Accelerating Divergent Applications on SIMD Architectures Using Neural Networks

BEAYNA GRIGORIAN and GLENN REINMAN, University of California, Los Angeles

The purpose of this research is to find a neural-network-based solution to the well-known problem of branch divergence in Single Instruction Multiple Data (SIMD) architectures. Our approach differs from existing techniques that handle branch (or control-flow) divergence, which use costly hardware modifications, low-utilization masking techniques, or static prediction methods. As we examine divergent applications, we characterize the degree of data-dependent control flow seen in each and isolate the code regions (or “kernels”) that cause the most performance degradation due to branch divergence. We then train neural networks (NNs) offline to approximate these kernels and inject the NN computations directly into the applications as substitutes for the kernels they approximate. This essentially translates control flow into nondivergent computation, trading off precision for performance. As our methodology manipulates application source code directly, it is inherently platform agnostic and can be adopted as a general means for accelerating divergent applications on data-parallel architectures. In this article, we present the **Neural-izer**, an automated software flow for kernel identification, NN training, and NN integration, as well as supplementary user-controlled optimization techniques. Evaluating our approach on a variety of divergent applications run on a Graphics Processing Unit (GPU), we on average achieve performance gains of 13.6× and energy savings of 14.8× with 96% accuracy.

Categories and Subject Descriptors: C.1.2 [Architectures]: Parallel Architectures—*Single instruction, multiple data*

General Terms: Design, Performance

Extension of the Following Conference Paper:

Beayna Grigorian and Glenn Reinman. 2014. Accelerating Divergent Applications on SIMD Architectures Using Neural Networks. *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD)*. 317–323.

New Material Provided in this Journal Article:

- (1) Expanded version of abstract;
- (2) More complete version of introduction, which includes additional background information on branch divergence and artificial neural networks;
- (3) New motivational case study on amount of divergent control flow in “triangle intersection detection” application;
- (4) New motivational case study on impact of divergence on performance of Newton-Raphson method, as well as potential benefits to be gained from neural network approximations;
- (5) New subsection on “Details of NN Integration” added to methodology section;
- (6) Additional information on evaluation metrics provided in results section;
- (7) New results on varying kernel scope;
- (8) New results on varying neural network topology.

This research is supported by the NSF Expedition in Computing Award # CCF-0926127, by the NSF Graduate Research Fellowship Grant # DGE-0707424, and by C-FAR (one of six centers of STARnet, an SRC program sponsored by MARCO and DARPA).

Authors’ addresses: B. Grigorian and G. Reinman, 4731G Boelter Hall, University of California, Los Angeles (UCLA), Los Angeles, CA 90095; emails: {bgrigori, reinman}@cs.ucla.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/03-ART2 \$15.00

DOI: <http://dx.doi.org/10.1145/2717311>

Additional Key Words and Phrases: Branch divergence, approximate computing, neural networks, hardware acceleration

ACM Reference Format:

Beayna Grigorian and Glenn Reinman. 2015. Accelerating divergent applications on SIMD architectures using neural networks. *ACM Trans. Architect. Code Optim.* 12, 1, Article 2 (March 2015), 23 pages. DOI: <http://dx.doi.org/10.1145/2717311>

1. INTRODUCTION

Single Instruction Multiple Data (SIMD) architectures [Seiler et al. 2008; Gschwind 2006; Lomont 2011; Lindholm et al. 2008; Lee et al. 2011] are well known for their high performance and efficiency in executing data-parallel computation. They are generally considered to be more area efficient than Multiple Instruction Multiple Data (MIMD) architectures for particular applications, as they amortize the cost of instruction processing over multiple datapaths [Meng et al. 2010; Kapasi 2004]. By definition, SIMD computes multiple datasets in lock-step via wide datapaths under a single control flow. However, this design choice makes it fundamentally less effective at dealing with diverse control flow, particularly when control flow is data dependent. As SIMD has become an increasingly popular architectural choice for accelerating high-performance applications, many of which feature divergent control flow, the need to address its Achilles heel of branch divergence has become more evident.

Formally, branch divergence is defined as the case where multiple channels (e.g., lanes or threads) simultaneously executing the same set of instructions arrive at a point of divergent control flow (e.g., “if-else” branch) and take different paths [Vaidya et al. 2013]. In practice, this can occur with “if-else” branches, “switch” statements, “for”/“while” loops, and so on. Applications with the SIMD divergence problem, or “divergent applications,” have been dealt with in a variety of ways in prior art. One approach is predication, in which all paths are executed sequentially and masks are used to ensure data correctness [Fung et al. 2007; Kapasi et al. 2000]. With this approach, branch divergence may at best be avoided by conversion into a data dependency. However, this results in inefficiency due to added overhead for data generation and processing, as well as reduced *useful* utilization from uncommitted results. Prior art has also investigated compiler-based optimizations [Diamos et al. 2011] and algorithmic modifications, such as data migration [Wald 2011] and restructuring of control flow [Wu et al. 2012]; however, these static techniques are not effective in all situations, such as in applications with data-dependent branches. Other architectural approaches, for instance, a large warp microarchitecture [Narasiman et al. 2011] and thread block compaction [Fung and Aamodt 2011], require complex hardware modifications that consume power and area and nevertheless experience lowered useful utilization for divergent control flow.

Unlike prior art that handles divergence by tolerating the aftermath of control flow, we approach this problem from an entirely different angle: we choose to exploit the intelligent learning capabilities of neural networks (NNs) to approximate away the control flow and, as a result, the divergence itself. Recent work [Esmailzadeh et al. 2012b] has used NNs for hardware-based acceleration; we propose to apply it in a new, software-based approach to address the divergence problem. With our methodology, we identify a potentially divergent code region (or “kernel”), train an Artificial Neural Network (ANN) offline to approximate that kernel, and inject the ANN computation directly into the code in place of the kernel. By converting the control-flow region into nondivergent, approximate computation, we **remove the divergence problem entirely as a tradeoff for precision**. Note that as our technique directly manipulates code without the need for costly hardware modifications, this is a **platform-agnostic approach**.

By performing nonlinear multivariate regression, ANNs have an innate ability to subsume computation in exchange for an inexact representation [Haykin 1998]. With the development of advanced learning techniques [Rumelhart et al. 1986] for training ANNs, these models have become more flexible than various forms of polynomial regression. The success of ANNs in modeling classification problems [Zhang 2000] also suggests they are well suited for approximating regions of control flow. However, applying an ANN to branch divergence is a complex and nontrivial process, as the effectiveness of an ANN relies on how well it adapts to fit a specific problem—this is critical for ensuring acceptable error rates for applications. For this purpose, we have developed a complete methodology, including a software flow and supplementary optimization techniques, to apply neural approximation as a solution for SIMD divergence.

This work provides the following novel insights in neural approximation: (1) with no additional hardware, transforming an application’s computation model can overcome microarchitectural limitations of SIMD designs, leading to significant performance and energy gains; and (2) divergent control-flow regions can be effectively approximated using NNs. In addition, our contributions include (a) *Neuralizer*: an end-to-end, automated software flow for identifying eligible kernels (i.e., code regions) in an application, training and evaluating NNs to approximate those kernels, and integrating the NNs into the application in place of those kernels; (b) optimization techniques customized for different types of divergence—these include heuristics (divergence profiling, score-based ranking, and scope precedence) to improve the search for ideal kernels as well as improve NN performance; and (c) detailed evaluation of our approach (including evaluation of the heuristics and the impact of NN topology) on a Graphics Processing Unit (GPU) across a range of divergent applications from various domains.

The rest of this article is organized as follows: Section 2 motivates the need for addressing SIMD divergence and highlights the power of an NN-based solution. Section 3 characterizes the kernels we are targeting, while Section 4 describes our overall methodology. We present our evaluation approach and results in Section 5. Prior art is reviewed in Section 6. And we conclude with Section 7.

2. MOTIVATION

Branch divergence can have a significant impact on the performance of an SIMD architecture. With GPUs, for instance, this problem is often referred to as “warp divergence.” In this case, threads clustered in a given warp are subject to execution of any path followed by any thread in the warp; masks are then used to invalidate unintended results and maintain program correctness. This process results in excessive computation. Recent performance studies on GPUs have shown that as the number of divergent threads in warps is increased, the overhead of divergent control flow increases linearly [Che et al. 2008].

In Figure 1, we provide an example of warp divergence, where a warp of eight threads comes across a data-dependent “switch” statement (shown in the left half of the figure). The switch statement has three possible outcomes (i.e., three cases), each of which invokes a particular function (A, B, or C). The right half of the figure reveals the specificity of the threads to the cases of the switch statement, where a large degree of divergence is demonstrated: three of the threads fall within the first case, one thread falls within the second case, and four threads fall within the third (default) case. Due to the microarchitectural limitations imposed by the GPU, all three of the functions are computed by all threads, while the majority of results are masked away. Notably, while all threads execute function B, only one thread will actually make use of these results. If this one thread were to not fall within the second case of the switch statement, none of the threads would need to execute function B, resulting in less divergence-induced performance degradation. We now briefly examine two motivational case studies.

Fig. 1. Example of warp divergence in a GPU.

	Kernel Scope (Level of Branch Nesting)	% of Program Execution Time (Varies Based on Data)
	1st Level	10.85% – 38.51%
	2nd Level	11.58% – 27.23%
	3rd Level	13.42% – 26.90%
	4th Level	7.35% – 64.15%

Fig. 2. Pseudocode and profiling results for triangle intersection algorithm (jmeint).

2.1. Case Study I: Triangle Intersection

The triangle intersection algorithm used in graphics applications is an example of practical code subject to heavy control flow. It takes as input the vertices of two triangles and determines if they intersect. The pseudocode for the algorithm (based on the jmeint benchmark described in Section 5.1) is shown in Figure 2. Within this pseudocode, the

Fig. 3. Impact of divergence in Newton-Raphson algorithm.

number of floating point operations (*FP Ops*) and comparisons (*Compares*) is specified on the right-hand side of each line of computational code. Also, the right-hand side of the figure lists the portions of execution time (*% of Program Execution Time*) consumed by each of the four highlighted kernel scopes. The percentages of execution time are presented as ranges based on a series of randomized inputs, revealing the breakdown of execution time to vary drastically depending on the nature of the data. Overall, we observe that a series of significant chunks of computation are segregated by nonpredictable, data-dependent control flow. As the divergence is data dependent, it is not possible to format or reorganize the data a priori to avoid it. Static branch resolution and precomputation, therefore, become invalid options. Alternatively, if one were to break up the kernel into multiple subkernels based on points of divergence, this would result in (1) overhead from binning intermediate values, (2) loss in data locality, and (3) overhead from sorting output data of each subkernel and reinitializing computation for subsequent subkernels. As such, the inadequacies of these existing solutions call for a better approach to overcoming branch divergence.

2.2. Case Study II: Newton-Raphson Method

The Newton-Raphson method [Ortega and Rheinboldt 1970] is an iterative approach for finding roots of an equation. The algorithm starts from an initial guess provided by the user and iteratively refines the solution until either it finds an exact root of the given equation or it is able to provide an approximation of the root within tolerable error bounds. While it is a powerful approximation-based approach, this algorithm is not guaranteed to converge and return an acceptable result. In other words, when there is *algorithmic divergence*, the program continues processing for some number of iterations before determining it cannot converge to an acceptable result and consequently returns a failure indicator. The four main cases of *algorithmic divergence* for the Newton-Raphson method are (1) divergence at inflection points, (2) zero-valued derivatives (leading to division by zero), (3) oscillations near local maxima or minima, and (4) root jumping (e.g., for oscillating functions with many roots). These cases result from both the characteristics of the equations themselves and the initial guess inputted to the algorithm. To demonstrate the impact of divergence on GPU performance, we have taken the Newton-Raphson method and generated synthetic datasets for it, where we intentionally include a precise amount of data that will forestall the algorithm from converging to an acceptable result (i.e., will lead to *algorithmic divergence*). Figure 3

Fig. 4. Performance gains from approximating Newton-Raphson algorithm with NNs of different topologies.

displays the impact on GPU performance when a given percentage of the data causes *algorithmic divergence*. Notably, the performance of the GPU degrades by $11\times$, going from 0% to even 1% divergence.

To overcome this performance degradation, we introduce the use of NNs that approximate the divergent code. Since this entire algorithm is essentially one large “while” loop, we approximate the entire benchmark with a single NN and evaluate the mathematical representation of that NN (i.e., compute a series of multiplication, summation, and sigmoid operations—please refer to Section 4.3 for more information on integrating an NN into code) instead of the original control-flow-heavy Newton-Raphson algorithm. Temporarily holding off on evaluating loss of accuracy in the benchmark results, which is addressed in Section 5, we first see the potential performance benefits of neural approximation. Figure 4 demonstrates the performance gains achieved as we alter the topology of the NN. Here, an $NN_M \times N$ topology represents an NN with two hidden layers, the first with M number of neurons and the second with N —Section 4.2 provides a detailed description of the NN model. As expected, the smaller NN topologies achieve higher speedups as they require less computation to evaluate their outputs. Yet we note that even the largest topology, which has two hidden layers of 32 neurons each, achieves a $6\times$ performance gain when as little as 1% of the data leads to *algorithmic divergence*. Moreover, all the other topologies yield performance gains even when there is 0% divergence, highlighting the added benefit of reducing the amount of computation by including nondivergent code in the kernel being approximated. While this case study may be considered an extreme example of control-flow divergence, it depicts how severely SIMD performance may be impeded and demonstrates the potential impact of using neural approximation to eliminate divergence at its source.

3. KERNEL CHARACTERIZATION

In order for kernels to be properly targeted for approximation, the following criteria must be satisfied:

- **Pure Function:** no side effects (e.g., cannot modify external state)
- **Fixed-size Inputs/Outputs (I/O):** no dynamic, variable-length inputs or outputs
- **“Approximable” Region of Code:** imprecision does not result in program failure

The function purity and I/O constraints are similar to the kernel constraints imposed by other hardware acceleration schemes [Venkatesh et al. 2011; Cong et al. 2012a; Esmaeilzadeh et al. 2012b]. Additionally, since the NN will be approximating

application kernels, the “approximability” constraint means the application itself must also be able to tolerate imprecision. For example, one cannot use an NN to approximate a kernel that computes the exact value of a memory address used to access data; however, if subsequent computation on that data is, for instance, a heuristic-based feature detection algorithm from computer vision, approximation of the final result of that algorithm would be tolerable.

There is a large body of prior work [Baek and Chilimbi 2010; Chaudhuri et al. 2011; Esmaeilzadeh et al. 2012a; Cho et al. 2012; Sidiroglou-Douskos et al. 2011] that has performed in-depth examination of this topic of approximability. The majority of these works have deemed it the **responsibility of the programmer to determine which code regions are suitable for approximation**. Prior art even includes programming language support for controlling precision [Sampson et al. 2011] and verifying quantitative reliability [Carbin et al. 2013] in applications. Similar to these efforts, our work uses programmer annotations to delineate scope or code regions that should be considered approximable by the compiler.

Aside from the criteria listed previously, there are also several characteristics to consider when identifying appropriate kernels. Note that these characteristics are not criteria, meaning they are not required to be implemented into the compiler (e.g., they are not used in the compiler that generated the evaluation results presented in this article); they instead serve as valuable guidelines for the user or programmer. First, it is important for the kernel to have a relatively small number of inputs and outputs. Kernels with large numbers of inputs/outputs will not only lead to larger NN computations (and less of a performance improvement) but also require more time for the NN training. An example of a nonideal kernel, for instance, would be one that iterates over a large array and executes one trivial computation for each element. In this case, the overhead of passing the inputs through the NN would likely outweigh the benefits of the approximation. Similarly, another important characteristic is that the relation between inputs and outputs of the kernel should not be too high-dimensional. Otherwise, one would require an NN large enough to handle the approximation of that high-dimensional functionality, and consequently, the large NN would be much more difficult to train with an acceptable error rate. The reason for this is that more hidden layers are required for learning higher-dimensional functionality, yet the iterative, locally optimal algorithms commonly used for NN training, such as the back-propagation algorithm [Rumelhart et al. 1986], break down as numbers of hidden layers increase. Again, while these characteristic-based considerations we have mentioned are not hard-set rules, they can be useful for finding ideal regions to target with neural approximation.

4. METHODOLOGY

In existing acceleration schemes [Cong et al. 2012a, 2012b; Esmaeilzadeh et al. 2012b], the process of choosing which kernels to target is often conducted in an ad hoc manner and is difficult to optimize as the space of targetable kernels grows exponentially. By specifically targeting potentially divergent regions, we maintain the ability to achieve significant gains by addressing the most critical issue for SIMD architectures, yet we narrow the scope enough to allow for **automation beginning with kernel identification**. Various aspects of our software flow, the *Neuralizer*, are detailed in the following sections. To aid the process of creating effective neural approximations, we also discuss novel optimization techniques for improving ANN performance and accuracy. As previously mentioned, this overall approach is platform agnostic; however, for the purposes of this article, we examine branch divergence within the scope of a GPU. As such, our software flow leverages the CUDA-enabled ROSE [Quinlan 2000] compiler infrastructure.

Fig. 5. Overview of the *Neuralizer* software flow for automated neural acceleration of divergent applications.

4.1. Neuralizer

The *Neuralizer* is a software flow that operates offline and automates kernel identification, divergence estimation, training data collection, NN training, and NN integration, ultimately generating a set of *neuralized* versions of a given application. For this process, the programmer is only responsible for determining the approximability of the code. Figure 5 is a high-level illustration of this flow.

Static Compilation. The *neuralization* process begins by accepting an application’s source code and performing static, compiler-based analysis to identify eligible, potentially divergent kernels. At first, the compiler marks all regions of control flow as kernel candidates. Here, a “region of control flow” refers to a region of code that is encompassed by a control-flow statement (e.g., “if-else” branch, “while” loop, etc.). The compiler looks for these control-flow statements in its internally generated dataflow graph (i.e., finds nodes from which dataflow diverges) and identifies the entire control-flow region (i.e., until the dataflow once again converges) as a potential kernel of interest, or “kernel candidate.” Once it has identified kernel candidates, the compiler then eliminates candidates based on the criteria described in Section 3. Specifically, the compiler checks for purity and fixed-size I/O using internally generated data flow graphs from intermediate representation (IR) data and uses the programmer annotations to ensure approximability. After identifying eligible kernels, the compiler refactors the application’s source code, converting each kernel into a function with well-defined I/O (similar to the processing of OpenMP pragmas). The compiler also instruments the source code to enable profiler-driven probing of I/O values of the kernels. Once refactoring and instrumentation are complete, the code is compiled for the given SIMD platform.

Dynamic Profiling. The next step of the *Neuralizer* involves dynamic profiling. The profiler receives the compiled executable corresponding to the refactored, instrumented source code and runs it using real input data of the application. Leveraging the statically instrumented I/O probes, the profiler collects the I/O values of each kernel. These I/O values form each kernel’s respective dataset and are later used for training NN approximations of the given kernels. Also, this tool gauges the amount of divergence in each kernel (represented by the percentage of thread instructions that were not executed by all threads in the warp) and ranks the kernels based on decreasing amounts of divergence.

NN Training. Using the datasets collected by the dynamic profiler (one dataset per kernel), our software flow proceeds by training NNs to approximate the kernels and

(a)

Pre Kernel-to-NN Conversion	Post Kernel-to-NN Conversion
<pre> func A (x, y) { float x = rand(); float y = rand(); float z = 0; while (z < 50 && z > -50) { if (x > 0 && y < 0) z *= x + y; else z -= x * y; } return z; } </pre>	<pre> func A (x, y) { float x = rand(); float y = rand(); float z = 0; float H1 = tanh(x*W11h1 + y*W12h1); float H2 = tanh(x*W11h2 + y*W12h2); float H3 = tanh(x*W11h3 + y*W12h3); z = tanh(H1*W3o1 + H2*W2o1 + H3*W3o1); return z; } </pre>

(b)

Fig. 6. (a) Sample MLP and its mathematical representation based on labeled nodes and edge weights. (b) Example of integrating MLP of (a) into sample code (shaded regions represent the kernel-to-NN conversion).

outputs the single “best” NN for each kernel. Details regarding the modeling of NNs are provided in Section 4.2. If an excessive number of kernels have been identified, the length of the training process can be shortened by trimming the list of kernels based on the rankings provided by the dynamic profiler. For instance, in our scheme, we include the top 10 most divergent kernels.

NN Integration. The final stage of the *Neuralizer* integrates the mathematical representations of the NNs directly into the refactored source code, replacing the potentially divergent kernels they approximate, and outputs modified source code (e.g., CUDA code in this case). For details regarding the integration of NNs into applications, please refer to Section 4.3. An example of kernel-to-NN conversion is shown for sample code in Figure 6(b) using the MLP model shown in Figure 6(a).

In order to examine application-level quality-versus-performance tradeoffs, our software flow considers replacing a subset of a given application’s kernels with their corresponding NNs. Given a set of K kernels, there are 2^K possible kernel replacement combinations. Note that some kernels are nested within others, which invalidates a subset of the combinations. If there is still an overabundance of combinations, ones with low cumulative kernel rankings are eliminated. Ultimately, this software flow outputs a set of *neuralized* versions of a given application; the versions are compiled using a

standard compiler for the given SIMD architecture, and performance gains and quality loss are evaluated for each. Performance gain depends on the size of the code regions being replaced by NNs, the total amount of neurons and connections required for all NNs being used, and the amount of divergence being removed. This essentially means making a worthwhile trade as time and energy spent on divergence are substituted for time and energy spent evaluating the NN. We determine the best *neuralized* version of a given application using the following speedup-to-error ratio:

$$\text{Speedup}/(\text{Error}/\text{Error_Threshold}).$$

Here, the speedup and error values correspond to the entire *neuralized* application, not individual kernels. Since error is not the same across different applications, this ratio also takes into account a user-defined error threshold set specifically for each application. This allows a user to define an acceptable range for the quality of results of the overall application and to have this range control the kernel approximations being integrated into the application. To further augment this process, a static approach for probabilistic analysis of the approximation accuracy could also be integrated with our tool chain. This analysis would be similar to the quantitative reliability analysis performed for systems with unreliable hardware [Carbin et al. 2013], yet in this case, the unreliable components would be the NNs (i.e., the entire function for evaluating a given NN would be associated with probabilistic characteristics of accuracy) as opposed to memory regions or individual arithmetic/logical operations. Though it is not featured in this work, performing supplemental probabilistic analysis would allow our tool chain to statically enforce acceptable error rates by restricting the search space of kernels and NN topologies, thereby saving time in the training process.

Aside from requiring the programmer, who has knowledge of the code, to determine approximable regions, this software flow is entirely automated. The overhead for kernel identification and NN integration are negligible and run on the order of seconds. Also, training-data collection and divergence estimation generally require a few minutes. NN training, however, incurs relatively more overhead. Training time depends on both the number of input/output neurons and the size of the training dataset. However, since the individual code kernels are independent, as are the NN topologies being explored, the training process can be effectively parallelized. On average, training a kernel for the given NN search space using fully parallelized execution on our 2GHz Intel Quad-Core-i7 CPU has a duration of approximately 15 minutes. Additionally, since this training is an offline process, users may take advantage of libraries of pretrained approximations for commonly used kernels. In terms of longevity, NN retraining is required only when the distribution of the original training data no longer represents that of user data, resulting in unacceptable quality of results (described further in Section 4.2).

While automatically selecting combinations of multiple approximated regions is certainly a nontrivial problem, there are several ways to crop the large search space (e.g., by invalidating combinations of nested regions or regions deemed inapproximable by the programmer) in order to make the problem more tractable. Subsuming multiple regions within an NN is also discussed as one of our optimizations. As our approach targets control flow, our most powerful heuristic involves prioritization by degree of divergence, which makes a significant impact in the runtime of the software flow (e.g., saves $4\times$ on computation for *neuralizing* jmeint). Other clever heuristics (e.g., ones that evaluate potential performance loss due to memory access latency) can also be used to tackle this problem, which becomes exponentially more difficult for larger applications. Moreover, the process of exploring various combinations of NNs is trivially parallelizable, transforming this from a time-consuming process to a resource-intensive one.

4.2. Details of NN Modeling and Training

We employ supervised learning via the conventional back-propagation algorithm [Rumelhart et al. 1986] to train multilayer perceptrons (MLPs) [Hornik 1991]. The MLP model is a feed-forward network structured as an input layer, followed by one or more hidden layers, and finally an output layer. Functionally, the NN is evaluated using a series of weighted-sum and sigmoid operations without requiring any control flow. An example of a single-hidden-layer MLP (with labeled nodes and edge weights), along with the mathematical representation of its functionality, is shown in Figure 6(a). We choose the MLP as our NN model not only because it is a simple network to manage but also because of its flexible approximation capabilities, as described by the Universal Approximation Theorem [Hornik 1991].

Although a given kernel requires a fixed number of neurons in the input/output layers, we can search for optimal NN topologies by modifying the numbers of neurons in the hidden layers. Empirically, we have found that including three or more hidden layers generally results in excessive overhead without much gain in accuracy. Therefore, we limit our search to NNs with one to two hidden layers, and explore one to 32 neurons per layer (increasing by powers of two), resulting in an exploration space of 42 topologies. We begin with the smallest NN topology, train it, and compute the cross-validated mean squared error (MSE) value (i.e., the MSE corresponding to a “test” subset of the kernel’s training data). We then incrementally explore larger topologies, saving those that minimize MSE. The best topology for a given kernel is determined as the smallest topology that achieves the minimal MSE, prioritizing accuracy over topology size.

Quality of results is highly sensitive to training data, and this issue of sensitivity is given significant attention in the NN literature [Haykin 1998; Zhang 2000]. For NN training to be successful, the training data must be representative of the distribution from which evaluation data is taken; otherwise, no guarantees can be provided for the quality of results. Manual intervention for collecting better datasets would be needed for cases where the training data is no longer representative, and this results in unacceptable loss in the quality of results. In our experiments, we use a subset of the benchmark’s dataset for training purposes and use the remaining data for evaluation purposes. As such, the evaluation data is taken directly from the distribution of the training data, curbing the need for adjustment of the training data. However, in real-time systems with varying input spaces, such as an autonomous robot that interacts with a changing environment, the training data would need to be updated with new data the system encounters, which would then be used to retrain the NNs; this process could certainly be automated to run in the background as the system continues functioning in real time. We also note that this process would not require the entire software flow to be rerun, particularly because the kernels of interest are likely to not change. As such, only the dynamic profiler would need to be rerun to collect nonrepresented evaluation data, and the NNs would then be retrained with this new data. If the NN topologies are kept constant, this process would run on the order of minutes.

Similar to the distribution of the training data, the size of the dataset also plays a role in the neural approximation results. While reducing the size of the dataset leads to faster NN training, it could potentially degrade the accuracy of the trained NN (e.g., if the dataset is too small to accurately represent the distribution from which evaluation data is taken). Furthermore, research has found that better results can be achieved by simply increasing the size of training datasets, for instance, with natural language disambiguation [Banko and Brill 2001]. However, increasing the size of training datasets without increasing the range of the inputs could also lead to overfitting of the data (i.e., lack of generalizability for new evaluation data) [Haykin 1998]. There is therefore a delicate balance to maintain between size and distribution of training data.

4.3. Details of NN Integration

We consider several important implementation decisions for integrating an NN into existing code. First, the sigmoid operation (the hyperbolic tangent function in our case) could be computed in one of two ways: (1) using a lookup table (LUT) with precomputed values [Meher 2010], in which case only the address for the lookup would need to be computed instead of the entire sigmoid operation, or (2) computing the actual function (e.g., using a math library). In our scheme, we found that the `tanh()` function from the CUDA math library [Nvidia 2014b] performed very efficiently on the GPU and even matched the performance of the LUT regardless of where we stored the LUT in memory. The reason for this is because memory accesses are very costly on a GPU, not to mention the need to still compute addresses for each table lookup.

The second important implementation decision is regarding the storage of weight and bias values of the NN. In our implementation, we statically integrated these values into the code, thereby having them stored as “immediate” values held in program memory. Other options include storage into shared or cached constant memory. While program memory storage may not be the best storage option for all hardware platforms, we found it to be the most efficient implementation for the GPU.

4.4. Optimization Techniques

The creation of an NN-based approximation that is both accurate and performance-wise effective is a nontrivial task. It is especially difficult when a generalized methodology is used to create NNs for a variety of applications. For this purpose, we provide supplementary optimization techniques to help improve the accuracy of the NNs, as well as the performance and energy gains of the applications. As previously discussed, **targeting divergent kernels is not a limitation of our approach—it is a way to intelligently guide kernel identification by focusing on the ultimate weakness of SIMD architectures**. Once these main benefits have been reaped, further optimization can include subsumption of nondivergent code as well.

Our first optimization technique enlarges kernel scope, potentially allowing more divergent as well as nondivergent code to be encompassed by a single NN. This is done by modifying the kernel ranking criteria of the *Neuralizer* such that kernels are first prioritized by decreasing levels of scope before being prioritized by decreasing amounts of divergence. For example, a kernel with nondivergent control flow (e.g., deterministic “for” loop) encompassing a series of divergent control-flow regions (e.g., data-dependent “if-else” statements) would be given priority over those individual divergent kernels. Allowing larger kernels to be subsumed by a given NN, we reduce the number of NNs needed, thereby increasing maximum potential gains. Also, functional complexity is not directly related to kernel size, meaning larger scopes may even allow for more accurate NNs. An example of results from optimizing across different kernel scopes for the triangle intersection algorithm can be found in Section 5.3.

The second optimization technique we employ enables generation of better datasets for NN training. If the kernel is from an approximate algorithm (e.g., an iterative solver), we reverse engineer a dataset with exact solutions to properly train the NN, potentially achieving a lower error rate than the original application. For instance, with our inverse kinematics benchmark (`invkin`, described in Section 5.1), we use *forward kinematics* to generate a “correct” dataset to train the NN, providing a better representation of the input-output relation. With this optimization, we see an average of 24% improvement in our training results and are even able to achieve lower error rates than the original applications (e.g., 1.8% vs. 7.1% error for inverse kinematics).

Our final optimization technique is for any kernel that can be described as a classifier (e.g., one that returns a Boolean true/false value). For these kernels, the NN

Benchmark	Algorithm Description	Applic. Domain(s)	Control Flow Characterization	Approximability
invkin	Solves 3-joint inverse kinematics problem based on Cyclic Coordinate Descent (CCD)	Robotics, Graphics	Data-dependent multiple compute iterations	Heuristic-based algorithm: Iterative solver with no closed-form solution; not guaranteed to compute globally optimal result
nrpoly3	Uses Newton-Raphson method to find roots of a cubic polynomial	Nonlinear Algebra, Fluid Mechanics	Data-dependent multiple compute iterations	Heuristic-based algorithm: Iterative solver; not guaranteed to algorithmically converge
julfrac	Solves equations for points in a complex plane to generate Julia set fractals	Complex Dynamics	Data-dependent multiple compute iterations	Heuristic-based algorithm: Iterative solver; approximation of infinite fractals over discrete values
jmeint	Determines whether or not 2 given triangles intersect in 3D space	Graphics, Gaming, Computer Vision	Data-dependent single compute iteration (highly divergent)	Circumstantially approximable: E.g. output of <i>jmeint</i> could be used in approximate physics-based simulation
swaptions	Computes pricing for various stock options using Heath-Jarrow-Morton (HJM) framework	Finance	Data-dependent single compute iteration encompassed by non-data-dependent multiple compute iterations	Heuristic-based algorithm: Monte Carlo (MC) simulation used to generate inexact solution; region following MC output is approximated

Fig. 7. Summary of benchmark descriptions, characteristics, and justifications for approximability.

approximation can be augmented by filtering its outputs with a threshold-based classifier. For instance, the NN used for triangle intersection detection (i.e., *textttjmeint* benchmark described in Section 5.1) is supplemented with a stump classifier rooted at zero, improving the average error rate from 7% to 0.02%.

5. EVALUATION

We now discuss our evaluation methodology, including the benchmarks and evaluation metrics we used, as well as our experimental setup and results.

5.1. Benchmarks and Evaluation Metrics

For our evaluation, we have selected divergent applications from a variety of domains. The benchmarks are selected because they (1) suffer from branch divergence, (2) potentially tolerate imprecision, and (3) could be beneficial for general applications. We purposely do not select from GPGPU benchmark suites because they primarily include compute-intensive workloads specifically optimized for GPUs (e.g., control flow is minimized). We instead aim to enlarge the space of SIMD-targetable benchmarks. A summary of our benchmarks can be found in Figure 7. These benchmarks originate from various sources [Bienia et al. 2008; Sampson et al. 2011; Ortega and Rheinboldt 1970; Sanders and Kandrot 2010] and have been converted to CUDA for execution on the GPU. Leveraging maximum data parallelism and memory coalescence, the CUDA-based GPU versions of these benchmarks run on average $20\times$ faster (min. $6\times$, max. $46\times$) than their C++-based, multithreaded versions running on a 2GHz Intel Quad-Core-i7 machine.

Since the type of final output varies across applications, **evaluation metrics must be application specific**. The inverse kinematics benchmark (*invkin*) receives coordinate values for a target location and computes angle values for the three joints; to evaluate the error, we use forward kinematics to find the location of the end effector based on the computed angle values and calculate its distance from the target location. With the Newton-Raphson method for finding roots of an equation (*nrpoly3*), we compare the outputted root value to the correct root value as an average relative error. Since the benchmarks for finding Julia set fractals (*julfrac*) and detecting triangle intersection (*jmeint*) each return a Boolean value, error is evaluated as a miss rate. The final benchmark, *swaptions*, outputs arrays of values; as such, its evaluation metric is based on the root mean square (RMS) of array difference (much like how RMS of image difference is used for evaluating image processing). These evaluation metrics are reiterated for convenience in Figure 8. This table also provides justifications for

Benchmark	NN Topology	Input Data Set *	NN MSE	Eval. Metric	Eval. Error
invkin nrpoly3	2 x 32 x 32 x 3	10000 sets of 2D coordinate values	0.0106%	Distance	1.75%
	5 x 2 x 0 x 1	16384 sets of cubic polynomial parameters and initial guess value	0.0139%	Avg. Relative Error	0.91%
julfrac jmeint	2 x 4 x 2 x 1	100x100 matrix of values	5.1071%	Miss Rate	6.56%
	18 x 1 x 0 x 1	102400 pairs of 3D triangle vertex coordinate values	7.4798%	Miss Rate	0.02%
swaptions	33 x 16 x 0 x 33	16384 sets of 33-element array values	0.7942%	RMS	8.79%

* Note: All inputs are represented as floating point values.

Fig. 8. Characteristics, NN training results, and benchmark evaluation results for neural approximations.

the approximability of these benchmarks.

5.2. Experimental Setup

With respect to the generation, training, and testing of our NN models, we use the open-source, C-based Fast Artificial Neural Network (FANN) library [Nissen 2003] with support for floating-point values. To allow for steady convergence of the back-propagation algorithm, we use a learning rate of 0.01 along with a maximum number of epochs of 5,000. Also, while 75% of the benchmark's real input data is used for collecting the training datasets of kernels, the other 25% is used for *postneuralization* evaluation of benchmark error. For performance and energy evaluations, our CUDA benchmarks are compiled with version 5.5 of the Nvidia CUDA compiler [Nvidia 2014a]; we then execute the benchmarks on an Nvidia GeForce GTX 480 GPU [Nvidia 2013], which features 448 cores running at 607MHz with 16 warps per block and 32 threads per warp.

Our performance metric is execution time, measured using the standard CUDA-based event timing constructs. For gauging power, we employ an electricity load monitoring device [International 2014], which measures from a system's main power source. We first measure power when the system is idle. Then, we run the GPU benchmark long enough for it to reach steady state, remeasure the power, and take the difference between the two measurements. Though this could include dynamic power consumption of non-GPU components, these are negligible compared to the tens to hundreds of watts consumed by the GPU and allow us to obtain more realistic results than possible with a simulator. Energy is then computed as the product of this power measurement and the execution time from the performance result.

In our results, we compare the following schemes:

- **GPU**: Original GPU benchmarks
- **GPU_Ideal**: Nondivergent version of benchmarks (i.e., still include control flow, but have all threads process the same data values)
- **NN**: Benchmarks integrated with trained NNs
- **NN_Ideal**: Benchmarks integrated with zero-hidden-layer NNs

GPU_Ideal represents the best possible scenario for the GPU, where all threads not only fall within the same path of control flow (thereby eliminating divergence) but also fall within the path that would result in the highest performance possible for the GPU; as such, *GPU_Ideal* represents a performance-wise optimal version of an approximation-based technique known as “branch herding” [Sartori and Kumar 2013]. Also, the *GPU_Ideal* version of a benchmark is considered “ideal” because in reality, a user does not have control over data divergence. Similarly, the zero-hidden-layer NN implementation presented with *NN_Ideal* is considered “ideal” because an MLP with no hidden layers does not have the capability to approximate any functionality and cannot realistically be used. Using these schemes, we are able to see the impact of divergence (*GPU_Ideal* vs. *GPU*), the benefit of using neural approximation to remove divergence

Fig. 9. CDF plot of benchmark error; point (x, y) signifies that y percent of the outputs see x percent of error or less.

while potentially subsuming nondivergent code (NN vs. GPU_Ideal), and the upper bound on performance and energy gains for neural approximations (NN_Ideal vs. NN).

5.3. Experimental Results

5.3.1. NN Approximation. The characteristics of our NN-based approximations, along with application-specific evaluation metrics and error values, are summarized in Figure 8. Note that NN_MSE represents the cross-validated mean squared error of the NN, while the *Eval. Error* represents the application-level error assessment (i.e., based on the 25% of the benchmark inputs used for postneuralization evaluation). For each of our benchmarks, optimal gains were achieved with the use of a single NN to subsume all of the divergent control flow. Our results, therefore, correspond to these single-NN configurations. In terms of the average evaluation error rates, all five of the benchmarks are well within 10%. To examine quality degradation in more detail, related work in approximate computing [Esmaeilzadeh et al. 2012b] uses a plot of the cumulative distribution function (CDF) of error in an application’s output. We adopt this same approach for analysis and provide a CDF plot of benchmark evaluation error in Figure 9. This distribution reveals that 80% to 100% of the outputs for all five benchmarks have less than 10% error.

While we do not claim the error rates demonstrated for our benchmarks to be acceptable by all standards, we observe them to be on par with the range of quality loss seen in other approximation schemes [Sampson et al. 2011; Esmaeilzadeh et al. 2012b; Cho et al. 2012; Sidirolou-Douskos et al. 2011]. As with all approximate computing, statistically improbable errors could still render an application’s output meaningless. Different users may also have different notions of acceptable ranges of error for even the same application (e.g., *invkin* used for controlling robot-assisted surgery vs. robot-assisted movement of large blocks); the user would therefore need to deem these approximations acceptable. For these reasons, we see an opportunity to combine our approach with existing mechanisms for online error validation and user-based error-threshold specification [Grigorian and Reinman 2014].

5.3.2. Performance and Energy Gains. Figure 10 and Figure 11, which are normalized to the original *GPU* benchmarks, display our performance and energy results, respectively. We see that the three iterative, constraint-based solvers, namely, *invkin*,

Fig. 10. Performance gains of the various schemes.

Fig. 11. Energy savings of the various schemes.

nrpoly3, and julfrac, have the highest speedups because of the extent of the divergence being converted to nondivergent computation. If divergence is removed (i.e., *GPU_Ideal* results), $9\times$ to $21\times$ speedup can be achieved. As a result of further reducing the number of instructions executed, the trained neural approximations (i.e., *NN*) achieve speedups of up to $26\times$. Although the *jmeint* benchmark lacks data-dependent loops, it still contains a significant amount of divergence due to data-dependent branches. As such, performance improves by $3.1\times$ when the divergence is ideally removed, and neural approximation gains $4.8\times$ speedup, which is achievable because a very small NN topology (single hidden layer with one neuron) is able to satisfy this benchmark's approximation requirements. Compared to the other benchmarks, *swaptions* has less divergence and requires a large NN with many input/output neurons; it therefore exhibits relatively modest improvements using neural approximation. In Figure 11, we see similar trends in energy savings, with $8\times$ to $32\times$ savings for the

Fig. 12. Dynamic instruction counts to verify the source of the performance and energy gains.

iterative solvers, $5\times$ for the highly divergent triangle intersection detection, and $1.6\times$ for the many-input-output swaptions benchmark.

These performance and energy benefits are achieved with a combination of divergence elimination and reduction of dynamic instructions. For our benchmarks, we gauge the impact of divergence elimination using the “warp execution efficiency” profiling metric supported by CUDA; this metric measures the average percentage of active threads (i.e., threads performing useful work) in each executed warp. As expected, branch divergence leads to inefficient resource utilization and lowers warp execution efficiency. Our benchmarks originally have an average warp execution efficiency of 55%; after *neuralization*, however, the warp execution efficiency is transformed to 100% for all benchmarks. To further verify the source of our performance and energy gains, we include dynamic instruction counts in Figure 12. These results reveal a greater disparity between *GPU* and *GPU_Ideal* than between *GPU_Ideal* and *NN* for all the benchmarks, and in some cases (e.g., *nrpoly3*), *GPU_Ideal* may even execute fewer instructions than *NN*. In other words, while the amount of work in these applications certainly changes as instructions are subsumed by NN approximations, **divergence elimination is the key source of these gains** as it reduces the execution of unnecessary instructions and improves the efficiency of threads. This demonstrates the significance of changing the *nature* (and not just the *amount*) of the workload in these divergent applications. Furthermore, the gains of the NN implementations match closely with the upper bounds set by the idealized NNs for most benchmarks, thereby providing support for the effectiveness and low overhead of our technique.

Based on Figure 10 and Figure 12, we see that the instruction count of some benchmarks reduces dramatically going from *GPU* to *GPU_Ideal* while there is significantly less speedup achieved (e.g., $100\times$ instruction reduction vs. $21\times$ speedup for *nrpoly3*), which indicates a notable degradation in the number of executed instructions per cycle (IPC). The reason behind this involves the nature of the benchmarks and their memory access patterns. In an iterative benchmark that is having difficulty converging (e.g., with the baseline *GPU* case), a given thread is repeatedly computing for the same inputs using locally cached values, causing there to be minimal memory latency. However, if each thread is converging quickly (e.g., with *GPU_Ideal* case), it is quickly moving on to new inputs, which are accessed with relatively higher memory latency. In other words, memory stalls are incurred when we iterate on a new region of data;

Fig. 13. Performance gains and accuracy for jmeint benchmark as kernel scope varies.

our technique does not remove these stalls—instead, it reduces the instructions that use data already existing in cache or shared memory. As such, in the baseline *GPU* case, the stalls are amortized over a larger number of instructions, whereas in the *GPU_Ideal* case, the stalls are amortized over fewer instructions, resulting in the observed IPC degradation. This effect exists for all benchmarks we approximate but is exaggerated for the iterative benchmarks (e.g., *invkin*, *nrpoly3*, and *julfrac*) due to their nature of iterating continuously over locally cached values until an acceptable result is generated.

5.3.3. Varying Kernel Scope. We now present results corresponding to the technique for exploring various levels of kernel scope. Figure 13 shows the results of using the same NN topology (one hidden layer with one neuron) to approximate the four different levels of scope for the jmeint benchmark (for an overview of the kernel scopes, please refer back to the pseudocode in Figure 2). The purpose of these results is to demonstrate how increasing/decreasing scope can impact the functional dimensionality of a kernel in application-specific ways. In other words, increasing scope does not necessarily increase dimensionality, and decreasing scope does not necessarily have the opposite effect. For instance, we see from these results that the first level of branch nesting in fact has the highest accuracy when the same NN topology is used, thereby suggesting that its functionality is the easiest for the NN to learn. Furthermore, since it subsumes the largest portion of the benchmark, it has the highest potential for speedup. The second level of scope, conversely, diminishes in accuracy as its large number of inputs and high-dimensional functionality are too complex for the single neuron in the hidden layer to approximate accurately. The third and fourth levels of scope, however, reduce the functional dimensionality once again and achieve acceptable accuracy, though their speedups are limited as they subsume smaller portions of divergent control flow.

5.3.4. Varying NN Topology. To demonstrate the need to explore the space of NN topologies for a given kernel, we present the results of varying the NN topology when approximating the *invkin* benchmark (Figure 14). Much like with the *nrpoly3* results presented earlier as motivation (Figure 4), we see the performance gains reduce as the size of the NN increases. This figure also displays the benchmark evaluation accuracy corresponding to each NN topology used, and we see how (though it is not guaranteed to always be the case) a larger NN topology could likely result in higher approximation accuracy. For instance, in this case, an accuracy within 95% could be achieved with a

Fig. 14. Performance gains and accuracy for invkin benchmark as NN topology varies.

4x4 or larger topology. Balancing these tradeoffs, we find that a topology within the range of 4x4 and 8x8 (e.g., the 4x8 used for our final performance and energy results) would be ideal for this benchmark. As the performance of a given NN topology can vary drastically from one kernel to the next, we find that exploration of NN topologies is key in optimizing the gains achieved by neural approximations.

6. RELATED WORK

6.1. SIMD Divergence

Prior art has proposed various techniques to address SIMD divergence. SIMD instruction set extensions [Lomont 2011] rely on the core they are tied to for implicit handling of control flow. Similar to the active masks on GPUs, VPU architectures [Smith et al. 2000] use “vector bit masks” to control the outputs of processing elements; this technique ensures correctness and potentially reduces execution time but still results in low *useful* utilization. Some GPUs [Lindholm et al. 2008] use “priority scheduling” of warps to hide latency of divergence, but this thread scheduling procedure incurs overhead while only resolving the latency of stalled threads instead of the latency of the divergent threads themselves. Other techniques [Fung et al. 2007; Meng et al. 2010; Fung and Aamodt 2011; Narasiman et al. 2011] dynamically modify (regroup, divide, or compact) the thread warps so as to reduce latency and memory divergence, yet this involves costly hardware modifications with limited impact on overall system utilization. Even static techniques [Diamos et al. 2011; Wu et al. 2012] lose efficiency with complex data-dependent control flow. Furthermore, accelerator-rich designs [Cong et al. 2012a, 2012b] either subsume the control flow into a monolithic accelerator, decompose the control flow into separate accelerators, or avoid it entirely by offloading it to the core; while they ensure program correctness, each of these alternatives can be costly in terms of area, power, and resource utilization.

Unlike many of these existing approaches, our methodology is platform agnostic and avoids incurring hardware overhead to handle divergence. Moreover, we explore the efficiency to be gained if the *nature* of the computation is regularized. Other software-based approximation techniques that similarly target data-parallel hardware, such as GPUs, include SAGE [Samadi et al. 2013] and Paraprox [Samadi et al. 2014]. Though NNs are not employed, these techniques also identify approximable code regions using compiler-based support and substitute those regions with approximate

implementations. Likewise, the approximation-based technique known as “branch herding” [Sartori and Kumar 2013] similarly aims to transform computation into a nondivergent form. This technique reduces branch divergence by “herding” threads of GPU warps down the same path for various control-flow regions and uses static analysis and profiling to minimize output degradation. In comparison, our NN implementations consistently outperform the performance-wise optimal implementations of branch herding (i.e., *GPU_Ideal*) while maintaining reasonable accuracy. The reason for this is that we instead approximate control-flow regions by exploiting the intelligent learning capabilities of NNs, allowing us to emulate the functionality of the different branch paths using the same nondivergent computation. As a result, our trained NNs achieve higher accuracy while subsuming more computation.

6.2. Neural Approximation

Neural networks [Haykin 1998] have been widely studied for pattern recognition, machine learning, and classification. In an effort to broaden the applicability of NNs and promote general-purpose neural acceleration, Chen et al. [2012] have developed software NN implementations of high-performance applications from the PARSEC [Bienia et al. 2008] benchmark suite. However, their methodology calls for complete manual reimplementations of entire benchmarks.

While neural approximation of code regions has been used in prior art, control-flow regions either have not been explored thoroughly or have remained entirely unexplored. For instance, the related work by Esmaeilzadeh et al. [2012b] has no control flow in the majority of its benchmarks, and only a single benchmark (*jmeint*) with complex control flow. Therefore, the ability to approximate control flow, particularly divergent control flow, is a new finding. Similar to our approach, Esmaeilzadeh et al. train NNs to mimic the functionality of code kernels from approximable programs. However, they require ad hoc kernel identification, whereas our divergence-guided approach strategically automates kernel identification while addressing major microarchitectural inefficiencies of SIMD designs. We further introduce supplementary techniques to optimize neural approximations, which allows us to achieve lower error rates (e.g., in *jmeint*). Lastly, while Esmaeilzadeh et al. use hardware-based neural processing units to which they offload the NN computations at runtime, we have developed an efficient software-based implementation to accelerate applications in a platform-agnostic fashion.

7. CONCLUSION

In this article, we have examined the problem of SIMD branch divergence and have presented our approach based on neural networks (NNs), where we approximate control-flow regions in order to trade off precision for gains in performance and energy. Our approach includes a complete methodology with an automated software flow and supplemental optimization techniques. While we evaluate our approach on a GPU, we maintain that these techniques can be generally applied for approximation-based acceleration of divergent applications on SIMD architectures. Our results show average performance gains of 13.6 $\times$ and energy savings of 14.8 $\times$ with about 4% loss in benchmark accuracy. Notably, we see performance gains of over 10 $\times$ and energy savings of over 8 $\times$ for algorithms that involve iterative, highly divergent computation. This research also highlights the importance of exploring different NN topologies and kernel scopes in the effort to find optimal neural approximations.

REFERENCES

- Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 198–209.

- Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. 26–33.
- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 72–81.
- Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 33–52.
- Swarat Chaudhuri, Sumit Gulwani, Roberto Lubliner, and Sara Navidpour. 2011. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 102–112.
- Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. 2008. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing* 68, 10 (2008), 1370–1380.
- T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam. 2012. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *IEEE International Symposium on Workload Characterization*. 36–45.
- H. Cho, L. Leem, and S. Mitra. 2012. ERSAs: Error resilient system architecture for probabilistic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 4 (2012), 546–558.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012a. Architecture support for accelerator-rich CMPs. In *Proceedings of the 49th Annual Design Automation Conference*. 843–849.
- Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012b. CHARM: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*. 379–384.
- Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 477–488.
- Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012a. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–312.
- Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012b. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 449–460.
- W. W. L. Fung and T. M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*. 25–36.
- Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 407–420.
- Beayna Grigorian and Glenn Reinman. 2014. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *NASA/ESA Conference on Adaptive Hardware and Systems*. 248–255.
- Michael Gschwind. 2006. Chip multiprocessing and the cell broadband engine. In *Proceedings of the 3rd Conference on Computing Frontiers*. 1–8.
- Simon Haykin. 1998. *Neural Networks: A Comprehensive Foundation* (2nd ed.). Prentice Hall PTR.
- Kurt Hornik. 1991. Approximation capabilities of multilayer feedforward networks. *Neural Networks* 4, 2 (March 1991), 251–257.
- P3 International. 2014. Kill A Watt. <http://www.p3international.com/products/p4400.html>. (2014).
- Ujval J. Kapasi. 2004. *Conditional Techniques for Stream Processing Kernels*. Ph.D. Dissertation. Stanford University.
- Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Bruce Khailany. 2000. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*. 159–170.
- Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*. 129–140.

- Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55.
- Chris Lomont. 2011. Introduction to Intel advanced vector extensions. In *Proceedings of the 2nd Annual ASCI Conference*. 132–137.
- P. K. Meher. 2010. An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks. In *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference*. 91–95.
- Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 235–246.
- Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 308–317.
- Steffen Nissen. 2003. *Implementation of a Fast Artificial Neural Network Library (FANN)*. Report. Department of Computer Science University of Copenhagen (DIKU).
- Nvidia. 2013. GeForce GTX 480. Retrieved from <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480>.
- Nvidia. 2014a. CUDA 5.5 Production Release. Retrieved from <http://developer.nvidia.com/cuda-downloads>.
- Nvidia. 2014b. CUDA Math Library. Retrieved from <http://developer.nvidia.com/cuda-math-library>.
- J. M. Ortega and W. C. Rheinboldt. 1970. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press.
- Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters* 10 (2000), 215–226.
- David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.
- Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 35–50.
- Mehrzad Samadi, Janghaeng Lee, Davoud Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–24.
- Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 164–174.
- Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.
- J. Sartori and R. Kumar. 2013. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia* 15, 2 (Feb. 2013), 279–290.
- Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics* 27, 3 (Aug. 2008), 18:1–18:15.
- Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 124–134.
- J. E. Smith, G. Faanes, and R. Sugumar. 2000. Vector instruction set support for conditional operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 260–269.
- Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. 2013. SIMD divergence optimization through intra-warp compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 368–379.
- Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 163–174.
- Ingo Wald. 2011. Active thread compaction for GPU path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. 51–58.

Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. 2012. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. *International Journal of High Performance Computing Applications* 26, 2 (2012), 170–185.

Guoqiang P. Zhang. 2000. Neural networks for classification: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 30, 4 (2000), 451–462.

Received September 2014; revised December 2014; accepted January 2015