GPU Architecture and Applications in Artificial Intelligence

Connor Brooks
Department of Computer Science
Western Kentucky University
Bowling Green, KY 42101
connor.brooks499@topper.wku.edu

***Abstract*—The objective of this paper is to examine the design of GPUs with a focus on current and potential applications in artificial intelligence. The evolution of GPU design is traced and connected to its usage at various times in its development. Advantages of GPUs over typical CPUs are explained, then these advantages are shown to be important for various artificial intelligence applications that comprise growing fields of study. Current uses for GPUs in these fields are documented, and potential future implementations involving GPU usage are identified.**

***Index Terms*—GPU, GPU Architecture Applications, Artificial Intelligence, Parallel Programming**

I. Introduction

Graphics Processing Units (GPUs) became widespread in the 1990s as a common part of computers of the time. Designed for the rendering of graphics, GPUs are highly parallelized and are capable of performing many independent functions simultaneously. Because of this ability, most modern-day GPUs contain a much higher degree of computational ability than modern CPUs [1]. This is due to the focus of GPUs on throughput, as compared to the design of CPUs being focused on latency. The result is modern CPUs containing just a few cores, while GPUs contain hundreds. Because of the evolution of GPU design away from hardware implementations that are dedicated to graphics processing and towards more programmable, general-purpose units, they can be applied to various problems which may benefit from their massive computing power. Taking advantage of this higher degree of computational ability requires finding ways to use massively parallel processing on important tasks for the full capabilities of the GPU to be used.

The potential applications for general-purpose GPUs reach far beyond their birthplace of graphics rendering, and the continued abstraction of their design away from dedicated graphics functionality towards general-purpose computing is contributing to growing usage in fields such as diverse as financial modeling, scientific research, and oil and gas exploration. This paper examines the growing usage of GPUs for general-purpose computing and the history and potential of GPUs for implementation in the field of artificial intelligence. Many problems in this growing field require large computing tasks which can be performed on a parallel architecture, so GPUs are well-suited for these tasks. Applying GPUs to artificial intelligence problems then requires both identification of tasks to which these units capabilities will be applicable and structuring solutions in such a way to take advantage of these capabilities.

II. History of GPUs and GPU Design

A. Background

The history of GPUs does not begin with the first true GPU, but instead with research into parallel processing and the use of these kinds of programs. Techniques such as stream processing, which are critical to GPUs, date back as far as IBM research in the 1950s [2]. This research investigated specific methods of programming that would allow for some parallel processes. However, for many years, the development of CPUs was focused on minimizing latency through the evolution of technology and substantial progress was not made on further development of parallel processes. Special techniques or units were developed to handle computer graphics during this time, but GPUs were not created until the 1990s. During this decade, the advent of 3D graphics created a problem in which the same processes must be independently performed on many units at a rapid frequency. This problem led to the creation of the GPU, which used highly parallel processing to be optimized for tasks involving graphics [3].

The problem of processing 3D graphics involves performing certain functions, such as shading, on a vast amount of individual units (like individual pixels). Though CPUs could in theory successfully complete this task, the individual processes would create a massive bottleneck waiting for the processes on all prior units to complete before any new process could be executed. Because of this bottleneck, rendering detailed graphics was not feasible at a high enough frequency for real-time animation using CPUs. This situation led to the development of a unit that would be designed with a focus on enabling parallel processing so that these functions could be executed simultaneously when possible; thus, the GPU was born.

B. GPU Architecture and Specialization

1) Basic GPU Architecture: As GPUs were developed with a specific purpose in mind of processing graphics (in contrast to the general-purpose design goal of CPUs), their architecture and design were optimized for this designated purpose. Rather than focusing on developing a single or few cores that are

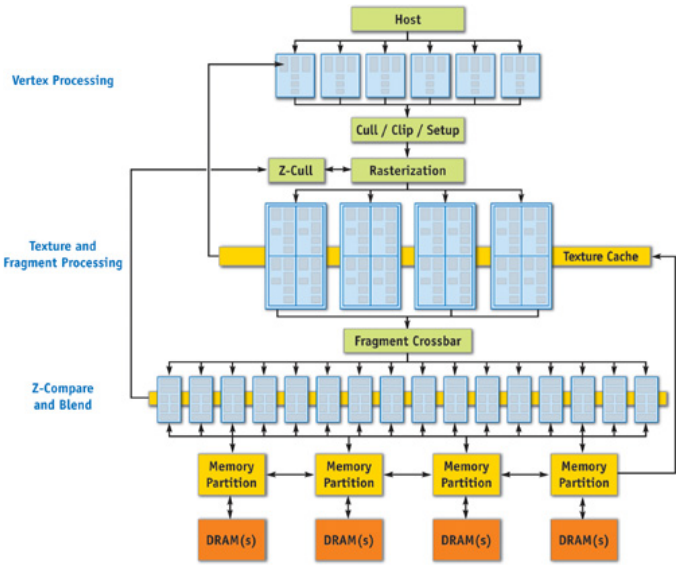

Fig. 1. GPU Architecture from NVIDIA GeForce 6 Series

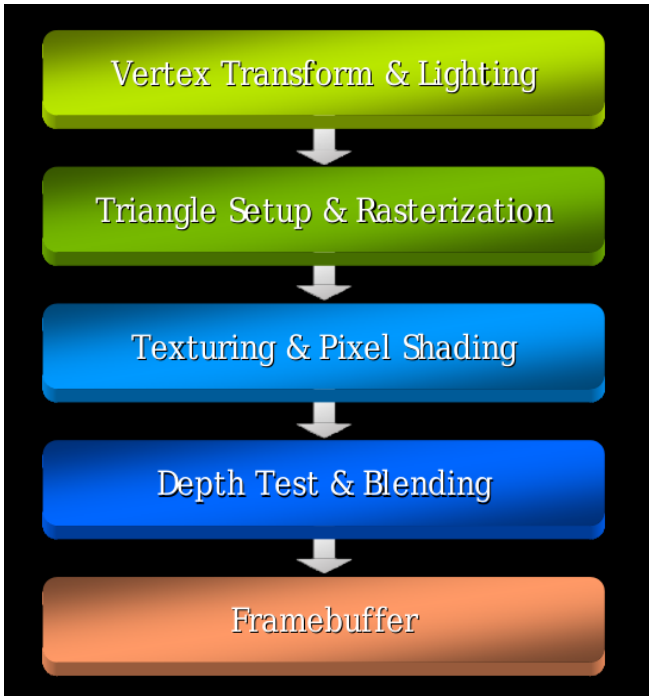

Fig. 2. Overview of Graphics Pipeline [4]

optimized for minimizing latency as was being done in CPU design, GPUs were designed with hundreds of cores to enable the handling of concurrent processes in order to maximize throughput. In addition to allowing parallel processes to take place through the addition of hundreds of separate cores, further design considerations optimized GPU for their use in graphics in the creation of task-optimized cores and the overall architecture of the unit. Rather than containing a single type of core, multiple types of cores were included to perform different segments of the larger task of graphics processing. The overall design of the GPU used a hardware pipeline that used different kinds of cores for the respective tasks for which they are specialized. Again, this pipeline was designed to be parallel, with numerous cores available for stages that allow parallel processing (although some bottleneck points still exist at certain tasks involving tasks that cannot be performed in parallel). This graphics pipeline was used in the design of GPUs to perform several specialized sequential functions on each unit.

Through this further specialization of hardware to implement the graphics pipeline, GPUs were optimized for graphics by not only allowing parallel execution of independent functions through the use of hundreds of cores, but also were optimized to take advantage of similar functions being executed on independent units through the use of a graphics pipeline which had specialized cores for specific tasks [1]. This design resulted in a highly-specialized device with powerful computational ability for its intended purpose. The creation of GPUs allowed the processing of complex 3D graphics at rapid frequencies to allow for real-time 3D animation.

2) Load Balancing Capabilities: Despite the advantages of a parallel approach to graphics rendering, the implementation of efficient parallel processing is made more difficult by the non-static requirements of individual pixels and regions during rendering. The requirements of particular images and scenes can cause extreme fluctuations in the amount of computing resources needed at a particular stage for a particular data. Due to this fluctuation, a considerable loss of efficiency would result from maintaining a static pipeline that does address these dynamic alterations of computing power needed. Consequently, dynamic load balancing is included in modern GPUs that uses heuristics to handle fluctuations in pipeline load during operation [5]. This provides another powerful tool for parallel programming processes, ensuring that the GPUs resources are used as efficiently as possible even during dynamic tasks.

3) Hardware Support for Multithreading: A significant issue for both CPUs and GPUs is that of thread stalls due to outstanding instructions. In both cases, the goal is to hide these stalls and maintain efficient usage of computing resources rather than waiting on high-latency operations. CPUs handle this issue using logic to determine complex ways to execute instructions in an order such that stalls can be hidden, in combination with large data caches that allow rapid access to minimize stalls in the first place. GPUs, on the other hand, use hardware multithreading by maintaining more execution tasks than they are able to handle at one time, utilizing scheduling logic to decide which tasks to execute at any given time in such a way to hide stalls. This gives GPUs a much greater capability to hide thread stalls than CPUs, as demonstrated in the last column of Fig. 3.

4) Move Toward General Use: GPUs have continued to grow more powerful as technology (and the demands of consumers for state-of-the-art graphics) has advanced - a GPU in

Type	Processor	Cores/Chip	ALUs/Core[3]	SIMD width	Max T[4]
GPUs	AMD Radeon HD 4870	10	80	64	25
	NVIDIA GeForce GTX 280	30	8	32	128
CPUs	Intel Core 2 Quad[1]	4	8	4	1
	STI Cell BE[2]	8	4	4	1
	Sun UltraSPARC T2	8	1	1	4

[1] SSE processing only, does not account for traditional FPU
[2] Stream processing (SPE) cores only, does not account for PPU cores.
[3] 32-bit floating point operations
[4] Max T is defined as the maximum ratio of hardware-managed thread execution contexts to simultaneously executable threads (not an absolute count of hardware-managed execution contexts). This ratio is a measure of a processor's ability to automatically hide thread stalls using hardware multithreading.

Fig. 3. Comparison of CPU and GPU Throughput Architectures [5]

1995 contained about one million transistors, while GPUs had surpassed one billion transistors by 2008 [4]. Perhaps more significant than this development of technology, however, is the development of GPUs away from dedicated graphics functionality and toward more general computing abilities. In 2001, Nvidia and ATI began making their GPUs programmable, allowing for developers to control some processing in order to improve real-time rendering [2]. As these programmable processes, known as shaders, became more and more critical to the functionality of GPUs, the centrality of shaders to the GPU architecture grew. This allowed for the dissolution of the graphics pipeline into a more flexible core, with the distinct processes separated at a hardware level in the graphics pipeline now treated more similarly to parallel threads executed on a flexible core [4]. This produced a hardware architecture still optimized for parallelism, but with less of a focus on hardware specialization for graphics processing. The result is a device capable of massively parallel processing that can be utilized for a variety of applications using programming techniques that take advantage of its capabilities.

This broadening of GPU usage came with a large amount of support from GPU providers. In 2007, NVIDIA released the initial version of a new hardware and software architecture, CUDA (Compute Unified Device Architecture), to allow developers to program GPUs using high-level programming languages [6]. This was a large step forward, as the use of GPUs before this kind of interface required expressing problems in terms of graphic problems. With CUDA, programmers are able to use languages such as C++ to write programs that use thread blocks, which are implemented on the GPU as concurrently executing threads which can share memory and resources. The CUDA architecture regulates and IDs each thread and maintains efficient processing of parallel tasks without requiring the programmer to explicitly handle hardware execution, thread stalling, load balancing, or other considerations.

III. Constraints and Considerations of GPU Programming

While the potential for GPU usage in algorithmic design provides the hardware necessary for massive parallelism that can benefit the runtime of many processes, there are some special considerations that must be taken when creating a GPU implementation of an algorithm. NVIDIA's CUDA, which is one of the most widespread programming model for GPUs and is thus indicative of the type of programming required for GPU implementations, uses Single-Instruction Multi-Thread (SIMT) parallelism. This allows the same instruction to be performed on several distinct threads, while each thread uses its own data. This SIMT model utilizes a CPU as a host that sends out the instructions, and the GPU is used as a device that can process a number of threads dependent upon its number of cores. Data can be shared between the host and the device, and the device's capability for data storage again are dependent on the specific model of GPU being used. The specific GPU being used also affects the behavior of different memory levels and the organization of memory on the device in general. Thus, GPU programs need to take into account specific details of the GPU device being used in order to reach full optimization [7].

Additionally, it is worth noting that due to the focus on development of GPUs for high throughput over latency, the cores of a CPU are typically much more powerful and faster than the cores of the GPU. This means that implementations on GPUs must utilize parallelism to obtain speed-ups over CPUs. Otherwise, if a program is simply executed on a GPU rather than a CPU but is not designed to take advantage of the GPUs capabilities, the program will run much slower on the GPU than on the CPU. Additionally, memory access is not as optimized on GPUs as it is on CPUs, and the organization of memory is much more complex due to the number of cores and the requirement for localized memory levels for blocks of cores. Taking these factors into account and designing algorithms meant for implementation on GPUs thus provides an overhead to parallelism. The speed-up obtained by using multiple cores must be more significant than the overhead of parallelism for a GPU architecture to be worth using.

Due to the overhead of parallelism, not all problems are naturally designed for GPU implementation. Substantial research has been done into implementing GPUs for various problems, with results varying from problem to problem. Typically, problems that can be naturally conceptualized as composed of multiple independent tasks tend to perform better on GPUs, but clever ways of applying parallelism to problems without obviously independent tasks can also be created.

IV. Artificial Intelligence Applications

Artificial intelligence is a growing field which includes many tasks that require vast computational capabilities. The field of artificial intelligence contains many research areas which can benefit from the highly parallel architecture of GPUs. As GPU design has steadily moved towards creating GPUs that are more and more capable of general-purpose computing, many researchers within the field of artificial intelligence have begun to use GPUs for various applications that can utilize parallel processing. This trend has been encouraged by manufacturers such as NVIDIA, who have begun creating

dedicated models for use in and marketed to specific artificial intelligence application usage [8].

A. Neural Networks

A large subfield of artificial intelligence involves the study of neural networks and their application to solving various tasks. These networks contain hundreds or thousands of individual nodes connected to one another in layers. Input is processed forward through the network and then, based on the results, the individual nodes are slightly altered for each training round for a given network. Thus each round of processing involves thousands of computations, and thousands of rounds of training are typically necessary to achieve a well-performing neural network.

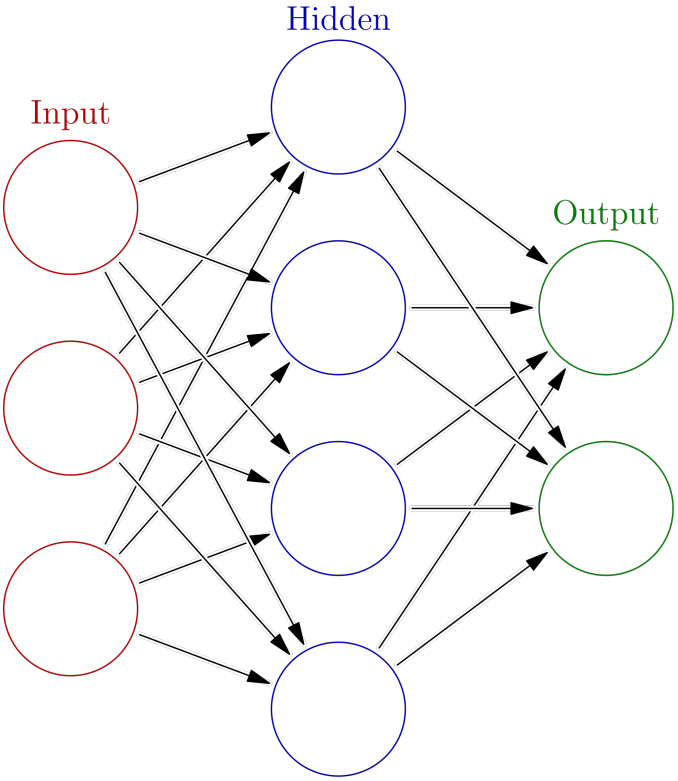

Fig. 4. Illustration of a Simple Neural Network

After neural networks began to grow in popularity in research due to their success in complex pattern matching and other tasks, they did not immediately begin to proliferate into commercial uses, remaining largely a focus of lab research. This was partially due to the early implementation of neural networks as software that ran on typical CPUs with no special hardware optimization for neural networks, resulting in costly computations that were often too impractical for use in industry [9]. This issue was further compounded due to increasing demonstration that neural networks could become much more powerful as they grew larger and contained more layers, resulting in much more complex networks. This problem became even more severe due to the way in which neural networks are processed resulting in the the number of computations involved in software implementations of neural networks growing quadratically compared to the size of the network. Clearly, for neural networks large and complex enough to handle needed tasks quickly enough to be useful in real world implementation, software implementation of neural networks was insufficient.

Today, one of the most common uses for GPUs in the field of artificial intelligence is in the creation and processing of neural networks for machine learning. Some researchers have suggested using the massively parallel architecture of GPUs to handle deep learning, which requires extensive amounts of processing, much of which can be done in parallel. Applications such as neural networks require individual functions performed on every node, which can number in the thousands. GPUs are well-suited to this task, and are consequently commonplace among neural network implementations. Their usage allows for the processing of the nodes within the neural network to be handled concurrently rather than sequentially, speeding up the time it takes to complete a round of processing. Several different types of neural networks have been implemented on both GPUs and CPUs and their processing time compared by various groups of researchers to measure the impact of GPU architecture on the capabilities of neural networks. Results, as shown in the following section, have demonstrated significant speed-ups.

1) Restricted Boltzmann Machine Implementations: Due to recognition of the parallel requirements of neural networks compounded with the growing generalization of GPU hardware, researchers began comparing the performance of neural networks implemented on CPUs to those implemented on GPUs in the early 2000s. Ly et al. [9] compared a CPU implementation of a Restricted Boltzmann Machine, a popular type of neural network, to an implementation using CUDA on a GPU, isolating kernels to handle specific components of the computation required for training the Restricted Boltzmann Machine and comparing performances on each individual kernel. They found that a speed-up of 66-fold could be obtained on the entire execution, although the individual kernels on the CUDA implementation could reach speed-ups surpassing thousands of times faster than the CPU implementation. The relatively smaller speed-up of the entire execution compared to the individual kernels was attributed to memory copies and synchronization of threads in the CUDA implementation. The results of the individual kernel speed-ups are demonstrated in Fig. 5.

2) Convolutional Neural Network Implementations: Another type of neural network which enables processing of complex tasks such as image processing is the convolutional neural network. The multiple layers in these networks essentially allow for features to be extracted by one layer, then fed forward to a network that uses these features in its calculation. Strigl et al. [10] compared GPU implementations of convolutional neural networks to CPU implementations, again using CUDA for the GPU implementations and found that the GPU implementations again were substantially faster, with speed-

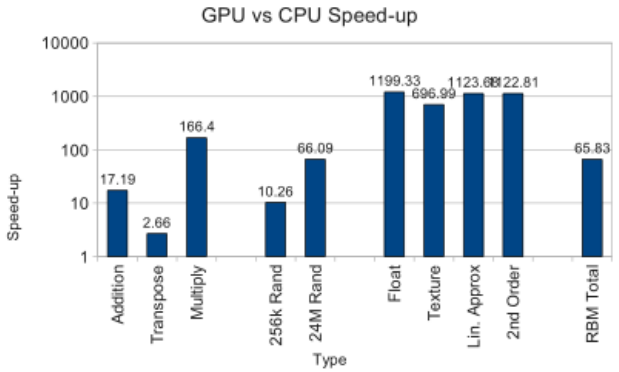

Fig. 5. Comparison of Restricted Boltzmann Machine Performance on CUDA and CPU [9]

ups ranging from 2- to 25-fold in their implementation. These speed-ups increased with the complexity of the networks. The results of their tests are shown in Fig. 6.

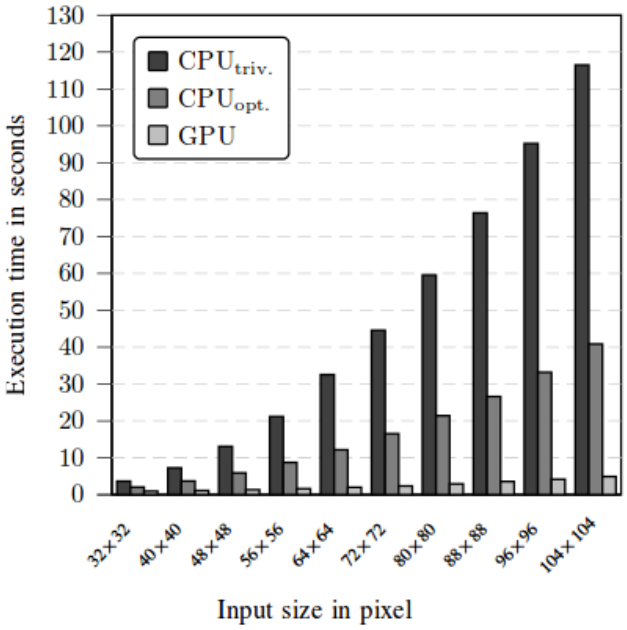

Fig. 6. Comparison of Convolutional Neural Network Performance on CUDA and CPU [10]

3) Deep Neural Networks: The growth in popularity of neural networks and their increased capabilities largely due to GPU implementation has allowed for the creation and implementation of deep neural networks. Deep neural networks are neural networks with multiple hidden layers of nodes, allowing the creation of much more complex networks. While these networks existed conceptually before the 2000s, they were exceedingly impractical to implement due to the computation time required to train them. After the usage of GPUs began to make neural network implementation more practical, deep neural networks were able to begin to be implemented on a large scale. Raina et al. compared GPU and CPU performance in processing deep belief networks, a kind of deep neural network [11]. In one example with a relatively small network (compared to the desired size of the network for their intended task), they found that the network could update using a million parameters in about 29 minutes using a GPU implementation, compared to taking more than a day to do the same amount of processing on a CPU implementation. Considering that their actual tasks were to use tens of millions of parameters rather than just a million, the use of a GPU did more than just speed-up their processing tasks, but actually made tasks feasible that would have simply been impractical to ever calculate on a CPU.

One issue with GPU usage for training large networks is that it is more difficult to successfully implement clusters of GPUs than it is to implement clusters of CPUs. This becomes an issue when using deep learning neural networks at extremely large scales that may be too large for a single GPU to handle. Thus, much research has also been done into using large-scale CPU clusters as an alternative to using GPUs for training neural networks, as this allows the growth to continue by adding more and more machines to the cluster. Coates et al. [12] investigate the creation of network infrastructure and algorithmic design to allow for the successful implementation of GPU clusters, providing a powerful method that can scale to handle the training of enormous neural networks. In their research, they analyze the training of deep neural networks with over one billion trainable parameters, such as those at the edge of modern deep learning research. Training neural networks at these scales using CPUs requires thousands of separate machines. Through utilizing a small GPU cluster, the researchers are able to demonstrate training a network at this scale in a similar amount of time while using only three separate machines. Their research provides a potentially scalable solution to handle research into larger and larger neural networks.

Deep neural networks have demonstrated advances in handling extremely complex problems, from landmark recognition to natural language processing, and are pushing forward the status of artificial intelligence research. This trend is one that GPU manufacturers have again taken note of: NVIDIA has even released an SDK specifically for the creation of deep neural networks [8]. As neural networks that enable deep learning can contain several layers of nodes and can grow substantially complex during training, the parallel capabilities of GPUs is critical to processing large, deep learning neural networks. As this technique of deep learning continues to grow and be applied to problems that required sophisticated networks with many layers, the role of GPUs and their focus on parallelism will become more and more critical to any task that can be completed by such networks.

4) Impact of GPUs on Neural Networks: As demonstrated above, GPUs have proven a powerful tool for dramatically speeding up implementations of neural networks. These speed-ups do not merely allow for quicker applications - in the case

of the exponentially growing complexity of neural networks, the speed-ups provided by GPUs can enable entirely new applications that would be impossible with standard software implementations. Ongoing research into neural networks has shown that these structures are able to be used to accomplish more and more complex tasks as the network grows larger. While comparatively small networks can handle simple digit classification, larger networks have continued to break records on tasks such as general image classification and competition against humans involving complex games. Consequently, the speed-ups of neural networks on GPU implementations is a significant push forward for all artificial intelligence research that uses these networks to learn how to perform complex tasks, allowing for more and more complex tasks to be handled by larger networks. The field of neural network research suffered a winter due to supposed limits on their capabilities, but the advent of GPUs helped end this winter and bring them back to the forefront of artificial intelligence research. As deep learning continues to become more widespread, the implementation of neural networks on GPUs will remain a significant factor of their success.

B. Optimization Problems

One classic subset of artificial intelligence problems is optimization problems. Optimization techniques search through a solution space to find the optimum solution according to some predefined criteria. These techniques can be applied to many, many different problems in various fields, and research into their solutions using artificial intelligence techniques dates back several decades. Widely differing techniques have been developed for these problems. One family of techniques is known as swarm-based optimization. These techniques involve the use of a large number of possible solutions which comprise a population. Different strategies within this family leverage this ”swarm” in various ways to find the optimum solution. As swarm-based optimization solutions usually require processing on a large amount of individual solutions, some of these techniques may lend themselves well to parallelism.

1) Genetic Programming: Genetic programming involves mimicking natural processes such as natural selection through the use of techniques such as an artificial form of natural selection that involves potential solutions which perform well being combined together to generate new generations of solutions. In order for this process to take place, a fitness function must routinely be performed on each solution, called an individual, to determine its status and evaluate how well it is performing a given task. This evaluation is the most time consuming component of such algorithms and makes up the bulk of computation required for many genetic programs as it must be performed frequently on a large set of individuals.

As the fitness functions for individual entities are often entirely independent of one another, a parallel approach to the problem can be applied. Such a situation is well-suited for the use of a GPU, which can process fitness functions for individual solutions in parallel. Harding et al. [13] found speed-ups reaching hundreds of times the performance on CPUs in their GPU implementation of general purpose genetic programming algorithms. The speed-ups increased with scale, allowing GPU implementations to handle complex problems with a large number of individuals that were beyond the capability of CPU implementations.

Similar to the case of neural networks, the remarkable speed-ups demonstrated by GPU implementations of genetic programming tasks go beyond merely optimizing the time to solve problems. Instead, the ability of the GPU implementations to outperform CPU implementations more dramatically the larger the scale of the problem creates situations in which GPU implementations can handle problems that are impossible or, at the very least, impractical for CPU implementations. This has an impact on the capabilities of the field of genetic programming as a whole, as it allows more complex tasks to attempt to be solved using genetic programming tasks.

2) Particle Swarm Optimization: Particle Swarm Optimization is an optimization technique that relies on a large set of ”particles” in the solution space to comprise the swarm. At each iteration, every individual particle adjusts its position, taking into account both the best position that this individual particle has found as well as the best position that has been found by any particle in the entire swarm. Like genetic algorithms, particle swarm algorithms require the processing of a fitness function on each particle to determine its relative success. As complex versions of these problems can involve many dimensions and a large number of particles, this can become very expensive computationally. The complexity is also related to the complexity of the fitness function. As the fitness function is executed once per particle per iteration, changes in complexity of this function cause significant changes in complexity of the entire technique.

Zhou et al. [14] benchmarked CUDA implementations of particle swarm optimization using various fitness functions, swarm sizes, and dimensional spaces to compare to analogous performance using CPUs. They found speed-ups that increased with growing complexity of fitness functions, larger swarm sizes, and larger dimensional spaces. Their results using three different fitness functions and growing swarm sizes are shown in Fig. 7, and their results using three different fitness functions and growing dimensional spaces are shown in Fig. 8. Again, the trend in this problem demonstrates a growing advantage of GPUs over CPUs as the scale of the problem grows. Particle swarm optimization has been applied to other artificial intelligence problems, including neural network training and machine learning, so advances on this technique through GPU implementations have impacts on these fields as well.

C. Search Problems

Another large subset of artificial intelligence problems are those that can be solved by some form of search algorithm. These problems can involve various specifications and goals, but all require finding a solution within a search space or collection of potential solutions. Not all of these problems lend themselves well to parallelism or GPU implementation, but some do.

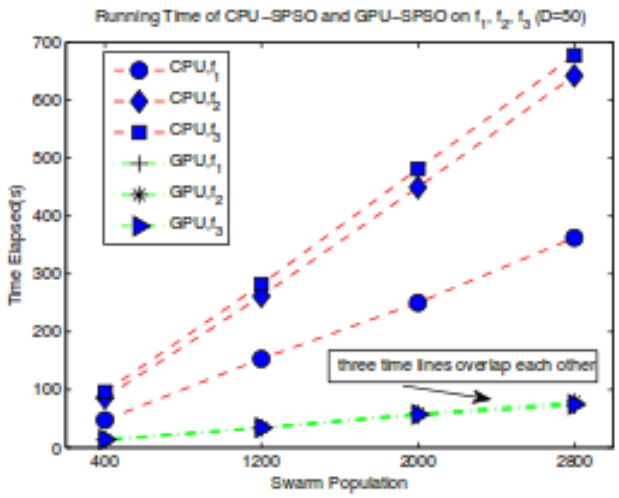

Fig. 7. Speed-up for Particle Swarm Optimization with growing number of particles [14]

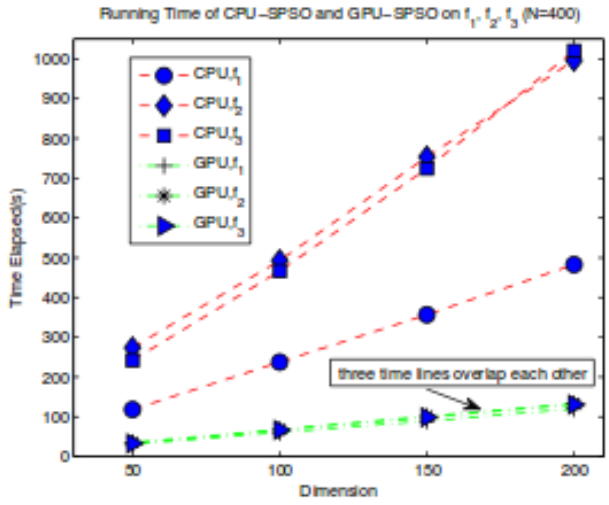

Fig. 8. Speed-up for Particle Swarm Optimization with growing number of dimensions [14]

1) K Nearest Neighbor: GPUs have also been examined for their use in solving the k nearest neighbor search problem, whose solutions have applications in a variety of fields. In this problem, given a set of reference points and a set of query points, the k nearest reference points for each query point are found. Exhaustive search for this problem requires computing the distance from each query point to every reference point, and individually sorting the reference point distances for each query point. This brute force solution is inefficient and can be quite costly depending on the cardinality of the reference set and the cardinality of the query set. The computation time grows polynomial with these sets' cardinality, and calculations can become inefficient with sufficiently large sets. There are various algorithms that provide more efficient solutions to this problem such as a kd-tree based method, but all solutions involve computing the distance from a query point to many reference points to check if these nodes are within k distance from the query.

This problem lends itself naturally to parallelization, since the distance calculations can be done in parallel naturally, and the distances of the individual query points can be sorted in parallel as well. This decreases the processing time substantially and allows for the computation of more complex scenarios that could otherwise not be completed in a convenient amount of time. Garcia et al. [15] compared the performance of brute force k nearest neighbor search on a standard C implemenation, a Matlab implementation, and a CUDA implementation, also comparing the performance of a more efficient kd-tree based method in a standard C implementation. They found that the CUDA method performed at a greater measure of efficiency as the number of dimensions and the scale of the problem grew. In their test results, complex scenarios resulted in CUDA implementation speed-ups of up to 120 times the standard C implementation. Their data is shown in Fig. 9.

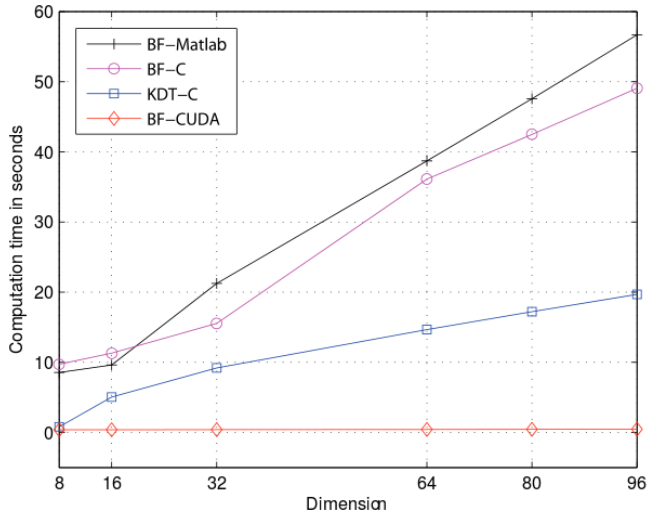

Fig. 9. Computation Time for the K Nearest Neighbor Search Problem [15]

2) Constraint Satisfaction: Constraint satisfaction problems involve a collection of variables and a collection of constraints on these variables. The solution is found by determining values for all variables such that all constraints are satisfied simultaneously. This kind of problem has applications in scheduling and satisfiability checking, among other uses. Additionally, many kinds of problems can be put into constraint satisfaction problems, allowing for techniques developed to solve these problems to be used for a large amount of problems. This style of programming is often applied to combinatorial problems, which are NP-hard [7]. Due to these applications, constraint satisfaction problems can become extremely complex depending on the number of variables and the nature of the

constraints.

One large component of constraint solvers is the constraint propagation engine. This is a process that must be executed frequently - after any variable is assigned a value (whether through a deterministic or stochastic method), it is necessary to then propagate this information through the list of constraints to restrict or set other variable values based on this new information. Campeotto et al. [7] implement a CUDA-based constraint propagation engine for use as part of a constraint solver. In their experiments, a CPU is still used to run the actual application, while their GPU is used only in situations which require a large amount of propagation that can be done in parallel. Through adding this GPU-based propagation engine, they reach speed-ups ranging from one to three times the pure CPU implementation. These speed-ups grow as they introduce more complex constraint systems and larger number of variables.

Single-walk and multi-walk approaches to constraint search problems use local neighborhood search to find solutions. Single-walk relies on a starting potential solution in the solution space, then evaluates the neighbors of the solution and moves toward the more promising solution. This continues until a local optimum is reached. Multi-walk uses a similar process, but utilizes multiple starting solutions to avoid becoming stuck at a local optimum. Arbelaez et al. [16] used a strategy that implemented parallelism at both the multi-walk and single-walk level to solve constraint search problems. At the multi-walk level, individual potential solution points could be handled in parallel to provide speed-up of the process. Additionally, at the single-walk level, the neighborhood of an individual potential solution can be evaluated in parallel, allowing for further speed-up. Using this approach, the researchers were able to obtain speed-ups of up to 17 times the speed of the sequential approach.

D. Robotics

Robotic control provides a large area of study within artificial intelligence, and this field continues to grow as robots become more prevalent. A significant amount of the computation required for many autonomous robotic control systems is involved in environment mapping. There are many techniques developed for this task, but it always requires utilizing sensor data to build an internal representation of the robot's environment which can be used for localization purposes, path planning, or trajectory modelling. While this mapping can be computationally intensive, it is also critical for many applications that it is done efficiently as robots must typically be controlled in real-time. Tuck et al. [17] attempted to leverage GPU architecture to optimize the speed of their existing robot control system. Their system utilized several different modules for various control tasks. They found that using a GPU implementation on the mapping module, specifically, enabled speed-ups of up to two orders of magnitude over their results using a CPU implementation. Their results are shown in Fig. 10.

Compute Kernel	Time (before)	Time (after)
Pathfinding (CPU)	0.038	0.156
Mapping	0.039	0.006
Map Merging	1.150	0.004

Fig. 10. Speed-up of Mapping using GPU [17]

Research into perception for robots investigates the way robots obtain and process information from sensors about their environments or their current state to build a model of their position relative to the environment. Ferreira et al. [18] investigated the use of GPUs through using CUDA in their investigation of applying a Bayesian model for multimodal perception.Algorithms that involved Bayesian models for this task were previously too slow to be able to run in real-time, which is required for perception. The researchers were able to speed-up processes that previously took 30 minutes to completion in a tenth of a second. This allowed their techniques to be used in real-time situations involving multimodal perception.

A relatively new branch of robotics called developmental robotics involves systems of control for robots which utilize autonomous "mental" development to gain competencies after interacting with their environment. This approach is growing in popularity as a way to study and experiment with not only robotic control, but also embodied intelligence and cognitive development in general. Peniak et al. [19] utilized GPUs when creating a software application for research into developmental robotics. The researchers found that their GPU modules obtained speed-ups of up to 1004 times comparable modules implemented on CPUs. Many of these modules involved tasks such as training neural networks which, as described previously, can be performed many times more efficiently on GPUs than on CPUs.

V. Limitations

The application of GPUs to artificial intelligence tasks has thus far proven successful in a variety of uses, as demonstrated in the previous section. Tasks which require processing of large amounts of data provide some natural opportunities for parallelism to be implemented through taking advantage of the architecture of GPUs. As GPUs continue to develop in both capability and general usability, this trend will undoubtedly continue, especially considering the growing marketing of GPUs to artificial intelligence researchers by major GPU manufacturers.

However, the transition to GPUs is not free. For one, as mentioned previously, the focus on throughput over latency in GPU development results in cores far less powerful than those of CPUs. Likewise, memory access is not as fast on GPUs and memory behavior may vary significantly between models. In order to take advantage of parallelism while also maintaining efficient behavior within these mentioned constraints of GPUs, it is necessary for software to be highly specialized. Though the difficulties in programming GPUs has been tremendously

helped by the introduction of CUDA and the move of GPU architectures toward general purpose computing, it is still a large task to develop software or move existing software to take full advantage of GPU architecture. Sufficient speed-ups must be attainable for this further development to be worthwhile.

A. Amdahl's Law

GPUs with rapidly growing numbers of cores do not provide the potential for unlimited speed-up. Rather, the potential of speed-up by utilizing the parallel architecture of a GPU is limited by Amdahl's Law. Amdahl's work on parallel computing proved several limits that apply to the power of parallelism. He demonstrated first that the factor by which the performance of a task increases due to parallelism must always be less than the factor by which the number of cores increases, due to some required sequential tasks and parallelism overhead. This overhead includes tasks such as the actual creation and destruction of threads, monitoring of access to data used by multiple threads, and cross-thread synchronization [20].

Thus, the factors involved in determining the speed-up include not only includes the number of cores, but also the percentage of workload that must be performed sequentially and the overhead of parallelizing however many threads are used. The work by Amdahl to determine the limits of speed-up possible by parallelism is synthesized into Amdahl's Law. Amdahl's Law, expressed as an equation that determines the speed-up using N cores and demonstrates the implicit limits to parallelism, is given in Fig. 11.

$$\text{Speedup (N)} = \frac{1}{S + \frac{1 - S}{N}} - O_N$$

Where:
N = Number of processor cores
S = Serial percentage of workload (expressed as a decimal in the range 0-1)
O_N = Parallelization overhead for N threads

Fig. 11. Amdahl's Law [20]

Amdahl's Law essentially states that parallelism has a finite maximum speed-up factor due to mandatory sequential processes and overhead. This is a restriction that cannot be removed by addition of more cores or even architecture improvements, but will always remain present. This law can be used as justification to focus on research and development of processor power and latency rather than throughput, as sequential processing will always be a limiting factor even in parallel computing. However, some have disagreed that Amdahl's Law sufficiently interprets the situation.

B. Gustafson's Trend

Gustafson's Trend or Gustafson's Law finds Amdahl's Law incomplete in its analysis, as it posits that further developments in computing power do not occur in isolation. Rather, these developments are typically accompanied by a growth of the data used by the problem set as applications grow to take full advantage of the technology available. The result is that the sequential portion of the process remains close to the same while a larger amount of data means a larger amount of parallelizable tasks [20]. Thus, the percentage of the task which must be processed sequentially can grow significantly smaller as the amount of data for a given task grows. In this way, parallelism can continue to provide further speed-ups through changes in the task despite the inherent limits of overhead and sequential processes.

Gustafson's Trend can be seen clearly in the field of neural networks. As described previously, the growth of neural networks has resulted in deep networks with several layers and a growing number of nodes. Through this growth of data, the potential for speed-ups continues to increase as predicted by Gustafson. As the current trend of neural networks continues to be towards larger, multi-layered networks, Gustafson's Trend provides a way to take this growth of data into account when predicting the potential speed-ups from parallelizing the processing of these networks. The result is that research into the usage of GPUs for parallelism remains worthwhile as potential speed-ups should continue to grow based on this trend. A version of Gustafson's Trend is shown in Fig. 12.

$$\text{Speedup (N)} = \frac{S + N\,(1 - S)}{S + (1 - S)} - O_N$$

Where:
N = Number of processor cores
S = Serial percentage of workload (expressed as a decimal in the range 0-1)
O_N = Parallelization overhead for N threads

Fig. 12. Gustafson's Trend [20]

As demonstrated in this equation, even as data continues to grow, it is important to minimize the overhead required by parallelism. While the amount of sequential tasks tends to stay about the same as more data is added in, the overhead can increase substantially. This is because of much of the overhead may include tasks such as monitoring thread access to data and synchronizing across threads; tasks which become more complex as the number of threads increases and, in the case of monitoring access to data, can also become more complex as the amount of data increases. This is an important area for developers of GPUs to continue to focus so that the required overhead for parallel processing can be minimized. These equations are important for the future of GPUs as they demonstrate both the potential speed-ups and the limitations of parallel approaches, which may help determine the cost-benefit of further GPU development.

VI. Looking Forward

As GPUs have proven to be powerful tools for artificial intelligence applications, forward progress involves two major factors. First, development and implementation of programming techniques that enable large-scale parallelism is necessary for the successful transition of tasks from CPU to GPU. Second, continued research and development into GPU architecture design and hardware will provide better, more powerful GPUs.

A. Programming Techniques

Development of programming techniques for GPUs is critical to take advantage of their architecture. Ryoo et al. [21] analyze a GeForce 8800 GTX and optimization of programming for these GPUs using CUDA. The GeForce 8800 is a top model of NVIDIA's GeForce 8-series, which implemented the first of NVIDIA's architectures to provide a unified-shader architecture and thus allow powerful tools for general purpose computing. The researchers identify both principles for selecting tasks to be executed on a GPU and principles of programming for optimization on GPUs.

Several factors are found to be important for how well code will perform on a CUDA implementation. First, the application should hide memory latency by using thread scheduling that has no overhead. As mentioned previously, it is critical to reduce overhead of parallelism for optimum performance of GPU-implemented applications, and thread scheduling provides a way to set up threads to hide the latency of memory access without incurring overhead. Since memory latency can be a major bottleneck for applications, proper thread scheduling that maintains a high "compute-to-memory-access ratio" provide optimum usage. Next, while the usage of global memory is beneficial to avoid redundant storage, it is important to make use of on-chip memory and optimize this usage to reduce bandwidth usage. The more shared memory that is used, the fewer threads that can be simultaneously executed due to memory access issues. Then, threads should be grouped when applicable. This can help avoid problematic divergent control flow issues. Finally, it is important to consider communication between threads. The GeForce 8-series does not provide global communication between all threads, which allows more efficient processing due to avoiding virtualization of hardware resources. Threads within the same block can still communicate through synchronization. However, this applies a limitation to the kinds of tasks and the methods of parallelism which can be implemented using CUDA on this architecture. These four principles provide major considerations for choosing code to be implemented on a GPU [21].

Next, once an application is chosen for GPU implementation, the researchers lay out three principles for optimizing an application's performance on GPU. First, as also noted by Amdahl's Law, the percentage of an application's instructions that are floating point operations determines maximum floating point throughput of the application. Maximum performance is reached by keeping all processors occupied. Next, the researchers found that for many different applications, the key component to achieving maximum performance is successfully handling global memory latency. Again, keeping all processors occupied is the solution to this problem. Enough threads must be created to keep the processors busy while other threads wait on global memory access. Finally, the bandwidth of global memory can be the limiting factor on the entire system. The access to global memory must be balanced with use of shared memory to obtain optimum results. Through consideration of these three components, applications can be optimized for CUDA implementation [21].

To demonstrate the potential for parallelism of various tasks and show the relation between various factors and the speed-up achieved by GPU-implementation, Ryoo et al. tested GPU-implementations of various applications and recorded the results. The applications they used are shown in Fig. 13, and the results are shown in Fig. 14. As these tables demonstrate, all applications chosen achieved at least some speed-up, with the range of speed-up varying significantly based on factors including the number of threads utilized, global memory usage, and the percentage of the application executed on the GPU. Both the speed-up of the kernel on the GPU and the speed-up of the overall application are given.

Application	Description	Source Lines	Kernel Lines	CPU Execution Parallelized
H.264	A modified version of the 464.h264ref benchmark from SPEC CPU2006. This is an H.264 (MPEG-4 AVC) video encoder. A serial dependence between motion estimation of macroblocks in a video frame is removed to enable parallel execution of the motion estimation code. Although this modification changes the output of the program, it is allowed within the H.264 standard.	34811	194	35%
LBM	A modified version of the 470.lbm benchmark from SPEC CPU2006. This uses the Lattice-Boltzman Method for simulating 3D fluid dynamics. The program has been changed to use single-precision floating point and print fewer status reports.	1481	285	$>99\%$
RC5-72	This application accelerates distributed.net's RSA RC5-72 bit challenge, which performs brute-force encryption key generation and matching.	1979	218	$>99\%$
FEM	Finite Element Modeling. Simulation of dynamic behavior of 3D graded materials.	1874	146	99%
RPES	Rys Polynomial Equation Solver. Calculates 2-electron repulsion integrals, which are a sub-problem of molecular dynamics.	1104	281	99%
PNS	Petri Net Simulation. Simulation of a mathematical representation of a distributed system.	322	160	$>99\%$
SAXPY	Single-precision floating-point implementation of `saxpy` from High-Performance LINPACK, used as part of a Gaussian elimination routine.	952	31	$>99\%$
TPACF	Implementation of Two Point Angular Correlation Function, used to find the probability of finding an astronomical body at a given angular distance from another astronomical body.	536	98	96%
FDTD	Finite-Difference Time-Domain. 2D electromagnetic wave propagation simulation in an arbitrary, user-defined medium.	1365	93	16.4%
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	490	33	$>99\%$
MRI-FHD	Computation of an image-specific matrix $F^H d$, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	343	39	$>99\%$
CP	Computation of electric potential in a volume containing point charges. Based on direct Coulomb summation, as described in [24].	409	47	$>99\%$

Fig. 13. List of Applications Tested [21]

Application	Max Simultaneously Active Threads	Registers per Thread	Shared Mem per Thread (B)	Global Memory to Computation Cycles Ratio	GPU Exec %	CPU-GPU Transfer %	Architectural Bottleneck(s)	Kernel Speedup on GPU	Application Speedup
Mat Mul	12288	9	8.1	0.276	16.2%	4%	Instruction issue	7.0X	2.0X
H.264	3936	30	55.1	0.006	2.6%	4.5%	Register file capacity and cache latencies	20.2X	1.47X
LBM	3200	32	84.2	0.415	98.3%	0.4%	Shared memory capacity	12.5X	12.3X
RC5-72	3072	42	0.3	$\simeq 0$	64.3%	0.5%	Instruction issue	17.1X	11.0X
FEM	4096	18	61	1.135	91.4%	$\ll 1\%$	Global memory bandwidth	11.0X	10.1X
RPES	4096	23	24.8	0.01	37.5%	1%	Instruction issue	210X	79.4X
PNS	2048	32	9.9	0.241	98%	$\ll 1\%$	Global memory capacity	24.0X	23.7X
SAXPY	12288	7	0.3	0.375	88%	4.5%	Global memory bandwidth	19.4X	11.8X
TPACF	4096	24	52.2	0.0002	34.3%	$\ll 1\%$	Shared memory capacity	60.2X	21.6X
FDTD	12288	11	8.1	0.516	1.8%	0.9%	Global memory bandwidth	10.5X	1.16X
MRI-Q	8192	11	20.1	0.008	$>99\%$	$\ll 1\%$	Instruction issue	457X	431X
MRI-FHD	8192	12	20.1	0.006	99%	1%	Instruction issue	316X	263X
CP	6144	20	0.4	0.0005	$>99\%$	$\ll 1\%$	Instruction issue	102X	102X

Fig. 14. Results of GPU-implementation [21]

B. Technology Development

As GPUs have proven extremely powerful over the last decade for usage in artificial intelligence applications, especially deep learning, development of chips has very recently begun to be tailored to this application. In early April of 2016, NVIDIA announced a new chip, the Tesla P100, specifically designed for deep learning applications. This chip was the results of more than $2 billion of research and development by NVIDIA [22]. According to the whitepaper for this chip, significant speed-ups for neural network implementations will result not only from an improved architecture, but also from the use of FP16 data rather than the FP32 or FP64 used by previous chips [23]. Additional improvements include significant increase in number of cores as well as improvements on unified memory and the ability to interrupt tasks at instruction-level granularity. There are around 15 billion transistors on the

P100, which is roughly three-times as many transistors as on NVIDIA's previous chips [22].

In conjunction with the development of this chip, NVIDIA has released servers marketed as "the first AI supercomputers in a box", which are servers that are purpose-built for deep learning [23]. These machines, called the DGX-1, utilize 8 of the P100 chips. NVIDIA claims this server has already demonstrated speed-up of 12-times the top results from last year's top models for training deep neural networks, as shown in Fig. 15. This graphic demonstrates speed-up on Alexnet, a groundbreaking deep neural network from 2012 that is sometimes used for benchmarking of the training times for deep neural networks. The DGX-1 server comes packaged with the latest release of CUDA and NVIDIA's Deep Learning SDK, so it is primed for immediate usage in deep learning applications.

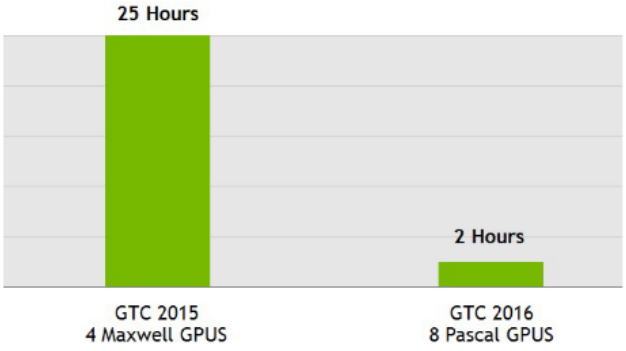

Fig. 15. Speed-up of DGX-1 [23]

This development, publicized during the writing of this paper, demonstrates the direction of GPU evolution moving forward. With NVIDIA taking the lead, GPU manufacturers have found a prime usage in deep learning applications. As continued research and development is done on this front, it is likely that a large segment of GPU development will become specialized for usage in these applications. This is also contingent on the continued usage of deep neural networks in both academic research and their implementation in industrial applications. As the usage continues to grow for deep networks, GPU manufacturers stand only to profit. Likewise, the development of specialized hardware will buoy the field of neural networks, providing the capability to implement more and more complex networks.

VII. Conclusion

The development of GPUs for use in graphics processing led to highly specialized units, but their design trend over the last decade has moved towards more general-purpose units. Their designs primary difference from traditional CPUs is the ability to conduct extremely large amounts of parallel processes and consequently have extremely high throughput computing abilities. Their evolution over the past two decades away from dedicated graphics usage and towards more general-purpose programmable devices has allowed for their implementation in various applications outside the traditional use of graphics rendering. This trend has been encouraged by major manufacturers of GPUs, and is consequently likely to continue with the development of future generations of GPU design.

The massively parallel structure of these units makes them ideal for many tasks within the field of artificial intelligence, and research over the last decade supports the use of GPUs in this field as preferable to traditional CPUs for many tasks. GPUs have exhibited powerful speed-ups over CPUs in solutions to various artificial intelligence problems involving neural networks, optimization problems, search problems, and even robotic control. Furthermore, these speed-ups have proven to grow larger as the scale of the problem grows. This is significant, as it allows for complex versions of these artificial intelligence techniques to be implemented on GPUs that would not even be possible on CPU implementations. Such a development has allowed research into complex versions of these solutions for problems that would have been too difficult for CPU implementation of solutions to handle. Continued development of solutions to tasks that use GPUs will involve further optimization of solutions to take advantage of the structure of GPUs through a growing focus on parallel processing.

References

[1] D. Luebke and G. Humphreys, "How gpus work," *IEEE Computer*, vol. 40, no. 2, pp. 96–100, 2007.

[2] M. Macedonia, "The gpu enters computing's mainstream," *Computer*, vol. 36, no. 10, pp. 106–108, 2003.

[3] S. Collange. Parallel programming: Introduction to gpu architecture. Lecture. [Online]. Available: http://www.irisa.fr/alf/downloads/collange

[4] D. Luebke, "Gpu architecture: implications & trends," *ser. SIGGRAPH*, vol. 2008, 2008.

[5] Y. Drew, "A closer look at gpus," *Communications of the ACM*, vol. 51, no. 10, 2008.

[6] Nvidia, "Nvidia's next generation cuda compute architecture: fermi," Tech. Rep., 2009.

[7] F. Campeotto, A. Dal Palu, A. Dovier, F. Fioretto, and E. Pontelli, "Exploring the use of gpus in constraint solving," in *Practical Aspects of Declarative Languages*. Springer, 2014, pp. 152–167.

[8] Deep learning using gpus. NVIDIA. [Online]. Available: https://developer.nvidia.com/deep-learning

[9] D. L. Ly, V. Paprotski, and D. Yen, "Neural networks on gpus: Restricted boltzmann machines," *see http://www. eecg. toronto. edu/˜ moshovos/CUDA08/doku. php*, 2008.

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

[11] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 873–880.

[12] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th international conference on machine learning*, 2013, pp. 1337–1345.

[13] S. Harding and W. Banzhaf, "Fast genetic programming on gpus," in *Genetic Programming*. Springer, 2007, pp. 90–101.

[14] Y. Zhou and Y. Tan, "Gpu-based parallel particle swarm optimization," in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE, 2009, pp. 1493–1500.

[15] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*. IEEE, 2008, pp. 1–6.

[16] A. Arbelaez and P. Codognet, "A gpu implementation of parallel constraint-based local search," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE, 2014, pp. 648–655.

[17] N. Tuck, M. McGuinness, and F. Martin, "Optimizing a mobile robot control system using gpu acceleration," in *IS&T/SPIE Electronic Imaging*. International Society for Optics and Photonics, 2012, pp. 83 010Z–83 010Z.

[18] J. F. Ferreira, J. Lobo, and J. Dias, "Bayesian real-time perception algorithms on gpu," *Journal of Real-Time Image Processing*, vol. 6, no. 3, pp. 171–186, 2011.

[19] M. Peniak, A. Morse, C. Larcombe, S. Ramirez-Contla, and A. Cangelosi, "Aquila: An open-source gpu-accelerated toolkit for cognitive and neuro-robotics research," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE, 2011, pp. 1753–1760.

[20] M. Gillespie, "Amdahl's law, gustafson's trend, and the performance limits of parallel applications," 2008.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[22] T. Simonite, "A 2 billion dollar chip to accelerate artificial intelligence," *MIT Technology Review*, April 2016. [Online]. Available: https://www.technologyreview.com/s/601195/a-2-billion-chip-to-accelerate-artificial-intelligence/

[23] Nvidia, "Nvidia tesla p100 whitepaper," NVIDIA, Whitepaper, April 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf