Scheduling Computation Graphs of Deep Learning Models on Manycore CPUs

Linpeng Tang*, Yida Wang†, Theodore L. Willke†, Kai Li*
*Princeton University, †Intel Corporation

Abstract

For a deep learning model, efficient execution of its computation graph is key to achieving high performance. Previous work has focused on improving the performance for individual nodes of the computation graph, while ignoring the parallelization of the graph as a whole. However, we observe that running multiple operations simultaneously without interference is critical to efficiently perform parallelizable small operations. The attempt of executing the computation graph in parallel in deep learning frameworks usually involves much resource contention among concurrent operations, leading to inferior performance on manycore CPUs. To address these issues, in this paper, we propose *Graphi*, a generic and high-performance execution engine to efficiently execute a computation graph in parallel on manycore CPUs. Specifically, *Graphi* minimizes the interference on both software/hardware resources, discovers the best parallel setting with a profiler, and further optimizes graph execution with the critical-path first scheduling. Our experiments show that the parallel execution consistently outperforms the sequential one. The training times on four different neural networks with *Graphi* are 2.1× to 9.5× faster than those with TensorFlow on a 68-core Intel Xeon Phi processor.

1 Introduction

Manycore processor architectures utilize many relatively low performance cores to achieve high overall performance [51, 60]. The architecture is particularly well-suited to high-performance computing (HPC) applications with lots of data parallelism, due to its large number of computing cores and wide vector processing units. One such application is deep learning [37], whose models can be expressed as computation graphs with nodes representing the operations and edges representing the dependencies between nodes [7] (more details in Section 2).

The efficiency of processing computation graphs on contemporary computing devices, especially graphic processing units (GPUs), has been extensively studied in the literature[11, 34, 53, 58]. Among these, many have focused on building efficient primitives to speed up single operations on one processor [27, 61] or optimizing distributed execution across multiple processors with a server [10, 25] and across a cluster [39, 62]. So far, little effort has been put into the scheduling of computation graphs on manycore processors. Some previous methods use one executor to run a computation graph operation-by-operation sequentially on GPUs [8, 30]; others use a naive way to allow multiple executors to run simultaneously [2, 10], which introduces contention between threads sharing computing and memory resources. These approaches result in the substantial under-utilization of CPUs, the most popularly available computing resource, on the deep learning workloads.

In this paper, we study how to efficiently execute computation graphs of deep learning models on manycore CPUs. Our experimental hardware platform is the Intel Xeon Phi processor, based on the Intel Many Integrated Core architecture (MIC) [51]. We show that sequential execution normally cannot exhaust the available resource of this processor and that naive parallel execution typically achieves poor performance

mainly due to sub-optimal thread scheduling and thread interference. Based on these observations, we propose *Graphi*, a generic high-performance execution engine for computation graphs on manycore CPUs. Our key idea is to profile a given computation graph, allocate resources to different agents (scheduler and executors) of the execution engine using the profiling results, and schedule operations intelligently with minimal interference. We compare running our execution engine on the manycore CPU with TensorFlow [2] on the same hardware.

To the best of our knowledge, *Graphi* is the first high-performance parallel execution engine with intelligent scheduling strategies for deep learning computation graphs on manycore CPUs. Although many of the techniques in *Graphi* are not new, our unique contribution is to identify issues of current deep learning frameworks on manycore CPUs, adapt proper techniques to a complex deep learning system, and make them work in synergy to greatly boost the overall performance. Moreover, we believe the concepts captured by *Graphi* can be incorporated into mainstream frameworks. Specifically, this paper makes the following contributions:

1. We demonstrate that by choosing proper parallelism scheme and using optimized scheduling, *Graphi* outperforms TensorFlow on the Intel Xeon Phi processor by 2.1× to 9.5× on 4 popular deep learning networks.

2. We demonstrate that operations typically used in deep learning models (e.g. matrix multiplication and element-wise operation) saturate at 8 or 16 cores on the Intel Xeon Phi processor, and parallelizing multiple operations without unnecessary thread interference is preferable. We show that parallel execution outperforms sequential by up to 3.4×.

3. We demonstrate that using a centralized scheduler to impose intelligent scheduling and eliminate software resource contention between autonomous executors further boosts overall performance of the execution engine by up to 19%.

The rest of the paper is organized as follows: Section 2 provides background on computation graphs and manycore CPUs, Section 3 delves deeper into the motivation for our work and the challenges we have via microbenchmarking of the manycore CPU. We present the overall design of *Graphi* in Section 4 and its implementation in Section 5. Section 6 discusses other optimization we considered during the system design. The evaluation is in Section 7, followed by the discussion of related work in Section 8. Section 9 summarizes the paper and proposes the future work.

2 Background

This section discusses the background of the two main aspects of the paper: computation graphs and manycore CPUs.

Computation graph The computation graph is a common way to specify computation tasks and their dependencies for execution [32]. This abstraction has found wide usage in dataflow computation [3, 15, 16, 22] and streaming data processing [9, 59]. Recently, deep learning frameworks such as TensorFlow [2], MXNet [10], neon [43], Theano [8] and Caffe [30] have used computation graphs to represent the required computation of deep learning models after compilation. A computation graph is a directed acyclic graph (DAG) with each node representing an operation which could be a matrix multiplication, a convolution or an element-wise operation, etc. A directed edge pointing from node A to B represents that operation B is dependent on operation A, i.e. the output of operation A serves as (part of) the input of operation B.

The training and inference of a deep learning model is essentially the execution of the corresponding computation graph. Training requires a larger computation graph which consists of both forward operations for computing the loss and backward operations for computing the gradients. A complete execution on the graph corresponds to one training iteration of a batch. The computation graph for inference is smaller since it only contains the forward operations. One complete execution of the graph typically results in the inference of a group of instances.

The execution engine of a computation graph uses an important abstraction *executor* to lead a team of

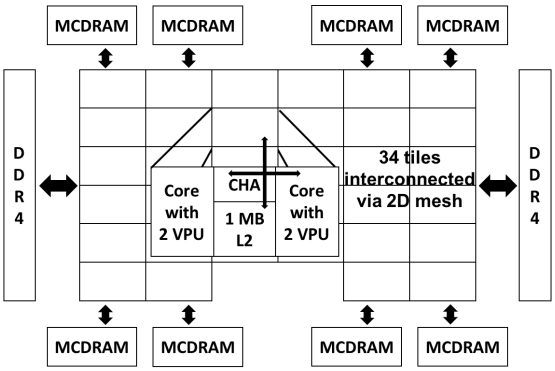


Figure 1: Architecture of Intel Xeon Phi processor 7250.

threads to run an operation at a time. The team of threads allows the executor to use thread-level parallelism to execute the operation efficiently. The size of the team can be configured or adjusted by the execution engine.

The conventional way of interpreting a computation graph is to execute operations in sequence according to a topological order of the graph. That is, starting from an operation with no dependencies (i.e. no other operations pointing to it), the execution engine picks one executable operation at a time to run. An operation is executable only after all operations pointing to it have finished (if any).

The sequential execution approach requires only one executor, and improves performance by exploiting the parallelism within each operation to utilize the available SIMD and multi-thread parallelism of a CPU or GPU. This method works well when the computation graphs have large operations and simple structures (e.g., AlexNet [34]).

A more advanced way is to execute operations with multiple executors in parallel when necessary, which is under explored and not optimized in popular frameworks. It holds the promise for improving the overall performance of computation graph execution on many complex networks.

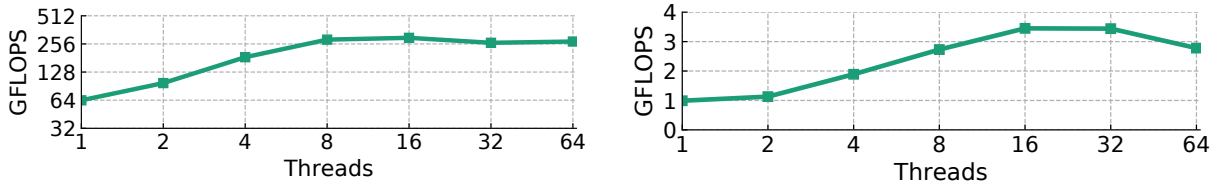
Manycore CPU This paper studies how to efficiently execute a computation graph on a manycore CPU. Our experimental hardware is a 68-core Intel Xeon Phi processor 7250 (code named Knights Landing or KNL, referred to hereinafter as the manycore CPU) based on the Intel Many Integrated Core (MIC) architecture. It allows the use of parallel programming toolkits such as OpenMP in the same way as programming on a typical multicore x86 processor. The processor runs at a clock frequency of 1.40 GHz and supports up to 4 hardware hyper-threads per core. Our experiments used one thread per core to eliminate interference among hardware threads running on the same core while achieving good performance.

Each core has 32 KB L1 data cache, and 32 KB L1 instruction cache. Every two cores are organized as a *tile* with 1 MB shared unified L2 cache. All tiles are interconnected as a 2D mesh. Cache coherence is maintained via a distributed directory provided. The mesh supports three modes of tile clustering (all-to-all, quadrant, and sub-NUMA clustering) to provide different levels of memory address affinity for better overall performance in different use cases. These cluster modes aim to lower latencies and improve bandwidth by reducing the distance of data traversals within the chip [54].

In this paper, the manycore CPU is configured in the *quadrant* mode, which offers symmetric memory access. For a more sophisticated system design, the manycore CPU may use the *sub-NUMA clustering* mode for better performance. The manycore CPU is equipped with a 16 GB multi-channel DRAM (MCDRAM) with bandwidth greater than 400 GB/s, as well as 96 GB DDR4 memory. Figure 1 depicts the architecture. Understanding of this architecture helped to shape the design of our execution engine in Section 4.

3 Motivations

This section first discusses the challenges of executing computation graphs on a manycore CPU efficiently, then performs microbenchmarking on the manycore CPU to further investigate the issues, which motivates the design of *Graphi*.



(a) GEMM **(b) Element-wise Multiplication**

Figure 2: Scalability of GEMM/element-wise multiplication operations in a typical LSTM on Intel Xeon Phi processor 7250.

3.1 Challenges

The conventional way of executing computation graphs in sequence does not work well on the many-core CPUs for networks with smaller operations and complex linking structures (e.g. long short-term memory (LSTM) [28], PathNet [20]). These networks have operations that are too small to fully take advantage of the compute power of all cores because of the thread management overhead, which is getting worse as the number of cores increases. In order to obtain better resource utilization, multiple operations should be run in parallel.

Fortunately, independent operations in a computation graph are always parallelizable. Modern frameworks like TensorFlow and MXNet provide parallel execution engines that can execute more than one operation at the same time. Nevertheless, these frameworks have not been carefully optimized for manycore CPUs, and the challenges described below can hinder their execution efficiency on the hardware.

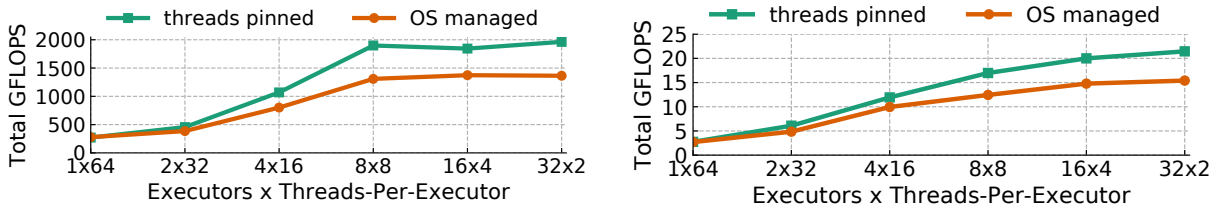
The first challenge is how to optimally schedule the operations of a computation graph on a number of operation executors. Scheduling M operations expressed as a DAG to run on N executors to minimize the *makespan*, the total time from execution starts until every operation finishes, is a well-known NP-hard problem by reduction from the 3-partition problem [38].

The approach TensorFlow and MXNet have taken is to maintain a centralized queue of the executable operations without dependencies, and allows an arbitrary executor to execute any operation that is at the head of the queue. Once the dependencies of an operation are all executed, it will be placed onto the queue. The scheduling continues until all operations are executed. This naive scheduling strategy is simple but may not perform well on the manycore CPU.

Another challenge is how to eliminate the interference among executors, which is commonly seen in the modern deep learning frameworks, leading to performance reduction of the parallel execution engine and even decreases the overall performance compared with sequential execution.

For instance, TensorFlow and Caffe2 (an updated version of Caffe) [55] use Eigen Library [1] as well as OpenMP for different operations, each with their own thread pool, which results in more software threads than available physical cores. This over-subscription causes either unnecessary resource contention or expensive thread context switching on the manycore CPU. Moreover, those frameworks do not explicitly specify on which cores the threads should run, making it likely for execution threads to compete for the same physical cores. Such contention can cause threads to straggle, and further hampers overall performance.

Contention may happen over software resources as well. One example is the centralized queue of the executable operations. When the number of executors is large and the execution time of an operation is small, the overhead of global queue polling contention becomes significant. In general, contention on the manycore CPU could be very severe due to its large number of cores.



(a) GEMM **(b) Element-wise Multiplication**

Figure 3: Performance of parallel operations with pinned vs. OS managed threads on Intel Xeon Phi processor 7250. There are multiple executors, each running GEMM/element-wise multiplication operations with a team of threads.

3.2 Microbenchmark Performance

We designed two microbenchmarks to validate the following two performance characteristics of the manycore CPU: 1) the scalability of small operations degrades at some point and 2) running multiple small operations in parallel without interference is beneficial. Although these concepts are well-studied in general, we do not see them embodied for typical deep learning operations on the manycore CPU. Therefore, we find it critical to validate them before designing a scheduling system for deep learning workloads on the manycore CPU.

The first microbenchmark was used to assess the scalability of a manycore CPU for small operations. This benchmark includes two commonly used operations in the computation graph of LSTM: matrix multiplication (GEMM) of size $[64, 512] \times [512, 512]$ implemented via Intel Math Kernel Library (MKL), and element-wise multiplication for 32 768 element pairs multi-threaded via OpenMP. The specific sizes are chosen to represent the medium size of LSTM suggested in the standard TensorFlow benchmark.

Figure 2 shows that the performance of GEMM saturates when the number of threads is greater than 8, whereas the element-wise multiplication saturates when it is greater than 16. Therefore, dedicating all available computing resources of the manycore CPU to a single operation is not optimal and the parallel computing power is largely wasted.

The second microbenchmark was designed to show the effect of resource contention within the manycore CPU. This microbenchmark consists of multiple GEMM and element-wise multiplication instances. The sizes of these two operations are the same as those in the first microbenchmark. We ran the microbenchmark in two modes on the same manycore CPU: manually pinning different threads to different physical cores and leaving the thread assignment to OS.

Figure 3 shows that the overall FLOPS of operations with threads pinned is higher than OS managed by up to 45%. This is because the OS is unaware of the layout of the physical cores of the manycore CPU, so it is likely that multiple threads ran on the same physical core, which can cause synchronization overheads and cache misses. Since a modern manycore CPU has private caches within each tile, the scheduling of executors on physical cores should be architecturally aware to reduce such cache contention. Pinning threads to cores properly removes such overheads.

By comparing the peak FLOPS in Figure 2 and Figure 3, we can see that the overall performance of running multiple small operations together without interference is more than 6× faster than running one single small operation using all the available resources. Such results validate the value of running multiple small operations in parallel.

Based on our benchmark experiments, we argue that an efficient parallel execution engine should avoid contentions on the manycore CPU and should execute small operations of a computation graph in parallel.

4 *Graphi* Design

We propose *Graphi*, a generic and high-performance execution engine to efficiently run computation graphs of deep learning models on the manycore CPU. *Graphi* has multiple kinds of agents, e.g. profiler, scheduler, and executor, whose functions will be described in detail later. We keep the following goals in mind while designing the system:

1. The system should be general purpose, and be able to execute different kinds of neural networks;

2. Given a computation graph, the system should be able to schedule and execute the operations in a way that minimizes the *makespan*.

3. In presence of a fleet of multi-threaded executors and a centralized scheduler, the system should avoid interference across these agents.

4.1 Overview

Figure 4 shows the architecture of *Graphi*. *Graphi* has two kinds of inputs: a compiled computation graph of a deep learning model and the number of cores of a manycore CPU. We assume that the computation graph is static, meaning that the graph will not change during the entire computation.

In order to fulfill design goal (1), *Graphi* is designed to be agnostic to the underlying neural network, only seeing the computation graph as a DAG. For an arbitrary DAG, *Graphi* will first profile it for a better scheduling strategy during In the initial few runs.

The *Graphi* profiler works with the execution engine in a feedback loop as shown in the upper part of Figure 4: the execution engine runs the graph to generate statistics and informs the profiler, while the profiler uses these statistics to collect information and improve scheduling. The execution engine then uses this information to optimize subsequent runs, therefore fulfilling design goal (2). In general, *Graphi* works to avoid possible hardware and software resource contention following (3).

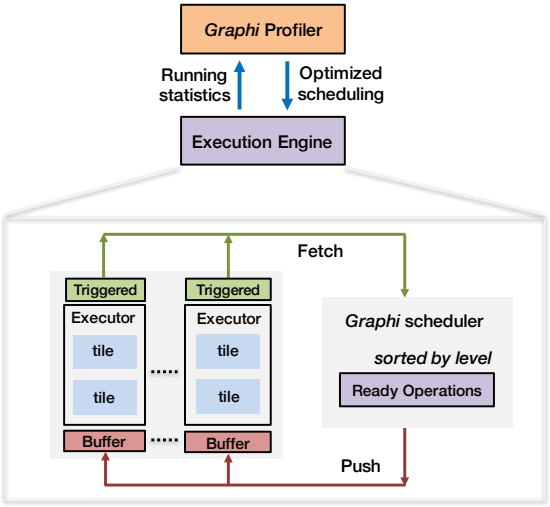


Figure 4: *Graphi* design overview.

4.2 Profiler

The *Graphi* profiler has two goals. The first is to determine the basic configuration for execution. Given the number of available cores, it comes up with different combinations of number of executors and threads per executor in order to find one with minimal execution makespan. For simplicity (see Section 6 for more discussions), we make the executors symmetric, i.e. all executors own the same number of threads. In this way the profiler only needs to enumerate through a small number of configurations. For example, assume there are 64 threads, then we may have 1 executor with 64 threads, or 2 executors with 32 threads each, etc., up to 64 executors with 1 thread each. The *Graphi* execution engine will then use the selected combination for the subsequent optimizations.

The second goal is to estimate the running time of each operation of the computation graph with the selected combination. This adds modest overhead (i.e. execution time) to the system, but since the computation graph is static, this information can be treated as invariant and only needs to be collected for the first few iterations. It can then be used by the scheduler to further improve the scheduling process.

4.3 Scheduler

Graphi uses a centralized scheduler to coordinate the operations running on different executors, which simplifies the system processing pipeline by concentrating the scheduling decisions to a single agent. The lower part of Figure 4 illustrates the design of our scheduler, and Algorithm 1 summarizes its workflow.

The centralized scheduler oversees the execution status. It keeps polling for the newly triggered operations from the executors, as well as allocating ready-to-run operations to the executors. The scheduling is done once all operations are executed. Since the scheduler has knowledge of the system state, i.e. which operations are executable and their dependencies with the other operations, it can make strategy accordingly of sending which operation to run once an executor becomes available. One analogy of this design choice is the centralized software defined networks (SDN) [24] compared with the traditional decentralized network protocols.

Although we know the graph structure and the estimated executing times of all operations by profiling the graph, an optimal offline scheduling solution is not feasible because there are unpredictable variations at run time. These variations can cause empty cycles spent waiting for operation dependencies. Therefore, we focus on an online scheduling solution.

We design the scheduler using an online algorithm [29] to prioritize the operations in the critical path of the DAG. Specifically, from the information of the computation graph structure and the estimated running time of each operation, we can derive a *level* value for each operation, which is defined as the longest accumulated time from this operation to the end (sink point) of the computation graph. *Graphi* sorts the read-to-run operations according to their level values decreasingly and always schedules the operations with higher level value first. In other words, the operations in the critical path are prioritized for earlier execution so that they will not become the bottleneck. We call this *critical-path first scheduling*.

The *Graphi* scheduler design improves upon existing scheduling schemes in two ways. First, it eliminates potential software resource contention. In the parallel execution engines of TensorFlow and MXNet, there is only one queue that maintains the operations ready for execution (i.e., have no dependencies or all dependencies are satisfied). All executors independently poll the same queue for the next operations. This results in a heavy contention on the global queue, especially when the number of executors is large and the execution time of an operation is small. *Graphi* avoids this issue by having the scheduler push operations to executor-specific operation buffers. Since the buffers are disjoint, the interference between polling executors is eliminated.

Algorithm 1 *Graphi* Scheduler

```
1: while hasPendingOperations() do
2:     Poll triggered operations from each executor
3:     Sort ready operations based on their level values
4:     while hasReadyOperations() and FindExecutor(e) do
5:         Get operation p with the maximal level value
6:         Put p into executor e's buffer
7:     end while
8: end while
```

The other advantage of the centralized scheduler is that it gives us flexibility to use different advanced scheduler polices. Current scheduling strategy is critical-path first, but the architecture allows us to easily implement other strategies. This is not possible in other state-of-the-art parallel execution engines, which lack a centralized scheduler. As discussed in Section 3, those execution engines schedule the operations in an arbitrary topological order, that is, whenever an executor is available, it randomly picks a ready operation to run. Since all executors work greedily, a global optimization strategy cannot be imposed.

4.4 Executors

In addition to the centralized scheduler, *Graphi* uses a fleet of executors harnessed by the scheduler. The number of executors, as well as the team size of threads per executor, are determined by the profiler. The executors are in charge of executing operations assigned

to them by the scheduler. Its workflow is shown in Algorithm 2.

As discussed in Section 4.3, the executors only need to poll their own operation buffers for operations to execute. In addition, to further reduce contention, each executor is also associated with its own triggered queue, where it outputs the triggered operations upon finishing one operation. These are then fetched by the scheduler for processing.

Algorithm 2 *Graphi* Executor

```
1: while true do
2:     /* Poll the buffer for new operation */
3:     if GetOperation(p) then
4:         Execute p with the team of threads
5:         Trigger p's depending operations
6:     end if
7: end while
```

In our design, we assign each executor exclusively to a number of tiles (see lower part of Figure 4), each of which consists of two physical cores and an exclusive L2 cache as described in Section 2. As a result, executors do not share the compute units nor L2 cache, and consequently the hardware resource contention we discussed in Section 3 is largely avoided.

5 Implementation

The implementation of *Graphi* leverages the computation graph toolkit (CGT) [50]. The main reasons for choosing CGT are its modular design of compilation and execution and its small code base. We use CGT's compilation component to compile a deep learning model into a computation graph. But, we add the profiler component, and completely redesign and reimplement the execution engine. These are the focuses of this paper, and the implementation can also be migrated to other deep learning frameworks.

This section gives an overview of a typical deep learning framework like CGT, and then describes the implementation issues in *Graphi*.

5.1 Overview of CGT

CGT consists of two main parts: 1) a *compiler* to compile from a model into a computation graph and 2) an *execution engine* to run the graph. The model definition is constructed in Python, represented as mathematical expressions relating to the inputs, the intermediate variables, and the outputs. The compiler translates high level expressions to a low level computation graph targeting a particular processor, in our case the manycore CPU.

Each variable will be assigned a memory location, and optimizations during compilation allow multiple variables to share the same location as long as their lifespans do not overlap. Compute-intensive operations implemented in C++ are also compiled into shared libraries as callable routines.

The execution engine, as well as the data structures of the computation graph, including variables, operations, and their dependencies, are also written in C++ for efficiency. The engine is discussed in more detail in the next susbsection.

5.2 Profiler and execution engine

As described previously, *Graphi* has three components: the profiler, the centralized scheduler, and a fleet of executors. This subsection discusses their implementation details.

Profiler. After determining the best configuration for graph execution, the profiler records the information of each operation for several runs, which includes start and end time, the input/output data address and size, as well as the executor running it. The computed duration is averaged over multiple iterations to reduce variance, and then it is used in the critical-path first scheduling. The data addresses help analyze data locality (see Section 6 for our attempts on improving locality). In addition, we use the profiling results to visualize the execution process , i.e. placing the operations to their running executors' timelines. This has been immensely helpful in analysis and debugging. Normally, the profiler only runs in the first few iterations, adding minimal overhead to a typical deep learning workload running for thousands of iterations.

Scheduler. We dedicate the client thread initiat-

ing the graph execution to run the scheduler, and it works in a busy loop. In each iteration, it first polls for the newly triggered operations. In order to do the critical-path first scheduling, it then maintains the operations in a *max binary heap* ordered by their *level* values, and these operations can be fired once an executor becomes available.

In order to efficiently check the available executors and make advanced scheduling decisions, the executor states are represented as a bit map, with 1 denoting the executor is idle, and 0 denoting busy. We use bit-scan intrinsics to find the number of trailing zeros in the bit map, which corresponds to the first executor now available to run. The scheduler then pushes the operation at the head of the heap to that executor's operation buffer.

Executors. Each executor polls for operation from its own operation buffer, executes the operation, and triggers this operation's dependencies. The operation buffer is implemented with a lock free ring buffer for high efficiency. This implementation is inspired by the per-thread run queue of MuQSS [33] scheduler, and enables us to buffer multiple operations to further reduce scheduling overheads and apply more sophisticated scheduling schemes.

In practice, we find that the load imbalance caused by a larger buffer size offset the benefit, so we buffer at most one operation in *Graphi*. Each executor spawns a team of OpenMP threads to run the operations. Our implementation chooses an even number of threads such that no two executors shared a tile (see Section 2), consequently avoiding L2 cache interference among executors.

We use the primitives in several software packages including LIBXSMM [27] for convolutions, Intel MKL for matrix multiplications, and OpenMP for loop of element-wise operations. The engine uses OpenMP for thread management. As long as the size of the thread team of an executor does not change across different operations, the OpenMP library will always reuse the same team of threads, with the executor thread being the master.

Before one executor launches, it creates an OpenMP parallel region for its team of threads, in which each thread in the team is pinned to a specific core. During the execution of subsequent operations, the thread will stay on the same core. We find this setting important for high performance because it eliminates resource contention as well as the overheads from thread migrations and context switches.

The executors discussed above are designed to run expensive operations such as convolutions, matrix multiplications, and large element-wise operations. In addition, there are also small operations like scalar addition in the computation graph. Both CGT and TensorFlow employ an optimization to directly run these small operations in the current thread/executor instead of pushing them to the ready-operation queue. We adopt the same idea in *Graphi*.

It is also worth noting that bootstrapping the computation graph requires running some small operations. The state-of-the-art execution engines usually piggyback them to the framework's client thread that initiates the graph execution, but this hinders the scheduling process, which runs on the same thread. To solve this issue, *Graphi* maintains a light-weight single-threaded executor to take care of these operations. In order to avoid interference, one core is reserved exclusively for this executor as well.

6 Optimization Considerations

Since the design of *Graphi* allows us to implement many scheduling policies, we experimented several approaches. This section reports some of such optimization attempts and the insights we gained.

Different executor thread team sizes. A complex network such as LSTM consists of operations with varying sizes and different scalability. We performed a study to execute the computation graph with a sequential interpreter running on varying number of threads, and found that running time of operations scaled differently with the number of cores used. Based on this, we classified the operations into multiple classes (e.g. 3) according to how well they scale, and made the scheduler preferably assign an operation to an executor of corresponding thread team size.

This technique indeed reduced the total CPU time of all the threads. However, the makespan of the whole graph execution did not improve. This was because

different executor sizes could cause work straggling when some big operations are scheduled to run on the executors with a small team of threads. Therefore, the current *Graphi* uses symmetric executors with the same number of threads. Whether varying team sizes is useful on other models requires more investigations.

Dynamic number of executors. We considered varying the number of executors dynamically during of the course of graph execution. For example, we tried to use different numbers of executors for forward and backward computations during a model training. The rationale is that typically the number of parallel operations doubles during the backward pass.

We found that two issues prevented this optimization method from being effective. First, there is a limitation with OpenMP such that thread reuse could not be guaranteed if the thread team size changed dynamically. Our experiments showed that the overhead of context switches between different threads on the manycore CPU is significant, at about 10-30 ms. This is aligned with the numbers provided in [6]. Second, as shown later in Figure 6, after certain saturation points, increasing the parallelism only by a factor of 2 reduces the overall running time only slightly. The optimization to double the executors during the backward pass might not be worthwhile especially when the operations are large enough.

Data cache locality. Pinning threads of an executor to specific cores gives control over the execution location of the threads. Combined with the knowledge from the centralized scheduler, we could naturally incorporate L2 cache locality in the *Graphi* execution engine. Note that when one operation finishes, it may trigger another using its result as the input data. In this case, we made the system remember the current executor as the *preferred executor* for this triggered operation, which would then have the priority to run as the next operation on the preferred executor. Such cache affinity idea had been studied in the past [47, 48].

When analyzing the execution times of individual operations, we found that only element-wise operations improved by a modest margin, while matrix multiplications did not improve. Our hypothesis is that this is due to the blocking scheme of Intel MKL on the input/output matrices. As long as the threads of one executor does not totally reside in the shared L2 cache within a tile (consisting of two cores), data still have to traverse between different L2 caches, defeating the purpose of cache affinity scheduling. Further experiments where each thread team resided in one tile showed modest yet consistent improvement, confirming our hypothesis, but we did not pursue the idea because of this restrictive setting.

In contrary to locality, we find that writing through the results of an element-wise operation to memory with stream store[1] slightly improves the overall performance. Since it is likely that the results will not be reused by the same executor, there is no need to fetch the overwritten data into the cache to cause additional overheads. Therefore we adopt this technique in all our design.

7 Evaluation

We tested our implementation of *Graphi* with 4 representative deep learning models including LSTM, PhasedLSTM, PathNet and GoogleNet. Our evaluation seeks to answer the following key questions:

1. What's the overall performance of *Graphi* on the manycore CPU? How does it compare with the state-of-the-art deep learning framework like TensorFlow? (§7.2)
2. How much parallelism is needed for running different computation graphs on the manycore CPU, and what is the relationship between parallelism and performance? (§7.3)
3. How much benefit does our centralized scheduler have on a hardware resource contention free baseline design? (§7.4)

7.1 Experiment Setup

Environment In the evaluation, we ran *Graphi* with various experiment settings and compared its per-

[1] achieved through `#pragma vector nontemporal`

Size	Sequence	Neurons
Small	20	128
Medium	30	512
Large	40	1024

(a) LSTM/PhasedLSTM

Size	Image	Neurons
Small	32	16
Medium	48	32
Large	64	48

(b) PathNet

Size	Image	Width
Small	128	1
Medium	192	2
Large	256	4

(c) GoogleNet

Table 1: Parameters of the deep learning models in evaluation.

formance with TensorFlow (version 1.2.0), a popular state-of-the-art deep learning framework on Intel Xeon Phi processor 7250 (see Section 2). *Graphi* compiled operations of the computation graph via ICC (version 17.0.420170411), and links to Intel MKL 2017 for matrix multiplication and LIBXSMM (version 1.8.1) for convolution. TensorFlow is configured to run multiple operations in parallel on the manycore CPU using Intel MKL 2017 for matrix multiplication and convolution.

Deep learning models The experiments ran the training phases of four neural networks. The first one is LSTM [28], a popular recurrent neural network model with applications in modeling text [56, 57], speech [17, 23], and video [46, 63]. Both our implementation and the TensorFLow benchmark of LSTM are based on [65].

The second is PhasedLSTM [42], a recent variant of LSTM which suits for processing asynchronous sensory events that carry timing information. TensorFlow provides PhasedLSTM cell for benchmarking, and we implemented an identical counterpart for *Graphi*. Note that the customized optimizations for LSTM cannot be easily applied to PhasedLSTM even if these two networks only have slight difference. However, the optimizations with *Graphi* apply to both since it is neural network agnostic.

Table 1a summarizes the three network sizes of LTSM-like networks following the Tensorflow convention.

The third one we used is a convolutional neural network (CNN) called PathNet [20] invented by DeepMind. PathNet is designed to be trained on multiple tasks simultaneously, leading to many parallel modules in each layer. We implemented this neural network for both *Graphi* and TensorFlow using one 3×3 convolution, followed by rectified linear units and a 2×2 pooling in each module. We chose 3 sizes for evaluation based on the original study, with number of layers set to 3, active modules per layer set to 6, and the remaining parameters summarized in Table 1b.

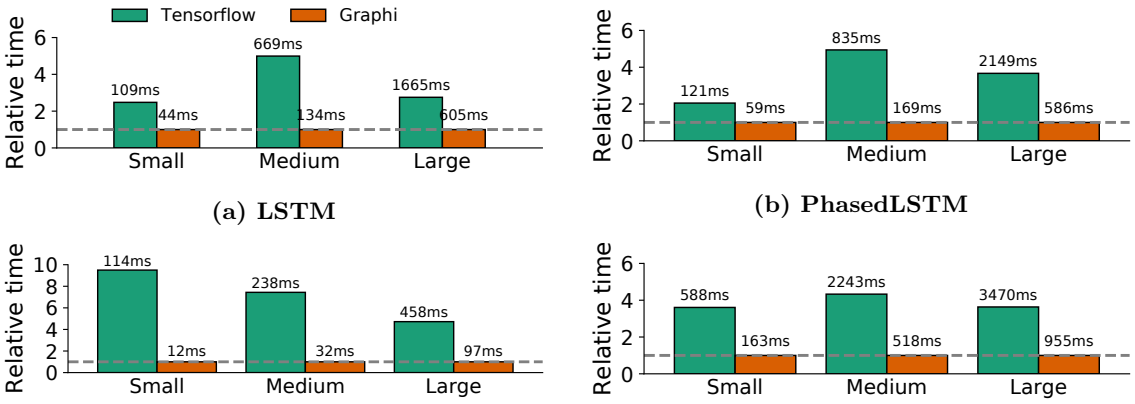
Lastly, we also evaluated on GoogleNet [58], a deep CNN model widely used in image classification. It does not have as many parallel operations compared to the previous 3 networks, so there is less room for optimization in *Graphi*, nevertheless, the "inception" modules in GoogleNet still consists of 2-3 parallel convolution/pooling operations, so sequential execution is suboptimal. We refer to the implementation provided in TensorFlow for evaluation as well, but vary the image size and multiply the number of output filters in each convolution by a constant factor (i.e. "width" of the network [64]) to obtain models of 3 different sizes (Table 1c).

For LSTM/PhasedLSTM/PathNet, the batch size is set to 64, and for GoogleNet, the batch size is 32, to maximally utilize the 16GB MCDRAM.

7.2 Overall Results

Figure 5 shows the overall results of batch training times of different models by both *Graphi* and TensorFlow. These are results of the best parallelization settings for both *Graphi* and TensorFlow. For clarity, we normalized the batch training time of different models.

The results show that *Graphi* achieves 2.1-9.5× speed-up compared with TensorFlow on the manycore CPU. For LSTM/PhasedLSTM, because both *Graphi* and TensorFlow relies on MKL for the time consuming matrix multiplication operations, the better results are largely attributed to *Graphi*'s execution engine.



(a) LSTM **(b) PhasedLSTM**

(c) PathNet **(d) GoogleNet**

Figure 5: Batch training time of TensorFlow and *Graphi* on the manycore CPU. y-axis shows the relative running time to *Graphi*, lower is better.

Since TensorFlow does not control thread placement, multiple threads often run on the same physical core, causing interference and unpredictable performance. Besides, TensorFlow uses Eigen for element-wise operations, which has its own thread pool, making the problem worse.

Moreover, Eigen divides all the element-wise operations into small chunks and manages them in a centralized job queue. This causes contention as well, and we think this helps to explain why *Graphi* performed best with the medium sized networks relative to the small/large: for the small networks, each operation is not divided into many chunks and the effect is not too damaging; and for the large networks, Eigen's job queue design is less of a bottleneck because each operation now takes longer to run.

For PathNet, *Graphi* achieved about 9.5× speedup on the large size, 7× on the medium, and 4× on the small. Aside from the difference of execution engines, we think this is also attributed to the building primitives, because LIBXSMM, the library *Graphi* uses for convolution, has been specially optimized for small convolutions compared with the Intel MKL convolution implementation.

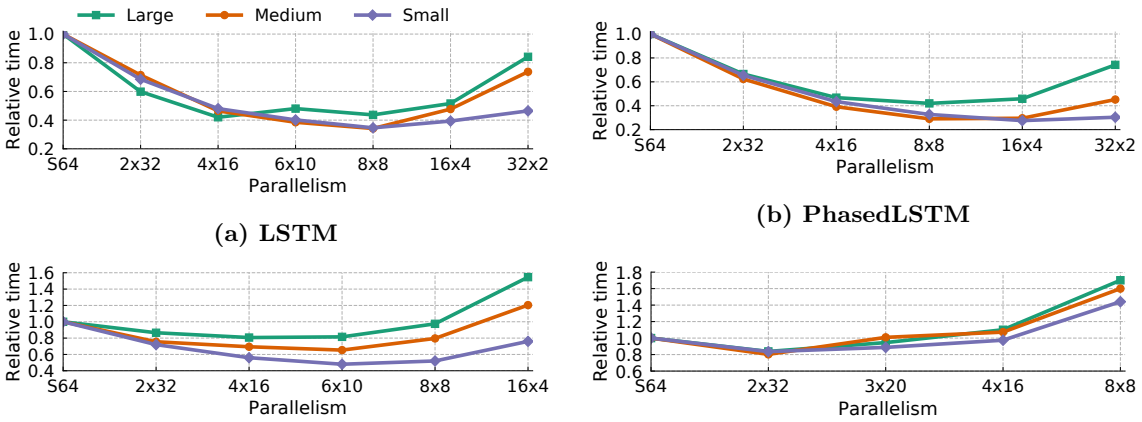
Graphi was about 3-4× faster on GoogleNet of all three different sizes. Although GoogleNet is a relatively simple network with less optimization opportunities, *Graphi* still benefits from the better parallel scheduling, in addition to the more performant primitives provided by LIBXSMM.

In the next two subsections, we try to decompose the speedup contribution made by our proposed schemes. We analyzed the effect of parallel execution (7.3) and intelligent scheduling (7.4), and the rest is attributed to the elimination of resource interference.

7.3 Varying number of executors

In this experiment we varied the number of executors in *Graphi* when running different neural networks, and observed how the performance changed. The Intel Xeon Phi processor 7250 has 68 cores, 2 of which was reserved for the scheduler and the light-weight executor, respectively as discussed in Section 5. We varied the number of executors from $k = 2, 4, 8, 16, 32$, and assigned $64/k$ cores to each executor. PathNet in our setting had 6 modules per layer, so in addition to the aforementioned settings, we added one setting with 6 executors, each using 10 cores. And for GoogleNet because it has 2-3 parallel operations, we also tried 3 executors each with 10 cores. For comparison, we also ran a sequential execution engine. Figure 6 shows the relative batch training times of *Graphi* compared with the sequential engine.

From the figure we can see that parallel executions of *Graphi* achieved significant speed-ups for all 4 mod-



(a) LSTM

(b) PhasedLSTM

(c) PathNet

(d) GoogleNet

Figure 6: Relative batch training time of *Graphi* under different parallelism settings compared with a sequential execution engine. $n \times k$ on x-axis means n executors, each using k cores. S64 refers to sequential execution engine using 64 cores.

els on Intel Xeon Phi processor 7250. For LSTM and PhasedLSTM, the highest improvements ranged from 2.3× to 3.1×, and for PathNet, they ranged from 1.2× to 2.1×. In general, the speed-ups for small networks are more pronounced, because the small operations in these networks lead to poor usage of the cores with a sequential execution engine. For GoogleNet with less parallelism opportunities, the speed up is smaller and around 1.2×, and the performance decreases rapidly when we have more than 2 executors.

Figure 6 also shows that different numbers of executors were needed to achieve the maximum performance for different networks. This highlights the importance of the profiling step (Section 4.2) of *Graphi* since the optimal number of parallel executors is related to the structure of the model. Specifically, in the four-layer LSTM/PhasedLSTM model, one cell from each layer can run in parallel, and there are 2-3 parallel operators in each cell, so the total number of parallelizable operations is around 8-12. PathNet/GoogleNet model has 6/2-3 modules in each layer that can run in parallel, respectively, corresponding exactly to the number of optimal number of executors needed.

When surpassing the optimal setting, the performance starts to decrease, with the large networks suffering most, because there are not enough parallel operations to utilize all the executors simultaneously, resulting in some executors being idle most of the time.

Parallelism	LSTM	PhasedLSTM	PathNet	GoogleNet
2×32	0.86	0.81	0.88	0.94
4×16	0.88	0.85	0.92	0.96
8×8	0.82	0.91	0.89	0.93
16×4	0.91	0.86	0.91	0.91
32×2	0.87	0.85	0.92	0.92

Table 2: Relative batch training time of *Graphi* vs. naive parallel scheduler on medium-sized networks.

Note that although we have enumerated through the configurations to obtain optimal number of executors needed in these experiments, in practice, it is also possible to infer some good settings through static analysis from the graph structure, just like what we have done above.

7.4 *Graphi* scheduler

Table 2 summarizes the relative training time of *Graphi* compared with the naive scheduling used in TensorFlow and MXNet on medium-sized networks

under various parallelism configurations. Note that in this comparison, we have eliminated all executor thread interference, so the performance difference only comes from the scheduler. For the sake of space, we only show the speed-up on medium sized networks, and the results on small/large networks are consistent.

Graphi achieved 8%-19% speed-up compared with the naive scheduling, in which all executors independently poll the centralized queue for operations. When the number of executors is large (e.g. in the manycore CPU), the heavy concurrent use of the centralized queue causes contention. In addition, part of improvement is also attributed to the critical-path first scheduling based on the operation-level profiling.

Specifically, there was greater speed-up on LSTM and PhasedLSTM because they have many more small operations, which results in severer contention on the global queue in the naive scheduling. Correspondingly, the improvement on GoogleNet is smaller because each operation is larger, resulting in less contention when polling the queue. Moreover, the more complex structures of the LSTM/PhasedLSTM computation graphs, the greater gain critical-path-first scheduling provides. In effect, the hand-optimized LSTM implementation by cuDNN [4] follows a diagonal parallel execution pattern for the LSTM cells on different layers and sequence locations. We visualized the operation execution trace and found the critical-path scheduler recovered the same pattern automatically (details omitted due to space limit), while the baseline scheduler failed to.

8 Related Works

Modern deep learning frameworks mostly express the computation of deep learning models in computation graphs. Caffe [30] and neon [43] use computation graphs with layers and explicitly labels the forward and backward paths, which limits the overall optimization opportunities. TensorFlow [2], MXNet [10], Theano [8] and Caffe2 [55] all express the computation in the pure computation graph. Such a design method leaves more room for optimization.

By default, the execution engines of the existing deep learning frameworks execute the computation graph in sequence according to its topological order. This is conceptually simple and feasible to neural networks with large operation since the within-operation parallelization is able to fully utilize the computation resources. For neural networks with smaller operations and more complex structures, TensorFlow and MXNet provide parallel execution engines for CPUs. However, their simple scheduling and thread interference often result in sub-optimal performance, especially on the manycore CPU. The *Graphi* execution engine proposed in this paper surmounts these two limitations to obtain high performance for various neural networks on the hardware.

The idea of expressing the computation as a directed acyclic graph (or in some context, dependence graph) dates back to the data flow computation [15, 16] in 1970s. Much of the later work focused on optimizing the execution performance in different scenarios especially in the compilers [13, 21, 36, 44, 45]. The work in this paper leverages the previous idea and applies it to a new application (deep learning computation graph) on manycore CPU architecture.

The online scheduling problem, i.e. how to dynamically schedule M jobs with dependencies to run on N workers has been studied for decades, including theoretical results such as achievable upper bounds of an online algorithm [19] as well as heuristic greedy solutions [12, 29]. The *Graphi* scheduler design is inspired mainly by the critical-path first scheduling algorithm [29] and the multiple queue skiplist scheduler [33].

How to eliminate thread interference has been an important performance optimization issue on multicore or manycore CPUs. Much of previous work focused on mapping different applications to different cores [14] or partitioning memory for multiple applications [40, 41]. Our work follows a similar rule-of-thumb to assure that different threads use disjoint resources so as to maximally reduce the resource contention in the *Graphi* execution engine.

The manycore CPU used in our experiments runs as an independent host. However, a manycore CPU can be viewed as an accelerator. Related work on optimizations for deep learning computations on the accelerators include GPU [11], FPGA [18, 26, 66, 67] and the first generation Intel Xeon Phi coprocessor

(Knights Corner) [68]. Their focus was on the optimizations of the critical deep learning primitives, e.g. convolution and matrix multiplication, which are the operations of a computation graph. Our work, in contrast, aims at optimizing the entire computation graph, and focuses more on inter-operation optimizations.

For specific neural networks like LSTM, customized optimizations such as operation fusion, network pruning and parallel execution of the computation graph (similar to *Graphi*) exist, including optimizations for GPUs [4] and FPGAs [26]. However, these are ad-hoc solutions specific to certain kinds of neural networks. Classical neural network like LSTM has many variants such as time-frequency LSTM [49], grid LSTM [31], LSTM with layer normalization [5] and dropout [52], phased LSTM [42], group LSTM [35], etc. *Graphi* is a generic execution engine, optimized for all variations without specializations.

9 Conclusions

This paper proposes *Graphi*, a generic and high-performance execution engine for running computation graphs of deep learning models on manycore CPUs. The focus of our work is on efficiently executing deep learning computation graphs, especially ones with small operations and complex structures. To achieve this goal, *Graphi* automatically finds the optimal parallel setting with profiling, minimizes the interference of parallel operations, and further improves execution efficiency with critical-path first scheduling.

Our experiments with four different neural networks showed that *Graphi* outperformed TensorFlow (optimized for CPU via MKL) by 2.1× to 9.5× on Intel Xeon Phi processor 7250. Further detailed analysis shows that proper parallelism without interference improves performance by 1.2× to 3.1×, and better scheduling results in 8% to 19% speed up and automatically recovers a handcrafted parallelization scheme for LSTM in cuDNN.

The work in this paper is a first step towards designing an efficient parallel engine for deep learning computation graph execution on the manycore CPU. Beyond the scope of this paper, we also have verified that *Graphi* achieves favarbale speedup on the latest multicore CPUs (Intel Xeon Platinum 8180, code named Skylake), demonstrating the generalizability of our framework. There are several future directions, including applying the ideas to GPUs and FPGAs as well as the distributed systems with multiple nodes, extending *Graphi* to handle dynamic computation graphs, and further optimizing *Graphi* for challenging memory hierarchies such as NUMA.

Acknowledgements The authors thank Alex Heinecke, Pradeep Dubey, Hans Pabst, Niranjan Hasabnis from Intel Corporation, Zhen Jia from Princeton University, Jie Gao from Stony Brook University, Yungang Bao from Institute of Computing Technology Chinese Academy of Sciences, and Xiaoqiang Zheng from Google for valuable discussions.

References

[1] 2017. Eigen: a C++ Linear Algebra Library. http://eigen.tuxfamily.org/. (2017). [Online; accessed 08-Aug-2017].

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera Computer System. *ACM SIGARCH Computer Architecture News* 18, 3b (1990), 1–6.

[4] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. *arXiv preprint arXiv:1604.01946* (2016).

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer Normalization. *arXiv preprint arXiv:1607.06450* (2016).

[6] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[7] Yoshua Bengio. 2009. Learning Aeep Architectures for AI. *Foundations and Trends® in Machine Learning* 2, 1 (2009), 1–127.

[8] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf.* 1–7.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2016. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).

[12] Edward G Coffman and Ronald L Graham. 1972. Optimal Scheduling for Two-Processor Systems. *Acta Informatica* 1, 3 (1972), 200–213.

[13] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[14] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-core Systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*. IEEE, 107–118.

[15] Jack B Dennis. 1980. Data Flow Supercomputers. *Computer* 11 (1980), 48–56.

[16] Jack B. Dennis and David P. Misunas. 1975. A Preliminary Architecture for a Basic Data-flow Processor. In *Proceedings of the 2Nd Annual Symposium on Computer Architecture (ISCA '75)*. ACM, New York, NY, USA, 126–132. https://doi.org/10.1145/642089.642111

[17] Yuchen Fan, Yao Qian, Feng-Long Xie, and Frank K Soong. 2014. TTS Synthesis with Bidirectional LSTM based Recurrent Neural Networks. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[18] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. 2009. CNP: An FPGA-based Processor for Convolutional Networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 32–37.

[19] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. 1993. Optimal Online Scheduling of Parallel Jobs with Dependencies. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*. ACM, 642–651.

[20] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. 2017. Pathnet: Evolution Channels Gradient Descent in Super Neural Networks. *arXiv preprint arXiv:1701.08734* (2017).

[21] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The Program Dependence Graph

and its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[22] MC Gilliland, Burton J Smith, and W Calvert. 1976. HEP-A Semaphore-Synchronized Multiprocessor with Central Control. (1976).

[23] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 6645–6649.

[24] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. 2005. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM Computer Communication Review* 35, 5 (2005), 41–54.

[25] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. 2016. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. *arXiv preprint arXiv:1606.04487* (2016).

[26] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA.. In *FPGA*. 75–84.

[27] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[29] Te C Hu. 1961. Parallel Sequencing and Assembly Line Problems. *Operations Research* 9, 6 (1961), 841–848.

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[31] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. 2016. Grid Long Short-Term Memory. In *4th International Conference on Learning Representations*.

[32] Richard M Karp and Rayamond E Miller. 1966. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411.

[33] Con Kolivas. 2016. MuQSS - The Multiple Queue Skiplist Scheduler. http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt. (2016). [Online; accessed 04-Aug-2017].

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[35] Oleksii Kuchaiev and Boris Ginsburg. 2017. Factorization Tricks for LSTM Networks. In *ICLR Workshop*.

[36] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. 1981. Dependence graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 207–218.

[37] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.

[38] Joseph YT Leung. 2004. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press.

[39] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 1. 3.

[40] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 367–376.

[41] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on*. IEEE, 374–385.

[42] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. 2016. Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences. In *Advances in Neural Information Processing Systems*. 3882–3890.

[43] Nervana. 2017. Intel® Nervana™ reference deep learning framework committed to best performance on all hardware. https://github.com/NervanaSystems/neon/. (2017). [Online; accessed 04-Aug-2017].

[44] Karl J Ottenstein and Linda M Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. *ACM Sigplan Notices* 19, 5 (1984), 177–184.

[45] David A Padua and Michael J Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29, 12 (1986), 1184–1201.

[46] Viorica Patraucean, Ankur Handa, and Roberto Cipolla. 2016. Spatio-temporal Video Autoencoder with Differentiable Memory. In *ICLR Workshop*.

[47] James Philbin, Jan Edler, Otto J Anshus, Craig C Douglas, and Kai Li. 1996. Thread scheduling for cache locality. In *ACM SIGOPS Operating Systems Review*, Vol. 30. ACM, 60–71.

[48] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.

[49] Tara N Sainath and Bo Li. 2016. Modeling Time-Frequency Patterns with LSTM vs. Convolutional Architectures for LVCSR Tasks.. In *INTERSPEECH*. 813–817.

[50] John Schulman. 2017. Computation Graph Toolkit. https://github.com/joschu/cgt/. (2017). [Online; accessed 04-Aug-2017].

[51] 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. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. ACM, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/1399504.1360617

[52] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. 2016. Recurrent Dropout Without Memory Loss. *arXiv preprint arXiv:1603.05118* (2016).

[53] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann Lecun. [n. d.]. Overfeat: Integrated recognition, localization and detection using convolutional networks. *http://arxiv.org/abs/1312.6229* ([n. d.]).

[54] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *Ieee micro* 36, 2 (2016), 34–46.

[55] Facebook Open Source. 2017. Caffe2. https://caffe2.ai/. (2017). [Online; accessed 08-Aug-2017].

[56] Ilya Sutskever, James Martens, and Geoffrey E Hinton. 2011. Generating Text with Recurrent Neural Networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 1017–1024.

[57] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Pnformation Processing Systems*. 3104–3112.

[58] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[59] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm @Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 147–156.

[60] András Vajda. 2011. *Programming Many-core Chips*. Springer Science & Business Media.

[61] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 167–188.

[62] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[63] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. 2015. Beyond Short Snippets: Deep networks for Video Classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4694–4702.

[64] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. *CoRR* abs/1605.07146 (2016). http://arxiv.org/abs/1605.07146

[65] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent Neural Network Regularization. *arXiv preprint arXiv:1409.2329* (2014).

[66] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Metworks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[67] Chi Zhang and Viktor K Prasanna. 2017. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System.. In *FPGA*. 35–44.

[68] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. 2016. ZNN–A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 801–811.