accuracy potential but is likely to be slower and less efficient. However, enlarging the network does not improve its accuracy beyond a certain point. Thus, the compiler considers a search space for the neural topology and picks the smallest network that delivers comparable accuracy to the largest network in the space. The neural network of choice is Multilayer Perceptron (MLP) that consists of a fully-connected set of neurons organized into layers: the input layer, a number of hidden layers, and the output layer. The number of neurons in the input and output layers is fixed and corresponds to the number of inputs and outputs of the code segment. The problem is finding the number of hidden layers and the number of neurons in each hidden layer.

The space of possible topologies is infinitely large. Therefore, we restrict the search space to the neural networks with at most two hidden layers. The number of neurons per hidden layer is also restricted to powers of two, up to 32 neurons. These choices limit the search space to 30 possible topologies. The maximum number of hidden layers and the maximum neurons per hidden layer are compilation options and can be changed if needed. These neural networks are trained independently in parallel. To find the best fitting neural network topology, we partition the application input datasets into a training dataset ($\frac{2}{3}$ of the programmer-provided application input datasets), and a selection dataset, (the remaining $\frac{1}{3}$). The training datasets are used during training, and the selection datasets are used to select the final neural network topology based on the application's desired quality loss. Note that we use completely separate input datasets to measure the final

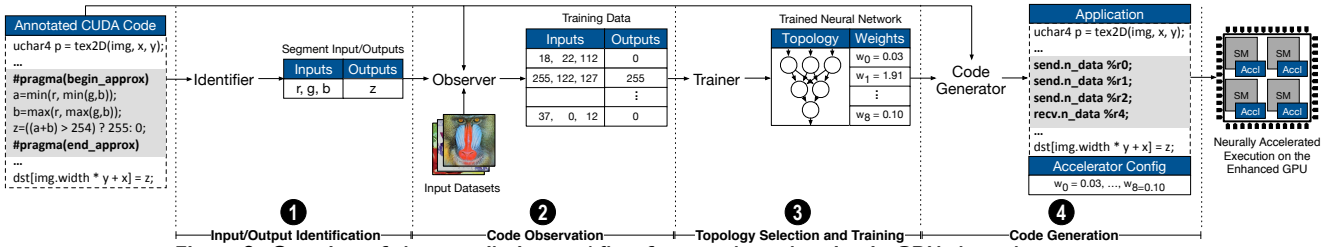


Figure 3: Overview of the compilation workflow for neural acceleration in GPU throughput processors.

quality loss in Section 6.

To train the networks for neural acceleration, we use the standard backpropagation [21] algorithm. Our compiler performs 10-fold cross-validation for training each neural network. The output from this phase consists of a neural network topology – specifying the number of layers and the number of neurons in each layer – along with the weights for the inputs of each neuron that are determined by the backpropagation training algorithm.

❹ **Code generation.** After identifying the neural network and training it, the compiler replaces the code segment with special instructions to send the inputs to the neural accelerator and retrieve the results. The compiler also configures the neural accelerator. The configuration includes the weights and the schedule of the operations within the accelerator. This information is loaded into the integrated neural accelerators once when the program loads for execution.

3 Instruction Set Architecture Design

To enable neural acceleration, the GPU ISA should provide three instructions: (1) one for sending the inputs to the neural accelerator, (2) one for receiving outputs from the neural accelerator, and finally (3) one for sending the accelerator configuration and the weights. To this end, we extend the PTX ISA as follows:

1. `send.n_data %r`: This instruction sends the value of register `%r` to the neural accelerator as an input.
2. `recv.n_data %r`: This instruction retrieves a value from the accelerator and writes it to the register `%r`.
3. `send.n_cfg %r`: This instruction sends the value of register `%r` to the accelerator and indicates that the value is for configuration.

We use PTX ISA 4.2 which supports vector instructions that can read or write two or four registers instead of one. We take advantage of this feature and introduce two vector versions for each of our instructions. The `send.n_data.v2 {%r0, %r1}` sends two register values to the accelerator and a single `send.n_data.v4 {%r0, %r1, %r2, %r3}` sends the value of four registers to the neural accelerator. The vector versions for `recv.n_data` and `send.n_cfg` have similar semantics. These vector versions reduce the number of instructions that need to be fetched and decoded to communicate with the neural accelerator. This reduction lowers the overhead of invoking the accelerator and provides more opportunities for speedup and efficiency gains.

As follows, these instructions will be executed in SIMT mode as other GPU instructions. GPU applications typically consist of kernels and GPU threads execute the same kernel code. The neural transformation approximates segments of these kernels. That is, each corresponding thread will contain the aforementioned instructions to communicate with the neural accelerator. Each thread only applies different input data to the same neural network. GPU threads are grouped into cooperative thread arrays (a unit of thread blocks). The threads in different thread blocks are independents and can be executed in any order. The thread block scheduler maps them to GPU processing cores called the streaming multiprocessors (SMs). The SM divides threads of a thread block into smaller groups called warps, typically of size 32 threads. All the threads within a warp execute the same instruction in lock-step. The three new instructions, `send.n_data`, `recv.n_data`, and `send.n_cfg` follow the same SIMT model. That is, executing each of these instructions, conceptually, communicates data with 32 parallel neural accelerators.

A typical GPU architecture, such as Fermi [22], contains 15 SMs, each with 32 SIMD lanes. That is, to support hardware neural acceleration, 480 neural accelerators need to be integrated. Hence the GPU-specific challenge is designing a hardware neural accelerator that can be replicated many times within the GPU without imposing extensive hardware overhead.

4 Accelerator Design and Integration

To describe our neural accelerator design and its integration into the GPU architecture, we assume a GPU processor based on the Nvidia Fermi. Fermi's SMs contain 32 double-clocked SIMD lanes that execute two half warps (16 threads) simultaneously, where each warp executes in lock-step. Ideally, to preserve the data-level parallelism across the threads and preserve the default SIMT execution model, each SM needs to be augmented with 32 neural accelerators. Therefore, the objective is to design a neural accelerator that can be replicated 32 times within each SM for a minimal hardware overhead. These two requirements fundamentally change the design space of the neural accelerator from prior work that aims at accelerating single-thread cores with only one accelerator.

A naïve approach is to replicate and add the previously proposed CPU neural accelerator to each SM [10]. These CPU specific accelerators harbor multiple processing engines and contain significant amount of buffering for weights and control. Such a design not only imposes significant hardware overhead, but also is an overkill for data-parallel GPU architectures as our results in Section 6.3 show. Instead, we tightly integrate a GPU specific neural network in every SIMD lane.

The neural algorithmic transformation uses Multilayer Perceptrons (MLPs) to approximate CUDA code segments. As Figure 5a depicts, an MLP consists of a

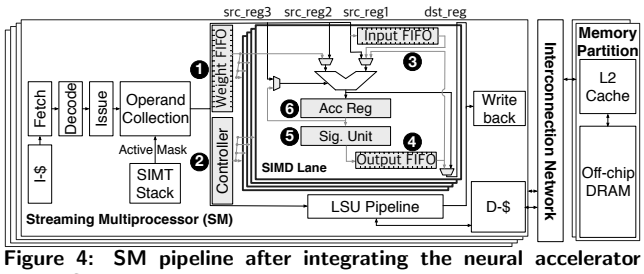


Figure 4: SM pipeline after integrating the neural accelerator within SIMD lanes. The added hardware is highlighted in gray.

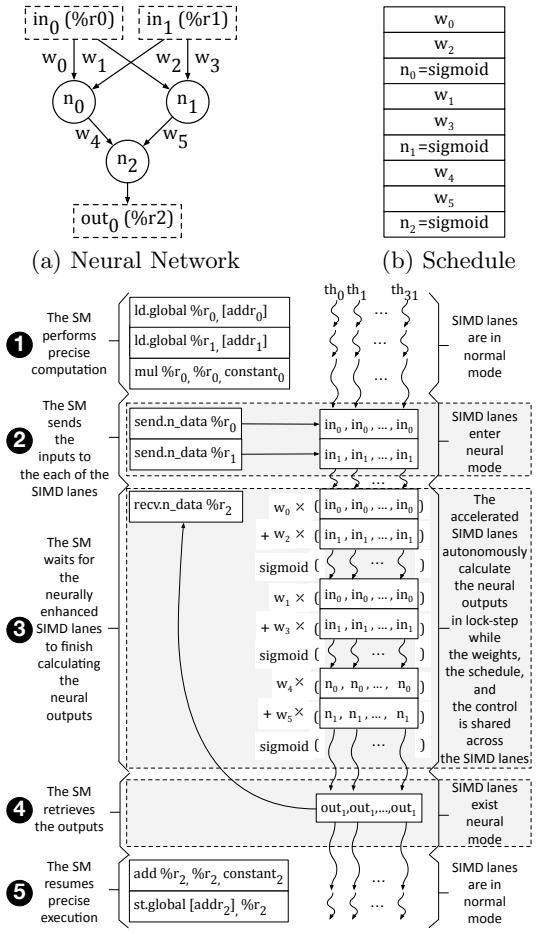
network of neurons arranged in multiple layers. Each neuron in a layer is connected to all of the neurons in the next layer. Each neuron input is associated with a weight value that is generated after training. All neurons are identical and each neuron computes its output (y) based on $y = sigmoid(\sum_i(w_i \times x_i))$, where x_i is a neuron input and w_i is the input's associated weight. Therefore, all the computations of a neural network are a set of multiply-add operations followed by the non-linear sigmoid operation. The neural accelerator only needs to support these two operations.

4.1 Integrating the Neural Accelerator

Each SM has 32 SIMD lanes, divided into two 16-lane groups that execute two half warps simultaneously. The ALU in each lane supports floating point multiply-add operation. We reuse these ALUs while enhancing the lanes for neural computation. We leverage the existing SIMT execution model to minimize the hardware overhead for the weights and control. We refer to the resulting SIMD lanes as neurally enhanced SIMD lanes.

In Figure 4, the added hardware components are numbered and highlighted in gray. The first component is the `Weight FIFO` (❶) that is a circular buffer and stores the synaptic weights. Since all of the threads are approximated by the same neural network, we only add one `Weight FIFO`, which is shared across all SIMD lanes. The Weight FIFO has two read ports corresponding to the two 16 SIMD lanes that execute two half warps. Each port supplies a weight to 16 ALUs. The second component is the `Controller` (❷) which controls the execution of the neural network across the SIMD lanes. Again, the `Controller` is shared across 16 SIMD lanes that execute a half warp (two controllers per SM). The `Controller` follows the SIMT pattern of execution for neural computation and enables the ALUs to perform the computation of the same input of the same neuron in the network.

We augment each of the SIMD lanes with an `Input FIFO` (❸) and an `Output FIFO` (❹). The `Input FIFO` stores the neural network inputs. The `Output FIFO` stores the output of the neurons including the output neurons that generate the final output. These two are small FIFO structures that are replicated for each SIMD lane. Each of the SIMD lanes also harbors a `Sigmoid Unit` (❺) that contains a read-only lookup table. This lookup table implements the nonlinear sigmoid function and is synthesized as combinational logic to reduce the area overhead. Finally, the `Acc Reg` (❻), which is the accumulator register in each of the SIMD lanes, retains the partial results of the sum of products ($\sum_i(w_i \times x_i)$) before passing it to the `Sigmoid Unit`.



(a) Neural Network (b) Schedule

(c) Accelerated Execution on the Enhanced SM

Figure 5: (a) Neural network replacing a segment of a GPU code. (b) Schedule for the accelerated execution of the neural network. (c) Accelerated execution of the GPU code on the enhanced SM.

One of the advantages of this design is that it limits all major modifications to the execution part of the SIMD lanes (pipelines). There is no need to change any other part of the SM except for adding support for decoding the ISA extensions that communicate data to the accelerator (i.e., input and output buffers). Scheduling and issuing these instructions are similar to arithmetic instructions and do not require specific changes.

4.2 Executing Neurally Transformed Threads

Figure 5c illustrates the execution of a neurally transformed warp, which contains normal precise and special approximate (i.e., `send.n_data`/`recv.n_data`) instructions, on its neurally enhanced SIMD lanes. The other simultaneously executing warp (similarly contains both normal and special instructions) is not shown for clarity. In the first phase (❶), SIMD lanes execute the precise instructions as usual before reaching the first `send.n_data` instructions. In the second phase (❷), SIMD lanes execute the two `send.n_data` instructions to copy the neural network inputs from the register file to their input buffers. These instructions cause SIMD lanes to switch to the neural mode. In the third phase (❸), the enhanced SIMD lanes perform the neural computation and store the results in their output buffers.

At the same time, the SM issues `recv.n_data`, but since the output of the neural network is not ready yet, the SM stops issuing the next instruction and waits for the neurally-enhanced SIMD lanes to finish computing the neural network output. In the fourth phase (❹), once the neural network output is ready, `recv.n_data` instruction copies the results from the output buffer to the register file and then in the fifth phase (❺) normal execution resumes. As there is no control divergence or memory access in the neural mode, our design does not swap the running warp with another warp *in the neural mode* to avoid the significant overhead of dedicated input/output buffers or control logic per active warp (SMs support 48 ready-to-execute warps).

4.3 Orchestrating Neurally Enhanced Lanes

To efficiently execute neural networks on the neurally enhanced SIMD lanes, the compiler needs to create a static schedule for the neural computation and arrange the weights in proper order. This schedule and the pre-ordered weights are encoded in the program binary and are preloaded to the `Weight FIFO` (Figure 4 ❶) when the program loads for execution. The compiler generates the execution schedule based on the following steps:

1. The computations for the neurons in each layer are dependent on the output of the neurons in the previous layer. Thus, the compiler first assigns a unique order to the neurons starting from the first hidden layer down to the output layer. This order determines the execution of the neurons. In Figure 5a. n_0, n_1, and n_2 show this order.
2. Then, for each neuron, the compiler generates the order of the multiply-add operations, which are followed by a sigmoid operation. This schedule is shown in Figure 5b for the neural network in Figure 5a. The phase (❸) of Figure 5c illustrates how the neurally enhanced SIMD lanes execute this schedule in SIMT mode while sharing the weights and control.

The schedule that is presented in Figure 5b constitutes the most of the accelerator configuration and the order in which the weights will be stored in `Weight FIFO` (❶ in Figure 4). For each accelerator invocation, SIMD lanes go through these weights in lock-step and perform the neural computation autonomously without engaging the other parts of the SM.

5 Controlling Quality Tradeoffs

To be able to control the quality tradeoffs, any approximation technique including ours, needs to expose a quality knob to the compiler and/or runtime system. The knob for our design is *the accelerator invocation rate*. That is the fraction of the warps that are offloaded to the neural accelerator. The rest of the warps will execute the original precise segment of code and generate exact outputs. In the default case, without any quality control, all the warps that contain the approximable segment will go through the neural accelerator which translates to 100% invocation rate. With quality control, only a fraction of the warps will go through the accelerator. Naturally, the higher the invocation rate, the higher the benefits and the lower the quality.

For a given quality target, the compiler predetermines the invocation rate by examining the output quality loss on a held-out evaluation input dataset. Starting from 100% invocation rate, the compiler gradually reduces the invocation rate until the quality loss is less than the quality target. During runtime, a quality monitor, similar to the one proposed in SAGE [6], stochastically checks the output quality of the application and adjusts the invocation rate.

We also investigated a more sophisticated approach that uses another neural network to filter out invocations of the accelerator that significantly degrade quality. The empirical study suggested that the simpler approach of reducing the invocation rate provides similar benefits.

6 Evaluation

We evaluate the benefits of the proposed architecture across different bandwidth and accelerator settings. We use a diverse set of applications, cycle-accurate simulation, logic synthesis, and consistent detailed energy modeling.

6.1 Applications and Neural Transformation

Applications. As Table 1 shows, we use a diverse set of *approximable* GPU applications from the Nvidia SDK [23] and Rodinia [24] benchmark suites to evaluate the integration of neural accelerators within GPU architectures. We added three more applications to the mix from different sources [25–27]. As shown, the benchmarks represent workloads from finance, machine learning, image processing, vision, medical imaging, robotics, 3D gaming, and numerical analysis. We did not reject any benchmarks due to their performance, energy, or quality shortcomings.

Annotations. As described in Section 2.1, the CUDA source code for each application is annotated using the #pragma directives. We use theses directives to delineate a region within a CUDA kernel that has fixed number of inputs/outputs and is safe to approximate. Although it is possible and may boost the benefits to annotate multiple regions, we only annotate one region that is easy to identify and is frequently executed. We did not make any algorithmic changes to enable neural acceleration.

As illustrated by the numbers of function calls, conditionals, and loops in Table 1, these regions exhibit a rich and diverse control flow behavior. For instance, the target region in inversk2j has three loops and five conditionals. Other regions similarly have several loops/conditionals and function calls. Among these applications, the region in jmeint has the most complicated control flow with 37 if/else statements. The regions are also diverse in size and vary from small (binarization with 27 PTX instructions) to large (jmeint with 2,250 PTX instructions).

Evaluation/training datasets. As illustrated in Table 1, the datasets that are used for measuring the quality, performance, and energy are completely disjoint from the ones used for training the neural networks. The training inputs are typical representative inputs (such as sample images) that can be found in application test suites. For instance, we use the image of lena,

Table 1: Applications, accelerated regions, training and evaluation datasets, quality metrics, and approximating neural networks.

	Description	Source	Domain	Quality Metric	# of Function Calls	# of Loops	# of ifs/ elses	# of PTX Insts.	Training Input Set	Evaluation Input Set	Digital NPU	
											Neural Network Topology	Quality Loss
binarization	Image binarization	Nvidia SDK	Image Processing	Image Diff	1	0	1	27	Three 512x512 pixel images	Twenty 2048x2048 pixel images	3 -> 4 -> 2 -> 1	8.23%
blackscholes	Option pricing	Nvidia SDK	Finance	Avg. Rel. Error	2	0	0	96	8,192 options	262,144 options	6 -> 8 -> 1	4.35%
convolution	Data filtering operation	Nvidia SDK	Machine Learning	Avg. Rel. Error	0	2	2	886	8,192 data points	262,144 data points	17 -> 2 -> 1	5.25%
inversek2j	Inverse kinematics for 2-joint arm	CUDA-Based Kinematics	Robotics	Avg. Rel. Error	0	3	5	132	8,192 2D coordinates	262,144 2D coordinates	2 -> 16 -> 3	8.73%
jmeint	Triangle intersection detection	jMonkey Game	3D Gaming	Miss Rate	4	0	37	2,250	8,192 3D coordinates	262,144 3D coordinates	18 -> 8 -> 2	17.32%
laplacian	Image sharpening filter	Nvidia SDK	Image Processing	Image Diff	0	2	1	51	Three 512x512 pixel images	Twenty 2048x2048 pixel images	9 -> 2 -> 1	6.01%
meanfilter	Image smoothing filter	Nvidia SDK	Machine Vision	Image Diff	0	2	1	35	Three 512x512 pixel images	Twenty 2048x2048 pixel images	7 -> 4 -> 1	7.06%
newton-raph	Newton-Raphson equation solver	Likelihood Estimators	Numerical Analysis	Avg. Rel. Error	2	2	1	44	8,192 cubic equations	262,144 cubic equations	5 -> 2 -> 1	3.08%
sobel	Edge detection	Nvidia SDK	Image Processing	Image Diff	0	2	1	86	Three 512x512 pixel images	Twenty 2048x2048 pixel images	9 -> 4 -> 1	5.45%
srad	Speckle reducing anisotropic diffusion	Rodinia	Medical Imaging	Image Diff	0	0	0	110	Three 512x512 pixel images	Twenty 2048x2048 pixel images	5 -> 4 -> 1	7.43%

peppers, and mandrill for applications that operate on image data. Since the regions are frequently executed, even a single application input provides large number of training data. For example, in sobel a 512×512 pixel image generates 262,144 training data elements.

Neural networks. The "Neural Network Topology" column shows the topology of the neural network that replaces the region of code. For instance, the topology for blackscholes is $6 \rightarrow 8 \rightarrow 1$. That is the neural network has 6 inputs, one hidden layer with 8 neurons, and 1 output neuron. These topologies are automatically discovered by our compiler and we use the 10-fold cross validation technique to train the neural networks. As the results suggest, different applications require different topologies. Therefore, the SM architecture should be changed in a way that is reconfigurable and can accommodate different topologies.

Quality. We use application-specific quality metrics, shown in Table 1, to assess the quality of each application's output after neural acceleration. In all cases, we compare the output of the original precise application to the output of the neurally accelerated application. For blackscholes, inversek2j, newton-raph, and srad that generate numeric outputs, we measure the average relative error. For jmeint that determines whether two 3D triangles intersect, we report the misclassification rate. The convolution, binarization, laplacian, meanfilter, and sobel that produce image outputs, we use the average root-mean-square image difference. In Table 1, the "Quality Loss" column reports the whole-application quality degradation based on the above metrics. This loss includes the accumulated errors due to repeated execution of the approximated region. The quality loss in Table 1 represents the case where all of the dynamic threads with the safe-to-approximate region are neurally accelerated.

Even with 100% invocation rate, the quality loss with neural acceleration is less than 10% except in the case of jmeint. The jmeint application's control flow is very complex and the neural network is not able to capture all the corner cases to achieve below 10% quality loss. These results are commensurate with prior work on CPU-based neural acceleration [11, 13]. Prior work on GPU approximation such as SAGE [6] and Paraprox [7] reports similar quality losses in the default setting. EnerJ [16] and Truffle [28] show less than 10% loss for some applications and even 80% loss for others. Green [29] and loop perforation [30] show less than 10% error for some applications and more than 20% for others. Later, we will discuss how to use the invocation rate to control the quality tradeoffs, and achieve even lower quality loss when desired.

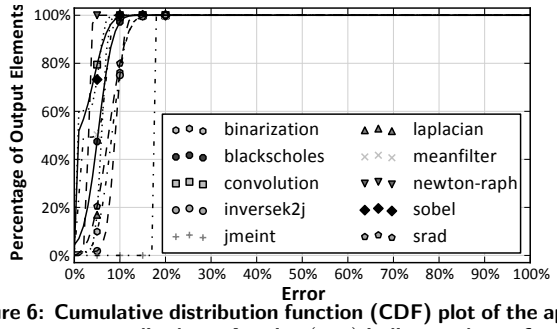


Figure 6: Cumulative distribution function (CDF) plot of the applications output quality loss. A point (x, y) indicates that y fraction of the output elements see quality loss less than or equal to x.

To better illustrate the application quality loss, Figure 6 shows the Cumulative Distribution Function (CDF) plot of the final quality loss for each element of the output. Each application output is a collection of elements – an image consists of pixels; a vector consists of scalars; etc. The loss CDF shows the distribution of output quality loss among the output elements and shows that very few output elements see a large loss. As shown, the majority of output elements (from 78% to 100%) see a loss less than 10%

Table 2: GPU microarchitectural parameters.

System Overview: No. of SMs: 15, Warp Size: 32 threads/warp; **Shader Core Config**: 1.4 GHz, GTO scheduler [35], 2 schedulers/SM; **Resources / SM**: No. of Warps: 48 Warps/SM, No. of Registers: 32,768; **Interconnect**: 1 crossbar/direction (15 SMs, 6 MCs), 700 MHz; **L1 Data Cache**: 16KB, 128B line, 4-way, LRU; **Shared Memory**: 48KB, 32 banks; **L2 Unified Cache**: 768KB, 128B line, 16-way, LRU; **Memory**: 6 GDDR5 Memory Controllers, 924 MHz, FR-FCFS [36]; **Bandwidth**: 177.4 GB/sec.

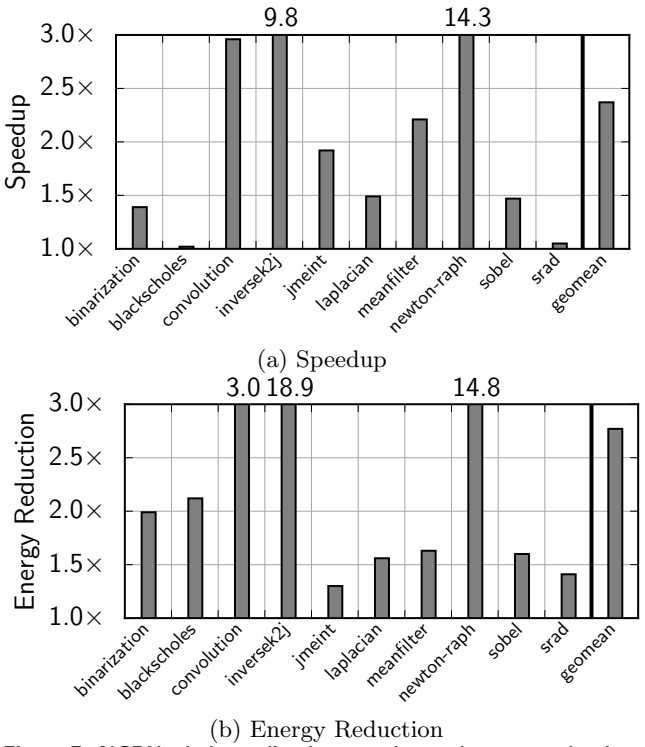
6.2 Experimental Setup

Cycle-accurate simulations. We use the GPGPU-Sim cycle-accurate simulator version 3.2.2 [31]. We modified the simulator to include our ISA extensions and include the extra microarchitectural modifications necessary for the integration of neural accelerators within GPUs. The overhead of ISA extensions that communicate with the accelerator are modeled. For baseline simulations that do not include any approximation or acceleration, we use the unmodified GPGPU-Sim. We use one of the GPGPU-Sim's default configurations that closely models the Nvidia GTX 480 chipset with Fermi architecture. Table 2 summarizes the microarchitectural parameters of the chipset. We run the applications to completion. We use NVCC 4.2 with -O3 to enable aggressive compiler optimizations. Moreover, we optimize the number of thread blocks and number of threads-per-block of each kernel for the simulated hardware.

Energy modeling and overheads. To measure GPU energy, we use GPUWattch [32], which is integrated with GPGPU-Sim. To measure the accelerator energy, we also generate its event log during the cycle-accurate simulations . Our energy evaluations use a 40 nm process node and 1.4 GHz clock frequency. Neural acceleration requires the following changes to the SM and SIMD lanes and are modeled using McPAT [33] and CACTI 6.5 [34]. In each SM, we add a 2 KB weight FIFO. The extra input/output FIFOs are 256 bytes per SIMD lane. The sigmoid LUT which is added to each SIMD lane contains 2048 32-bit entries. Since GPUWattch also uses McPAT and CACTI, our added energy models, which use the same tools, provide a unified and consistent framework for energy measurement.

6.3 Experimental Results

Performance and energy benefits. Figure 7a shows the whole application speedup when all the invocations of the approximable region are accelerated with the neural accelerator. The remaining part (i.e., the non-approximable region) is executed normally. The results are normalized to the baseline where the entire application is executed on the GPU with no acceleration. The highest speedup is observed for newton-raph (14.3×) and inversek2j (9.8×), where the bulk of execution time is spent on approximable parts (see Figure 1). The lowest speedup is observed for blackscholes and srad (about 2% and 5%) which are bandwidth-hungry applications. While a considerable fraction of the execution time in blackscholes and srad is spent in the approximate region (See Figure 1), the speedup of accelerating these two applications is modest. That is because these applications use most of the off-chip bandwidth, even when they run on GPU (without acceleration). Due to bandwidth limitation, neural acceleration cannot reduce the execution time. Next, we study the effect of increasing



(a) Speedup

(b) Energy Reduction

Figure 7: NGPU whole application speedup and energy reduction.

the off-chip bandwidth on these two applications and show that with reasonable improvement in bandwidth, even these benchmarks observe significant benefits. On average, the evaluated applications see a 2.4× speedup through neural acceleration.

Figure 7b shows the energy reduction for each benchmark as compared to the baseline where the whole benchmark is executed on GPU. Similar to the speedup, the highest energy saving is achieved for inversek2j (18.9×) and newton-raph (14.8×), where bulk of the energy is consumed for the execution of approximable parts (see Figure 1). The lowest energy saving is obtained on jmeint (30%) since for this application, the fraction of energy consumed on approximable parts is relatively small (See Figure 1). On average, the evaluated applications see a 2.8× reduction in energy usage.

The quality loss when all the invocations of the approximable region get executed on neural accelerators (i.e., the highest quality loss) is shown in Table 1 (labeled Quality Loss). We study the effects of our quality control mechanism for trading off performance and energy savings for better quality later in this section.

Area overhead. To estimate the area overhead, we synthesize the sigmoid unit using Synopsys Design Compiler and NanGate 45 nm Open Cell library, targeting the same frequency as the SMs. We extract the area of the buffers and FIFOs from CACTI. Overall, the added hardware requires about 0.27 mm^2. We estimate the area of the SMs by inspecting the die photo of GTX 480 that implements the Fermi architecture. Each SM is about 22 mm^2 and the die area is 529 mm^2 with 15 SMs. The area overhead per SM is approximately 1.2% and the total area overhead is 0.77%. The low area overhead is because our architecture uses the same ALUs that are already available in each SIMD lane, shares the weight

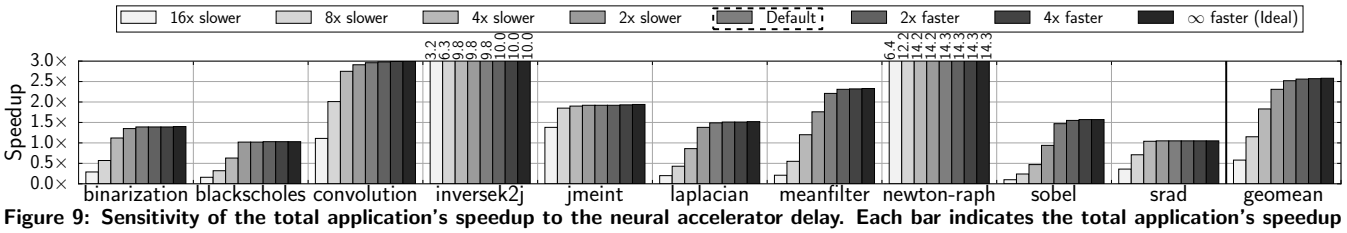


Figure 9: Sensitivity of the total application's speedup to the neural accelerator delay. Each bar indicates the total application's speedup when the neural accelerator delay is altered by different factors. The default delay for neural accelerator varies from one application to the other and depends on the neural network topology trained for that application. The ideal case (∞ faster) shows the total application speedup when neural accelerator has zero delay.

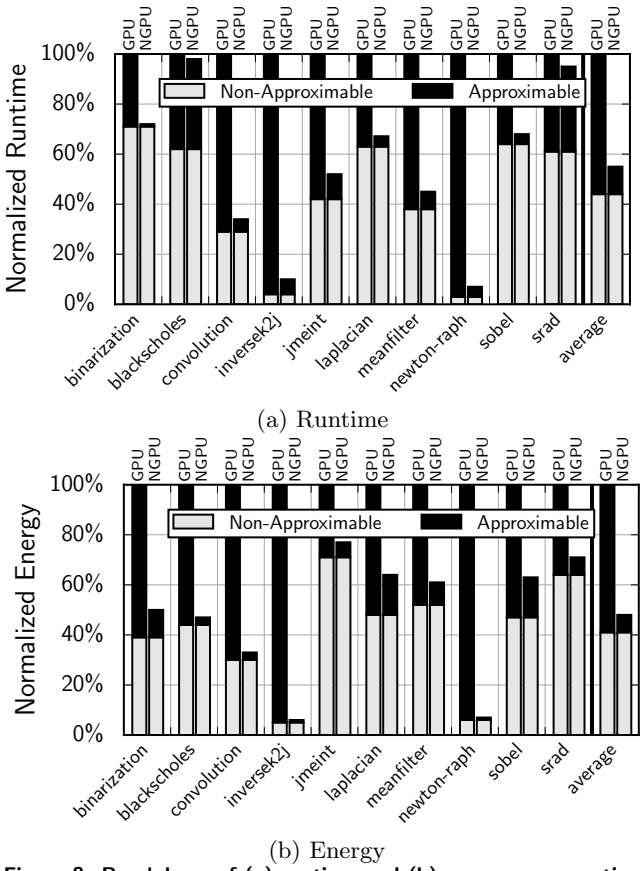


Figure 8: Breakdown of (a) runtime and (b) energy consumption between non-approximable and approximable regions normalized to the runtime and energy consumption of the GPU, respectively. For each application, the first (second) bar shows the normalized value when the application is executed on the GPU (NGPU).

buffer across the lanes, and implements the sigmoid unit as a read-only lookup table, enabling the synthesis tool to optimize its area. This low area overhead confirms the scalability of our design.

Opportunity for further improvements. To explore the opportunity for further improving the execution time by making the neural accelerator faster, Figure 8a shows the time breakdown of approximable and non-approximable parts of applications when applications run on GPU (no acceleration) and NGPU (neurally accelerated GPU), normalized to the case where the application runs on GPU (no acceleration). As Figure 8a depicts, NGPU is effective at reducing the time that is spent on approximable parts for all but two applications: blackscholes and srad. These two applications use most of the bandwidth of the GPU, and

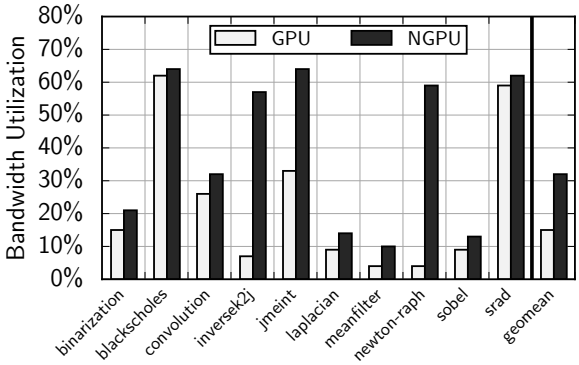


Figure 10: Memory bandwidth consumption when the applications are executed on GPU (first bar) and NGPU (second bar).

consequently, do not benefit from the accelerators due to the bandwidth wall. The rest of the applications significantly benefit from accelerators. On some applications (e.g., binarization, laplacian, and sobel), the execution time of approximable parts on NGPU is significantly smaller than the execution time of the non-approximable parts. Hence, no further benefits are possible with faster accelerators. For the rest of the applications, the execution time of approximable parts on NGPU, although considerably reduced, is comparable to and sometimes exceeds (e.g., inversek2j) the execution time of non-approximable parts. Thus, there is a potential to further speed these applications up with faster accelerators.

We similarly study the opportunity to further reduce the energy usage with more energy-efficient accelerators. Figure 8b shows the energy breakdown between approximable and non-approximable parts when applications run on GPU and NGPU, normalized to the case where the application runs on GPU. These results clearly shows that neural accelerators are effective in reducing the energy usage of applications when executing the approximable parts. For many of the applications, the energy that is consumed for running approximable parts is modest as compared to the energy that is consumed for running the non-approximable parts (e.g., blackscholes, convolution, jmeint, etc.). For these applications, a more energy-efficient neural accelerator may not bring further energy savings. However, there are some applications, such as binarization, laplacian, and sobel, for which the fraction of energy that is consumed on neural accelerators is comparable to the fraction of energy consumed on non-approximable parts. For these applications further energy saving is possible with a more energy-efficient implementation of neural accelerators (e.g., analog neural accelerators [11]).

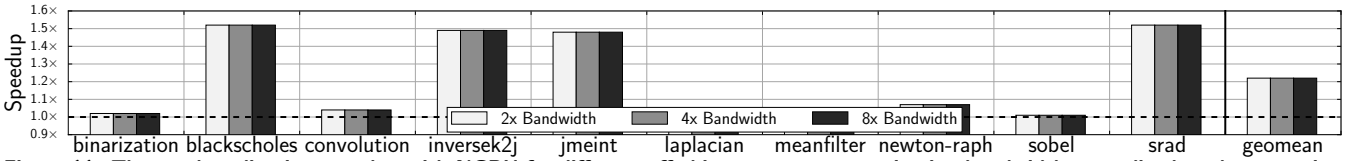


Figure 11: The total application speedup with NGPU for different off-chip memory communication bandwidth normalized to the execution with NGPU with default bandwidth. The default bandwidth is 177.4 GB/s.

Sensitivity to accelerator speed. To study the effects of accelerators' speed on performance gains, we vary the latency of neural accelerators and measure the overall speedup as shown in Figure 9. We decrease the delay of the default accelerators by a factor of 2 and 4 and also include an ideal neural accelerator with zero latency. Moreover, we show the speedup numbers when the latency of the default accelerators increases 2×, 4×, 8× and 16×. Unlike Figure 8a that suggests performance improvement for some applications by benefiting from faster accelerators, Figure 9 shows virtually no speedup benefit by making neural accelerators faster beyond what they offer in the default design. Even making accelerators slower by a factor of two does not considerably change the speedup. Slowing down the accelerators by a factor of four, many applications observe performance loss. (e.g., laplacian).

To explain this behavior, Figure 10 shows the bandwidth usage of GPU and NGPU across all applications. While on the baseline GPU, only two applications use more than 50% of the off-chip bandwidth (i.e., blackscholes and srad), on NGPU, many applications use more than 50% of their off-chip bandwidth (e.g., inversek2j, jmeint, and newton-raph). As applications run faster with accelerators, the rate at which they access data increases, which puts pressure on off-chip bandwidth. This phenomena shifts the bottleneck of execution time from computation to data delivery. *As computation is no longer the major bottleneck after acceleration, speeding up thread execution beyond a certain point has marginal effect on the overall execution time.* Even increasing the accelerator speed by a factor of two (e.g., by adding more multiply-and-add units) has marginal effect on execution time. We leverage this insight to simplify the accelerator design and reuse available ALUs in the SMs as described is Section 4.1.

Sensitivity to off-chip bandwidth. To study the effect of off-chip bandwidth on the benefits of NGPU, we increase the off-chip bandwidth up to 8× and report the performance numbers. Figure 11 shows the speedup of NGPU with 2×, 4×, and 8× bandwidth over the baseline NGPU (i.e., 1× bandwidth) across all benchmarks. As NGPU is bandwidth limited for many applications (See Figure 10), we expect a considerable improvement in performance as the off-chip bandwidth increases. Indeed, Figure 11 shows that bandwidth-hungry application (i.e., blackscholes, inversek2j, jmeint, and srad) observe speedup of 1.5× when we double the off-chip bandwidth. After doubling the off-chip bandwidth, no application remains bandwidth limited, and therefore, increasing the off-chip bandwidth to 4× and 8× has little effect on performance. It may be possible to achieve, the 2× extra bandwidth by using data compression [37] with little changes to the architecture of existing GPUs.

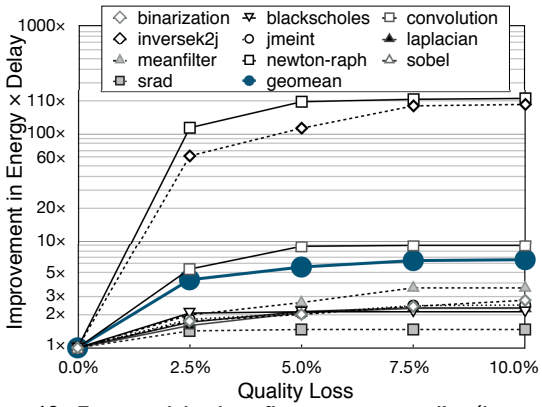


Figure 12: Energy×delay benefits vs output quality (log scale).

While technologies like 3D DRAM that offer significantly more bandwidth (and lower access latency) can be useful, they are not necessary for providing the off-chip bandwidth requirements of NGPU for the range of applications that we studied. However, even without any of these likely technology advances (compression or 3D stacking), the NGPU provides significant benefits across most of the applications.

Controlling quality tradeoffs. To study the effect of our quality control mechanism, Figure 12 shows the energy-delay product of NGPU normalized to the energy-delay product of the baseline GPU (without acceleration) when the output quality loss changes from 0% to 10%. The quality control mechanism enables navigating the tradeoff between the quality loss and the gains. All applications see declines in benefits when invocation rate decreases (i.e., output quality improves). Due to the Amdahl's Law effect, the applications that spend more than 90% of their execution in the approximable segment (inversek2j and newton-raph), see larger declines in benefits when invocation rate decreases. However, even with 2.5% quality loss, the average speedup is 1.9× and the energy savings is 2.1×.

Comparison with prior CPU neural acceleration. Prior work [10] has explored improving CPU performance and efficiency with NPUs. Since NPUs offer considerably higher performance and energy efficiency with CPUs, we compare our NGPU proposal to CPU+NPU and GPU+NPU. For the evaluation, we use MARSSx86 cycle-accurate simulator for the single-core CPU simulations with a configuration that resembles Intel Nehalem (3.4 GHz with 0.9 V at 45 nm) and is the same as the setup used in the most recent NPU work [11].

Figure 13 shows the application speedup and energy reduction with CPU, GPU, GPU+NPU, and NGPU over CPU+NPU. Even without using neural acceleration, GPU provides significant performance and efficiency benefits over NPU-accelerated CPU by leveraging data level parallelism. GPU offers 5.6× average speedup and 3.9×

average energy reduction compared to CPU+NPU. A GPU enhanced with our proposal (NGPU) increases the average speedup and energy reduction to 13.2× and 10.8×, respectively. Moreover, as GPUs already exploit data-level parallelism, our proposal offers virtually the same speedup as the area-intensive GPU+NPU. However, accelerating GPU with the NPU design imposes 31.2% area overhead while our NGPU imposes 1.2%. GPU with area-intensive NPU (GPU+NPU) offers 17.4% less energy benefits compared to NGPU mostly due to more leakage. In summary, our proposal offers the highest level of performance and energy efficiency across the examined benchmarks with the modest area overhead of approximately 1.2% per SM.

7 Related Work

Recent work has explored a variety of approximation techniques that include: (a) approximate storage designs [38, 39] that trades quality of data for reduced energy [38] and longer lifetime [39], (b) voltage overscaling [28, 40, 41], (c) loop perforation [30, 42, 43], (d) loop early termination [29], (e) computation substitution [6, 9, 29, 44], (f) memoization [7, 8, 45], (g) limited fault recovery [42, 46–50], (h) precision scaling [16, 51], (i) approximate circuit synthesis [19, 52–57], and (j) neural acceleration [10–15].

This work falls in the last category; yet, we exclusively focus on the integration of neural accelerators within GPU throughput processors. The prior work on neural acceleration mostly focuses on single-threaded CPU code acceleration by either loosely coupled neural accelerators [12–14, 58, 59] or tightly-coupled ones [10, 11]. Grigorian et al. study the effects of eliminating control-flow divergence by converting SIMD code to software neural networks with no hardware support [15]. However, prior work does not explore tight integration of neural hardware in throughput processors; and does not study the interplay of data parallel execution and hardware neural acceleration. Prior to this work, the benefits, limits, and challenges of integrating hardware neural acceleration within GPUs for many-thread data-parallel applications was unexplored.

There are several other approximation techniques in the literature that can or have been applied to GPU architectures. Loop perforation [30] periodically skips loop iteration for gains in performance and efficiency. Green [29] terminates loops early or substitute compute intensive functions with simpler, lower quality versions that are provided by the programmer. Relax [46] is compiler/architecture system for suppressing hardware fault recovery in approximable regions of code, exposing these errors to the application. Fuzzy memoization forgoes invoking a floating point unit if the inputs are in the neighborhood of previously seen inputs. The results of the previous calculation is reused as an approximate result. Arnau et al. use hardware memoization to reduce redundant computation in GPUs [8]. Sartori et al. propose a technique that mitigates branch divergence by forcing the divergent threads to execute the most popular path [9]. In case of memory divergence, they force all the threads to access the most commonly demanded memory block. SAGE [6] and Paraprox [7] perform compile-time static code transformations on GPU kernels that include data compression, profile-directed memoization, thread fusion, and atomic operation optimization. Our quality control mechanism takes inspiration from the quality control in these two works.

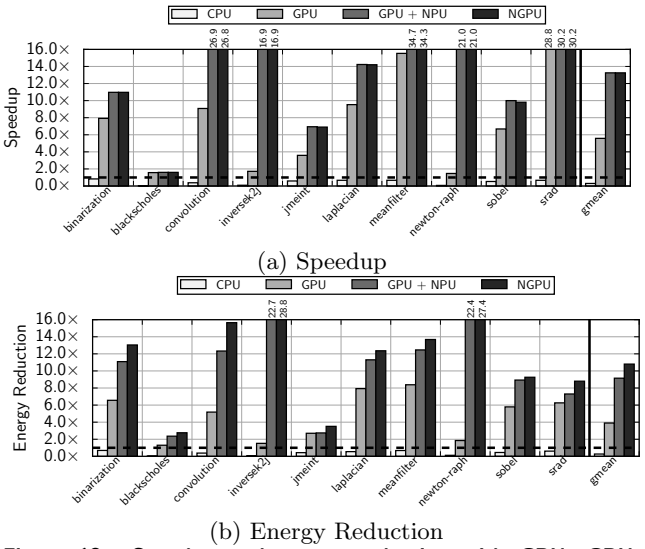


Figure 13: Speedup and energy reduction with CPU, GPU, GPU+NPU, and NGPU.(The baseline is CPU+NPU, which is a CPU augmented with a NPU accelerator [10]).

In contrast, we describe a hardware approximation technique that integrates neural accelerators within the pipeline of the GPU cores. In our design, we aim at minimizing the pipeline modifications and utilizing existing hardware components. Distinctively, our work explores the interplay between data parallelism and neural acceleration and studies its limits, challenges, and benefits.

8 Conclusion

Many of the emerging applications that can benefit from GPU acceleration are amenable to inexact computation. We exploited this opportunity by integrating an approximate form of acceleration, neural acceleration, within GPU architectures. Our neurally accelerated GPU architecture, provides significant performance and efficiency benefits while providing reasonably low hardware overhead (1.2% area overhead per SM). The quality control knob and mechanism also provided a way to navigate the tradeoff between the quality and the benefits in efficiency and performance. Even with as low as 2.5% quality loss, our neurally accelerated GPU architecture (NGPU) provides average speedup of 1.9× and average energy savings of 2.1×. These benefits are more than 10× in several cases. These results suggest that *hardware* neural acceleration for GPU throughput processors can be a viable approach to significantly improve their performance and efficiency.

9 Acknowledgments

This work was supported by a Qualcomm Innovation Fellowship, NSF award CCF#1553192, Semiconductor Research Corpo. contract #2014-EP-2577, and a gift from Google.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling,"

in *ISCA*, 2011.
[2] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, 2011.
[3] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.
[4] J. Gantz and D. Reinsel, "Extracting value from chaos." `http://www.emc.com`.
[5] "GeForce 400 series." `http://en.wikipedia.org`, 2015.
[6] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: self-tuning approximation for graphics engines," in *MICRO*, 2013.
[7] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.
[8] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," ISCA, 2014.
[9] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *Multimedia, IEEE Transactions on*, 2013.
[10] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.
[11] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.
[12] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *HPCA*, 2015.
[13] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable socs via neural acceleration," in *HPCA*, 2015.
[14] L. McAfee and K. Olukotun, "EMEURO: A framework for generating multi-purpose accelerators via deep learning," in *CGO*, 2015.
[15] B. Grigorian and G. Reinman, "Accelerating divergent applications on SIMD architectures using neural networks," in *ICCD*, 2014.
[16] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.
[17] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.
[18] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "Flexjava: Language support for safe and modular approximate programming," in *FSE*, 2015.
[19] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Language support for approximate hardware design," in *DATE*, 2015.
[20] J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *POPL*, 1979.
[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *PDP*, 1986.
[22] "Whitepaper: NVIDIA Fermi." `http://www.nvidia.com`.
[23] "NVIDIA corporation. NVIDIA CUDA SDK code samples." `http://www.nvidia.com`.
[24] 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 *IISWC*, 2009.
[25] jMonkeyEngine, 2015.
[26] O. A. Aguilar and J. C. Huegel, "Inverse kinematics solution for robotic manipulators using a cuda-based parallel genetic algorithm," *AAI*, 2011.
[27] M. Creel and M. Zubair, "A high performance implementation of likelihood estimators on gpus," in *CES*, 2013.
[28] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.
[29] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.
[30] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.
[31] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.
[32] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in gpgpus," in *ISCA*, 2013.
[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[34] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.
[35] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.
[36] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *Archit. News*, 2000.
[37] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: Enabling efficient data compression," in *ISCA*, 2015.
[38] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *ASPLOS*, 2011.
[39] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *MICRO*, 2013.
[40] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *DATE*, 2006.
[41] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.
[42] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.
[43] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou, "Patterns and statistical analysis for understanding reduced resource computing," in *Onward!*, 2010.
[44] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," in *PLDI*, 2009.
[45] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, 2005.
[46] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.
[47] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *HPCA*, 2007.
[48] X. Li and D. Yeung, "Exploiting application-level correctness for low-cost fault tolerance," *J. Instruction-Level Parallelism*, 2008.
[49] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in *SELSE*, 2009.
[50] Y. Fang, H. Li, and X. Li, "A fault criticality evaluation framework of digital systems for error tolerant video applications," in *ATS*, 2011.
[51] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.
[52] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of approximate sequential circuits," in *DATE*, 2014.
[53] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *DAC*, 2012.
[54] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.
[55] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014.
[56] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Synthesizing parsimonious inexact circuits through probabilistic design techniques," *ACM Trans. Embed. Comput. Syst.*, 2013.
[57] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *CF*, 2012.
[58] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, 2013.
[59] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *ASP-DAC*, 2014.