A Survey of Techniques for Optimizing Deep Learning on GPUs

Sparsh Mittal and Shraiysh Vaishay
Department of Computer Science and Engineering, IIT Hyderabad, India.
E-mail:{sparsh,cs17btech11050}@iith.ac.in.

Abstract

The rise of deep-learning (DL) has been fuelled by the improvements in accelerators. Due to its unique features, the GPU continues to remain the most widely used accelerator for DL applications. In this paper, we present a survey of architecture and system-level techniques for optimizing DL applications on GPUs. We review techniques for both inference and training and for both single GPU and distributed system with multiple GPUs. We bring out the similarities and differences of different works and highlight their key attributes. This survey will be useful for both novice and experts in the field of machine learning, processor architecture and high-performance computing.

Index Terms

Review, GPU, hardware architecture for deep learning, accelerator, distributed training, parameter server, allreduce, pruning, tiling.

1 Introduction

The rise of deep-learning has been fuelled by the improvements in accelerators. Even though many FPGA/ASIC[1]-based custom-accelerators have been recently introduced, GPU continues to remain the most widely used accelerator for DL training/testing, for several reasons. High-level programming languages such as CUDA make GPU easier to program than FPGA, and the programmable architecture of GPU makes it useful for a wider range of DL applications than ASIC. High performance of GPU, continued improvements in its architecture and software-stack, its availability in cloud/data-centers and a large user-base make GPU a universal accelerator for DL applications [1].

"Accelerating DL models" is, however, akin to chasing a moving target. As DL models are becoming more pervasive and accurate, their compute and memory requirements are growing tremendously and are likely to outpace the improvements in GPU resources and performance. For example, training DNNs takes a huge amount of time, e.g., 100-epoch training of ResNet-50 on ImageNet dataset on one M40 GPU requires 14 days [2]. To reduce the training time, researchers have used clusters with hundreds of GPUs; however, achieving high resource utilization efficiency and reducing communication overhead is vital

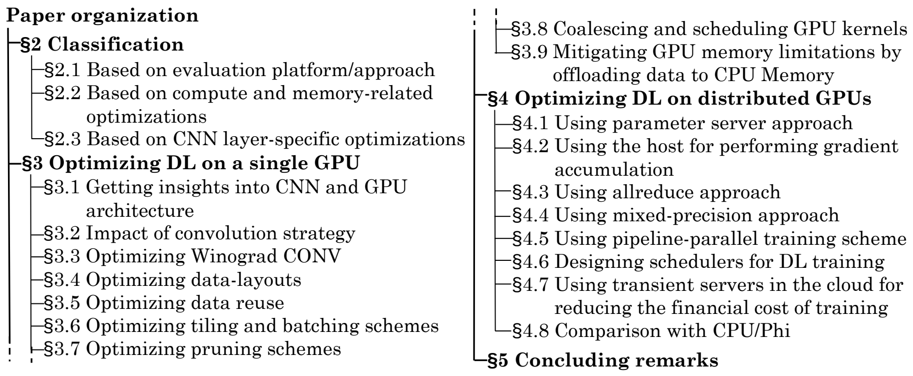
• *This work was supported in part by "Semiconductor Research Corporation" (SRC).*

1. We use the following acronyms frequently in this paper: application-specific integrated circuit (ASIC), backward/forward propagation (FWP/BWP), bandwidth (BW), batch normalization (BN), compressed sparse row (CSR), convolution (CONV), convolution/deep/recurrent neural network (CNN/DNN/RNN), cooperative thread arrays (CTAs), deep learning (DL), direct memory access (DMA), fast Fourier transform (FFT), feature extraction (FE), feature map (fmap), field programmable gate array (FPGA), fully-connected (FC), generalized matrix multiplication (GEMM), global memory (GlM), instruction/memory/thread-level parallelism (ILP/MLP/TLP), least/most significant bit (LSB/MSB), local response normalization (LRN), matrix multiplication (MM), parameter server (PS), peripheral component interconnect express (PCIe), register file (RF), shared memory (ShM), single instruction multiple data (SIMD), streaming multiprocessor (SM)

for achieving proportionate gains in performance. Similarly, during inference, meeting the latency targets while achieving high data-reuse and throughput is a major challenge.

Evidently, tackling these challenges will require optimizations at multiple levels such as microarchitecture, programming-language, system, and cluster-level. Further, due to the unique architecture of GPU [3], a straightforward implementation of even well-known optimization techniques such as pruning, batching, tiling, etc. is unlikely to provide high performance [4, 5]. Hence, an adaptation of these techniques to GPU architecture is required for reaping their full potential. Similarly, to leverage large-scale GPU-based clusters for DNN training, the challenges of distributed computing need to be mitigated. Many recent research projects aim at addressing these challenges.

Contributions: In this paper, we present a survey of techniques for optimizing deep-learning applications on GPUs. Figure 1 provides an overview of this paper. Section 2 classifies the research projects on several key metrics. Section 3 reviews techniques for optimizing CNNs on a single GPU. Section 4 summarizes works that optimize training on a distributed system with multiple GPUs. These sections are further divided into several parts, and although we summarize a technique at only one place, most techniques span across categories. In these sections, we also provide the background on relevant concepts.



Paper organization
- **§2 Classification**
  - §2.1 Based on evaluation platform/approach
  - §2.2 Based on compute and memory-related optimizations
  - §2.3 Based on CNN layer-specific optimizations
- **§3 Optimizing DL on a single GPU**
  - §3.1 Getting insights into CNN and GPU architecture
  - §3.2 Impact of convolution strategy
  - §3.3 Optimizing Winograd CONV
  - §3.4 Optimizing data-layouts
  - §3.5 Optimizing data reuse
  - §3.6 Optimizing tiling and batching schemes
  - §3.7 Optimizing pruning schemes
  - §3.8 Coalescing and scheduling GPU kernels
  - §3.9 Mitigating GPU memory limitations by offloading data to CPU Memory
- **§4 Optimizing DL on distributed GPUs**
  - §4.1 Using parameter server approach
  - §4.2 Using the host for performing gradient accumulation
  - §4.3 Using allreduce approach
  - §4.4 Using mixed-precision approach
  - §4.5 Using pipeline-parallel training scheme
  - §4.6 Designing schedulers for DL training
  - §4.7 Using transient servers in the cloud for reducing the financial cost of training
  - §4.8 Comparison with CPU/Phi
- **§5 Concluding remarks**

Fig. 1. Organization of the paper

Scope: For the sake of a focused presentation, we limit the scope of this paper as follows. We review works that optimize GPU architecture to match the characteristics of DL applications and the works that adapt DL applications to get maximum performance from GPU. In other words, we include the works that co-optimize GPU architecture and DL applications. We do not include the works that focus on GPU-optimizations for conventional applications, or hardware-unaware optimizations to DL applications. We include the works that perform experiments using desktop/server-scale GPUs and not mobile GPUs [6].

2 Classification

2.1 Based on evaluation platform/approach

Table 1 classifies the research works based on several categories. It first shows the neural network models and the DL phases (inference or training) which have been optimized. Then, it summarizes the DL framework and the GPU model used by different works. Finally, it highlights the works that compare GPU with other platforms and those that perform "CPU-GPU heterogeneous computing" to bring the best of both CPU and GPU together.

2.2 Based on compute and memory-related optimizations

Unlike CPUs, GPUs do not have large-size caches. Also, each GPU core is simpler and runs at a lower frequency. Hence, their single-thread performance is much lower than that of CPUs. Since GPUs are throughput-oriented processors, achieving large performance on GPUs requires exploiting its massive-multithreaded architecture and complex memory hierarchy [64, 65]. Table 2 shows the compute-related bottlenecks in different works along with the optimizations used by them. For example, pipelining overlaps computations with data-communication to hide communication latency. In GPUs, all threads

TABLE 1
A classification based on context and evaluation

Category	References
NN architecture	
RNN	[7, 8]
CNN	nearly all others
Machine learning phase evaluated/optimized	
Inference	[5, 8–22]
Training	[4, 5, 7, 9–12, 16, 19, 21, 23–51, 51–58]
Framework/library used for evaluation or comparison	
cuDNN	[4, 8, 9, 11, 17–20, 23, 24, 27, 31–34, 36, 39, 42, 44, 54, 59]
cuBLAS	[14, 18, 27, 54, 59]
Caffe	[7, 11, 13, 14, 18, 19, 21–23, 25, 28, 29, 40, 41, 47, 49, 52, 55, 57, 58]
TensorFlow	[8, 19, 21, 28, 30, 35, 38, 41, 43, 51, 53]
PyTorch	[28, 37, 38]
MPI	[37, 43, 56]
NCCL	[37, 43]
Others	MAGMA [5, 12, 13, 60], TensorRT [8], NERVANA [18], cuSPARSE [14], clBLAS [33], MXNet [21]
GPU make	
AMD	Radeon Fury X [29], FirePro W7100 [29], W8000 [33]
NVIDIA	nearly all others
GPU model used	
Fermi	GTX 480 [10, 16], M2050 [52]
Maxwell	GTX Titan X [4, 10, 17, 18, 24, 44, 52], GTX 750Ti [41], GTX 970m [18], GTX 980 [9, 20], M40 [35, 46]
Kepler	GTX 680 [16], K20 [16, 48], K20C [7, 18, 49, 52], K20m [59], K40 [9, 25, 27–29], K40c [23], K40m [12, 22, 26, 36, 45], K80 [19, 25, 30, 41, 46, 47, 53], GTX Titan Black [11]
Pascal	Titan X (Pascal) [13, 31], GTX 1080 [27], GTX 1080Ti [14, 34], P40 [38, 43], P100 [14, 25, 30–32, 37–41, 50, 55, 58, 60]
Volta	Titan V [8], V100 [5, 21, 30, 31, 37, 40–42, 44, 56, 61, 62]
Comparative and collaborative evaluation	
Comparison of GPU architectures	[21, 27]
Comparison of GPU with	CPU [17, 19, 22, 25, 60, 63], Xeon Phi [33, 46]
CPU-GPU heterogeneous computing	[17, 22, 39]

of a warp execute the same instruction. If different threads of a warp take different paths of an `if` condition, then both true and false paths are serially executed such that only one set of the threads remain active at a time. This branch divergence leads to performance loss, and several works propose solutions for this issue, as shown in Table 2. Kernel fusion allows doing all the computations on a piece of data before sending it to off-chip memory. This increases the opportunities for data-reuse. Some other works use loop unrolling, perform scheduling and resource-management for GPU kernels, and optimize costly mathematical operations.

Table 3 discusses memory-related performance-issues and optimization schemes. In GPUs, GlM access efficiency is high if the accesses are coalesced. ShM access efficiency is high when there are no bank conflicts. An exception to this is when all threads of a warp access the same address. In this case, a broadcast operation is performed and no bank-conflict happens. The access efficiency of constant memory is high if all threads of a warp access the same address.

Table 3 also shows the strategies for reducing the memory capacity requirement and memory accesses of a CNN. Tiling reduces the effective memory footprint so that it can fit in the on-chip memory itself. Accesses to data without temporal locality can be bypassed from the cache, whereas techniques for increasing the data-reuse can be employed to serve these requests from on-chip memory.

2.3 Based on CNN layer-specific optimizations

Different layers of a CNN have different architectural properties [27, 67]. For example, while CONV layers are compute-intensive, FC layers are memory-intensive. Remaining layers have other inefficiencies which

TABLE 2
A classification of compute-related challenges and optimizations

Compute-related bottlenecks	Synchronization barrier between kernels [8], data-dependencies prohibiting parallelization [11], degree of parallelization being low due to small batch size [11]
Pipelining	[7, 9, 15, 17, 24, 26, 28, 37, 37, 39, 40, 43–47, 49, 56]
Padding	[16, 33]
Loop unrolling	[10, 16]
Creating scope for increasing batch size	by performing gradient aggregation on the host to reduce memory consumption on GPU [39], by transferring intermediate data from GPU memory to CPU memory [28], by using low-precision tensors [37], by dividing the image into patches [32], by using "layer-wise adaptive rate scaling" (LARS) scheme [43]
Avoiding branch-divergence	Avoiding condition-check by performing variable increment of program-counter [10], Grouping threads that perform multiplication on the same input fmap into the same warp so that all threads in a warp can skip zero-multiplication [10], avoiding irregular computations by computing the multiplications separately for each non-zero weight and then adding it [14]
Avoiding thread-idling	by using same thread-block size for different tile sizes [5]
Optimizing multiplications/divisions	Skipping multiplications [10], performing SIMD multiplication (which are more efficient than scalar multiplications) [4], achieving division operations using multiply and shift-operations [9]
Kernel-level optimizations	kernel-fusion [11, 15, 27], prioritizing/delaying kernel execution [15], adjusting GPU resources allocated to a kernel [15], compilers for deep-learning [66]
Optimization algorithms/heuristics	Dynamic programming [41, 44], integer linear programming [41], random forest [5], greedy algorithm [24]

TABLE 3
A classification of memory-related bottlenecks and optimizations

Memory-related bottleneck reasons	Uncoalesced accesses in GlM [11, 20], redundant accesses to GlM [11], bank-conflicts in ShM [11]
Tiling (blocking)	[5, 9, 11, 12, 16, 20, 31, 33, 36, 59, 60]
Use of on-chip memory (ShM) for	performing reduction operation [11], generating lowered matrix [9]
Use of constant memory for storing	input data [14], small filters [12, 33]
Use of texture memory	[27]
Avoiding ShM bank-conflicts	by using broadcast scheme when all threads access the same address [12], by using `float2` datatype [11, 12]
Use of different memories	Storing the data in constant memory in case of single input channel and in GlM in case of multiple input channels [12]
Avoiding fetching	zero-padding data [59], zero weights [10]
L1 cache bypassing	[27]
Increasing data-reuse	by reordering instructions or computations [10, 59], by increasing the number of elements computed by each thread [11, 12]
Pruning	[4, 13, 14]
Prefetching	[12, 34]
Reducing memory requirement	by image-splitting [32], reducing batch size [26, 41]
Avoiding the overhead of `cudaMalloc/cudaFree`	by allocating a large memory pool only once and then acquiring/releasing memory from this pool [24, 28]

can be mitigated by intelligent techniques. Table 4 shows the works that propose specific optimizations to different layers of a CNN.

3 Optimizing DL on a single GPU

In this section, we review architectural and system-level techniques for optimizing CNNs on a single GPU. We first review works that explore CNN execution on GPUs (Section 3.1) and study the characteristics of different CONV strategies (Section 3.2). Then, we review approaches for optimizing Winograd CONV (Section 3.3), data-layouts in memory (Section 3.4) and schemes for exploiting data-reuse opportunities (Section 3.5). Further, we discuss techniques for adapting tiling and batching schemes (Section 3.6) and pruning schemes to GPU architecture (Section 3.7). We also discuss techniques for improving GPU resource utilization by scheduling and coalescing GPU kernels (Section 3.8). Finally, we review techniques that utilize CPU memory, in addition to GPU memory, for training CNNs (Section 3.9).

TABLE 4
A classification based on CNN layer-specific optimizations

Category	References
Offloading data of only CONV layers and not FC layers	[24, 28]
Ring-based allreduce for CONV layers and hierarchical allreduce for FC layers	[43]
Node pruning on CONV layers and SIMD-aware weight pruning on FC layers	[4]
Communicating parameters of CONV layers by PS and parameters of FC layers by either PS or "sufficient-factor broadcasting"	[49]
Executing FC layers in PS and remaining layers in worker GPUs	[51]
Use of different batch-size for different layers	[41, 52]
CNN layer that is optimized	Pooling layer [11, 17], softmax layer [11], CONV/FC (nearly all others)

3.1 Getting insights into CNN and GPU architecture

Dong et al. [27] execute AlexNet on two different GPUs: a server GPU (Tesla K40 with Kepler architecture) and a desktop GPU (GTX1080 with Pascal architecture). **Performance behavior:** They perform experiments with batch sizes of 16, 64 and 128. They observe that GTX1080 provides higher performance than K40 due to its higher clock frequency and the larger number of SMs. The speedup is much higher for CONV, FC and pooling layers than for softmax and activation layers. They also analyze the reasons for the stall in each layer and find that CONV layers are compute-bound, whereas activation layers are memory-bound. Both GPUs show increasing throughput with increasing batch size and thus, they show good scalability.

Compute and memory characteristics: On K40, some layers show stall due to memory-bottleneck, but this stall-reason is absent on GTX1080, which shows that GTX1080 has an efficient data-path for enabling data-movement between core and memory. On K40, LRN, pooling and softmax layers are both compute and memory bound, but on GTX1080, they are memory-bound since GTX1080 has achieved higher improvement in compute performance than memory performance. For the same layer, ALU utilization efficiency is higher on GTX1080 than on K40.

Locality characteristics: They further use a batch size of 128 and find that both CONV and FC layers show spatial locality and hence, show a high hit rate of texture cache. By comparison, L1 cache cannot hold the working set due to its small size. Further, due to the higher clock frequency of GTX1080, ShM and L2 cache provide higher throughput and can handle a larger number of memory transactions. This also reduces the number of accesses to DRAM.

Since CONV and FC layers show high locality, they benefit significantly from texture cache and ShM. By comparison, other layers have low data reuse, and hence, they do not fully utilize the on-chip memories. The BW of ShM is nearly four times higher than that of texture cache since their access latencies are 38 cycles and ~440 cycles, respectively. Hence, texture cache BW becomes a bottleneck in CONV layers. Since the regular access pattern of activation layers allows memory-coalescing, the GlM throughput is higher for activation layers than that of remaining layers.

Optimization opportunities: They suggest that instead of increasing DRAM bandwidth, increasing the bandwidth of texture cache can provide larger improvement in performance. Further, since many layers do not use the L1 cache, selectively applying L1 cache bypassing for those layers can improve performance. Also, by fusing non-activation layers with CONV layers, the number of memory transactions can be reduced with only a small increase in the number of computations in CONV layers.

Lym et al. [31] propose an analytical model of performance and memory traffic of GPU while it runs CNNs. Their model handles `im2col` (image-to-column), which is most-widely used algorithm for CONV layers on GPUs. `im2col` performs replication and reordering of input matrices for exposing the parallelism. For a given CONV layer, `im2col` creates specific memory access and locality patterns and these are accounted in their model.

Their model separately estimates traffic in L1/L2 cache, ShM and GlM (DRAM) according to the data reuse granularity based on the GEMM kernel tiling factors. Then, based on memory traffic, memory-bandwidths and computation-throughput, overall performance is estimated. Evaluation of their model for AlexNet, ResNet, GoogleNet and VGG on three GPUs (V100, P100 and Titan XP) confirms its accuracy for different layers. Their model is useful for identifying the resource that bottlenecks the performance. For example, for most layers, throughput is the bottleneck since `im2col` GEMM already has high data reuse.

The layers which do not have a sufficient number of CTAs for hiding the load latency are constrained by DRAM latency and those having many CTAs for interleaving are constrained by DRAM bandwidth. Based on the bottleneck analysis, they also propose architectural changes to improve CNN performance.

3.2 Impact of convolution strategy

There are four strategies for performing CONV:

1) Direct CONV: This approach works by sliding the filter over the input feature and computing sum of dot-product between their elements.
2) GEMM-based: This approach transforms a 2D CONV into a GEMM operation. It flattens the 2D filter into a 1D array and fills the input features in a matrix in a manner that one output feature is generated on performing dot-product of the 1D array with every column of the matrix. This strategy is also termed as "lowering" and is shown in Figure 2. Lowering allows performing CONV using GEMM libraries which have been heavily optimized. The limitation of lowering is that it wastes memory BW and increases memory requirement due to duplication of input features. This overhead is negligible for dense MM but is unacceptable for sparse CONV due to its poor arithmetic intensity [14].

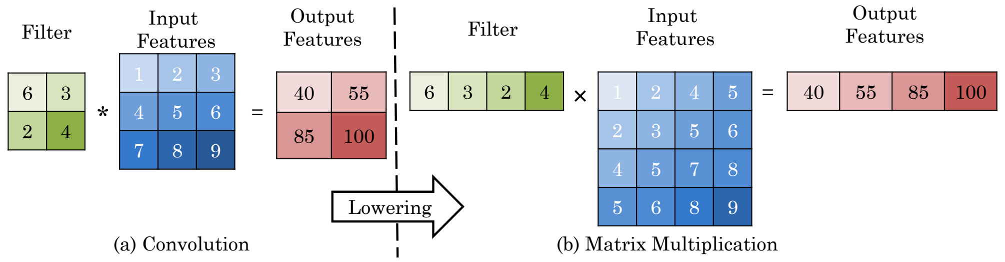


Fig. 2. Using lowering approach to convert (a) Conventional CONV using sliding window approach (b) GEMM-based CONV

3) FFT-based: CONV in time-domain is equivalent to multiplication in the frequency-domain, and this is the key idea behind FFT-based CONV. Here, we multiply FFT of input feature with FFT of the filter and then transform the result into time-domain using inverse FFT. 2D CONV is computed using 2D FFT, which itself is computed as 1D FFT of every row followed by 1D FFT of every column.
4) Winograd: 1D CONV with m outputs and r-tap filter is shown as $F(m,r)$. Input dimension is $m+r-1$. Let the filter and input be shown as g_i and d_i, respectively. Then, the Winograd CONV output is computed as follows: $Y = A^T[U \bigodot V] = A^T[(Gg) \bigodot (B^T d)]$
   Here, the matrices A, B and G are constant for a given value of m and r. Winograd CONV is performed in four phases:
   - Phase 0 (offline): Filter weights are transformed ($U = Gg$).
   - Phase 1: Input data is transformed ($V = B^T d$).
   - Phase 2: "Element-wise multiplication" is performed ($M = U \bigodot V$).
   - Phase 3: "Inverse transformation" is performed ($Y = A^T M$).

   2D CONV $F(m \times n, r \times s)$ is performed by nesting the 1D CONVs $F(m,r)$ and $F(n,s)$ along each dimension [10, 68]. Compared to direct CONV, Winograd CONV reduces the number of multiplications at the cost of extra addition operations.

Table 5 highlights the CONV strategy used by different works. Some works dynamically change the CONV strategy to exercise a tradeoff between performance and memory consumption, and these works are also shown in Table 5.

Chetlur et al. [9] discuss the implicit-GEMM based implementation of CONV in cuDNN. In their approach, fixed-size tiles of input matrices are loaded into on-chip memory and are utilized for computing a tile of the output matrix. Operation on one set of input tiles is overlapped with the transfer of the next set of tiles to hide the data-transfer latency. In GEMM-based approach, generating the lowered matrix in off-chip incurs large overhead. To avoid this overhead, their technique lazily generates the lowered matrix in on-chip memory. Compared to GEMM-based approach, their implementation additionally requires performing indexing for loading appropriate tiles of input matrix in on-chip memory. Indexing

TABLE 5
Strategies used for performing CONV

Category	References
Direct	[11, 12, 14, 16, 17]
Unrolling/matrix-multiplication	[5, 11, 61, 69]
FFT	[11, 16, 17, 36, 61]
Winograd	[10, 61]
Dynamically choosing optimal CONV strategy	[11, 24, 26, 28, 41]

computations require integer divide/modulo operations by "launch-time constant" values. To reduce their overhead, they implement these computations using integer multiply and shift-operations.

Li et al. [23] compare performance and optimization opportunities of various libraries/frameworks for optimizing CNNs on GPUs. They evaluate seven frameworks: Caffe, cuda-convnet2, cuDNN, fbfft [36], Theano-CorrMM, Theano-fft and Torch-cunn on four CNNs: AlexNet, GoogleNet, VGG, and Overfeat. They profile the performance of CONV layers since they take the largest fraction of execution time. They analyze a single CONV layer with seven frameworks for five different parameters, namely kernel size, stride, image dimension, mini-batch size and filter number. Their observations are as follows:

Functionality: cuda-convnet2 works well only when the mini-batch size is a multiple of 128. It also requires kernels and input images to be square. GEMM-based CONV works with all the tested parameters. FFT-based CONV (fbfft and Theano-fft) works with all the tested parameters except that stride size above 1 is not supported.

Performance: On the performance metric, fbfft is the best whereas cuDNN is placed second. Reason for the high performance of fbfft is that FFT-based CONV is more efficient than direct and GEMM-based CONV. In FFT-based CONV, the kernel must be padded to be of the same size as the inputs, which is costly for small kernel size. Hence, for small kernel size (e.g., less than 7), cuDNN is faster than fbfft. Theano-fft provides the least performance. Among the frameworks that perform GEMM-based CONV, cuDNN provides the highest performance, except that for filter-number above 160, Theano-CorrMM provides slightly better performance than cuDNN.

Functions taking the largest amount of time: Out of the frameworks that perform FFT-based CONV, in Theano-fft, majority of the time is spent in preparing and transferring data between CPU and GPU, whereas in fbfft, majority of time is spent in GEMM, FFT, inverse-FFT and data-transpose operations. In GEMM-based CONV, GEMM operation and unrolling take the largest and second largest amount of time, respectively.

Memory consumption: In terms of increasing memory consumption, CONV approaches are: direct, GEMM-based and FFT-based. Among GEMM-based frameworks, Torch-cunn takes the least memory, whereas for large kernels, cuDNN takes the least amount of memory. Among FFT-based frameworks, Theano-fft takes less memory than fbfft, but both of them take an unusually high amount of memory for certain configurations. Thus, cuda-convnet2 is best for platforms with small memory, whereas cuDNN balances multiple factors such as performance, memory consumption, and versatility.

GPU resource utilization: cuda-convnet2 has low occupancy on GPU, since each thread in cuda-convnet2 uses a high number of registers and hence, due to register-usage limit, only few threads can run at a time. Due to this, memory latency cannot be effectively hidden. By contrast, the poor performance of Theano-fft comes from issuing many accesses to GlM since it does not fully utilize ShM and registers. Theano-fft also has poor warp-execution efficiency due to thread-divergence arising from control-flow instructions.

Theano-fft, Theano-CorrMM, Torch-cunn, and Caffe show poor GlM load efficiency due to lack of memory-access coalescing. The ShM efficiency is lowest for Theano-fft due to the large number of bank-conflicts. Among GEMM-implementations, cuDNN provides highest ShM efficiency. Since the GEMM operations in Theano-CorrMM, Torch-cunn, and Caffe are performed by cuBLAS, which is optimized for ShM, these frameworks also show high ShM efficiency. Finally, the overhead of data-transfer between CPU and GPU is nearly-zero in fbfft, Caffe and cuDNN, but is slightly higher in other frameworks.

3.3 Optimizing Winograd CONV

Park et al. [10] present two techniques for improving the performance of Winograd CONV on GPU. They note that a large fraction of weights in a CNN are zero, especially after applying pruning technique. If any element of U is zero, their first technique avoids loading it and also skips the multiplication between elements of U and V. The corresponding code is shown in Figure 3(a). Due to the lockstep execution of threads in a warp, the above approach reduces latency only if all the threads of a warp operate on zero U values. Their technique groups threads performing multiplication on the same input fmap into the same warp, since their $U[i]$ value is same and hence, zero-result multiplication can be skipped in all the threads of a warp.

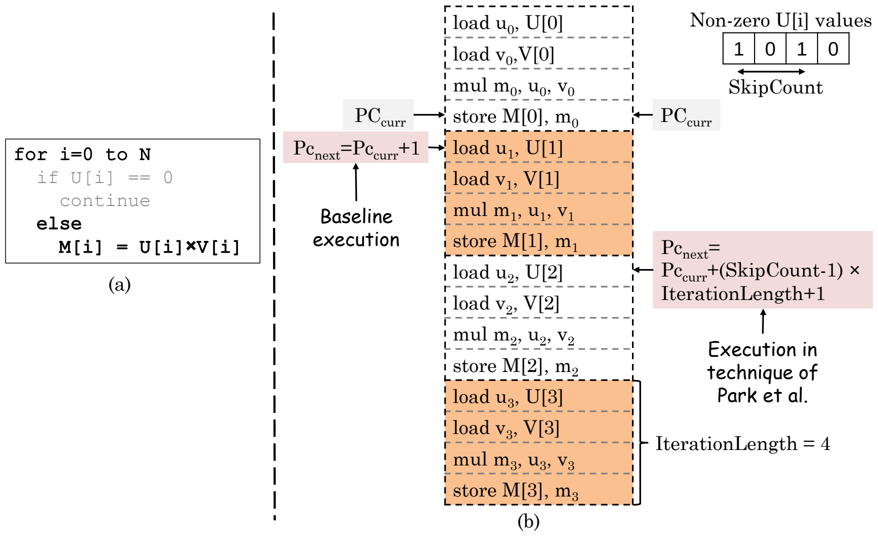
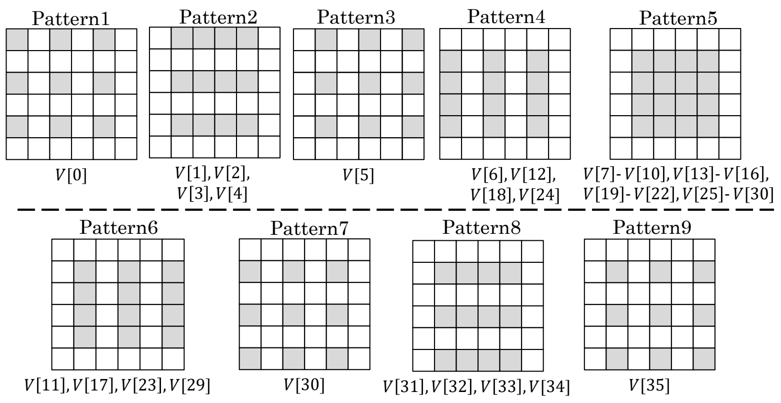


Fig. 3. (a) Code of Phase 2 of "Winograd CONV" after skipping multiplication with zero (N= number of elements in matrix M) (b) Incrementing PC based on SkipCount and IterationLength to avoid the need of checking conditions

They further note that since each iteration has only a few instructions, the relative overhead of condition-checking instructions becomes large. Hence, despite avoiding multiplications, above technique degrades performance, due to the overhead of instructions added for performing condition check. For mitigating this overhead, they add a bit-vector in every GPU core, such that i^{th}-bit of the vector is 0 if U[i] is 0, and vice versa. In normal case, in each cycle, PC (program-counter) is incremented by one, as shown in the left side of Figure 3(b). In their technique, after every iteration, the bit-vector is scanned to find the next non-zero bit. Then, the PC is incremented in a way that it directly jumps to read the instruction of the corresponding iteration. This is shown in the right side of Figure 3(b). Here, SkipCount is obtained by subtracting the present index from the next non-zero index and IterationLength shows the number of instructions in an iteration. Thus, without using condition-check instructions, their technique executes only those iterations whose $U[i]$ is non-zero.

The above technique is effective for small tile size such as 2x2 since in this case, phase 2 contributes to the largest portion of execution time. However, for large tiles such as 6x6, phases 1 and 3 are the largest contributor to execution time. Thus, for large tiles, the overhead of addition operations becomes large. For such case, they propose a second technique which increases the reuse of operands of add operation during their residency in RF. This reduces accesses to the on-chip cache. Since GPU has a huge number of threads, the per-thread RF capacity is small. Hence, maximally reusing the operands present in RF is important.

They propose the following optimization for phase 1, and the optimization proposed for phase 3 is similar to it. The input access pattern in computation of matrix V is shown in Figure 4. When $F(4x4, 3x3)$ operates on a 6x6 input tile (i.e., calculating 4x4 outputs with a 3x3 kernel), there are nine distinct access patterns in the computation of V[0] to V[35]. As an example, computing V[11] requires accessing 12 elements shown with the shaded square in Figure 4 and marked as Pattern6. They make two observations and propose corresponding optimizations based on them, which are shown in Table 6.



Pattern1
Pattern2
Pattern3
Pattern4
Pattern5
V[0]
V[1], V[2], V[3], V[4]
V[5]
V[6], V[12], V[18], V[24]
V[7]-V[10], V[13]-V[16], V[19]-V[22], V[25]-V[30]
Pattern6
Pattern7
Pattern8
Pattern9
V[11], V[17], V[23], V[29]
V[30]
V[31], V[32], V[33], V[34]
V[35]

Fig. 4. Data access scheme in phase 1 of Winograd CONV [10]

TABLE 6
Optimization opportunities exploited by Park et al. [10]

Observations	Corresponding optimizations
1. Input data elements required for computing some elements of V are same. For example, V[7]-V[10], V[13]-V[16], V[19]-V[22] and V[25]-V[30] are computed from the same parts of input data (Pattern5).	Additions are reordered such that elements of V requiring same inputs are calculated in close proximity of time. For example, after computing $V[11]$, $V[17]$ is computed instead of $V[12]$, since both $V[11]$ and $V[17]$ require same inputs (Pattern6).
2. Many patterns need common portions of input data and the amount of overlap is different in various pairs. As an example, Pattern4 and Pattern6 have no overlap whereas there are 8 common elements between Pattern4 and Pattern5.	They place those code sections at contiguous locations which have many input portions in common. Also, if the total inputs required for different access patterns is less than the number of registers per thread, all inputs required for those patterns are loaded together. For instance, 18 distinct input elements are required for Pattern7 to Pattern9, which is less than 32 register-per-thread limit, even on allocating 6 registers to V[30] to V[35]. Hence, these 18 elements are loaded together.

After applying these optimizations, the final order of computation is (Pattern1 and Pattern2 and Pattern3) $\rightarrow$ Pattern6 $\rightarrow$ Pattern5 $\rightarrow$ Pattern4 $\rightarrow$ (Pattern7 and Pattern8 and Pattern9). Overall, their techniques improve CONV performance significantly and the two techniques provide improvement for different tile sizes.

3.4 Optimizing data-layouts

Let F_h and F_w show the dimensions of CONV filter, C_o shows number of filters or output fmaps, C_i shows number of input fmaps, H_i and W_i show the height and width of a fmap and N_i shows the batch size. Then, Equation 1 shows the computation performed in CONV.

$$Out_{co}[Ni][Co][Hi][Wi] = \sum_{Ci=0}^{C} \sum_{f_{h=0}}^{F_H} \sum_{f_w=0}^{F_W} In_{co}[Ni][Ci][Hi+f_h][Wi+f_w] * filter[Co][Ci][f_h][f_w] \tag{1}$$

The 4D array shown in Equation 1 can be stored in different layouts. Assume that the following symbols are used: image height/width (H/W), number of images (N) and feature maps (C). Evidently, Equation 1 assumes NCHW layout, where the items along the lowest dimension W are stored contiguously in memory. Successive elements in H dimension are stored at a distance of W, and those in C dimension have a distance of $H * W$. Placement of data into different dimensions leads to multiple ways of storing the data in the memory. For example, with four dimensional data, there are 24 ways of storing the data in the memory. Li et al. [11] study the performance impact of data-layout on different CNN layers, since data-layout decides the grid and block dimensions in the GPU. The layouts used in various frameworks are shown in Table 7(a).

Optimizations to CONV layer: Li et al. [11] show that NCHW and CHWN layouts lead to large performance difference for both CONV and FC layers. Moreover, no layout provides higher performance across all the layers. To find the optimal layout, they study CONV layers from multiple CNNs, such as AlexNet, LeNet, etc. These layers use only a few distinct values of N (batch size), which are all multiple of 16. Hence, using N as the lowest dimension is reasonable since it facilitates memory coalescing due to corresponding thread-organization. Further, generally, the images are square, that is, the values of H and W are equal. However, the values of H and W vary across the CONV layers between 12 to 224. Also, the depth of input fmaps (C_i) is one for gray-scale images and three for RGB images in the first CONV layer and is a multiple of 16 in the remaining CONV layers. Due to this, memory accesses issued by a warp of 32 GPU threads becomes regular.

Based on these factors, the W and H dimensions can be combined, and N can be kept as the lowest dimension. This leads to two layout candidates: CHWN or HWCN. They note that the performance of HWCN is same as that of CHWN since it does not affect the coalescing characteristics of N dimension and data reuse behavior of remaining dimensions. They further note that since the 2D CONVs are performed on the H and W dimensions, the 4D matrix is generally mapped into a 2D array, and the CONV is performed as MM using NCHW layout. This layout is used in cuDNN and Caffe. Hence, they compare NCHW and CHWN layouts.

They vary the value of N or C, keeping the other values constant and compare the performance of NCHW and CHWN layouts. Since MM has only two dimensions, 2D MM requires matrix unrolling (along H and W) for expanding the input matrix and merging multiple dimensions into two dimensions. This overhead is higher for small-size matrices. Hence, when C is below a threshold, CHWN performs better as it does not require matrix expansion. When C is larger than this threshold, NCHW is better since matrix expansion improves data reuse and parallelism due to the merger of dimensions. From this, they derive a heuristic which is shown in Table 7(b)

TABLE 7
Summary of findings of and heuristics proposed by Li et al. [11]

(a) Layouts used in different frameworks	
Layout	is used in frameworks
CHWN	cuda-convnet
NCHW	Caffe, cuDNN
(b) Heuristic for deciding optimal layout	
Layout	should be used when
CHWN	C is below a threshold T_c (since NCHW incurs high overhead of memory transformation) or N exceeds a threshold T_n (since large N leads to better data reuse and memory coalescing)
NCHW	otherwise
(c) Synergy between data-layout and CONV strategy	
Layout	should be used with
CHWN	Direct-CONV
NCHW	FFT-based CONV or GEMM-based CONV

Changing the layout: They further discuss a technique for changing the layout from CHWN to NCHW. This requires transposing a 4D array, which can be done using a 4D thread-hierarchy. However, this approach is not bandwidth-efficient due to poor coalescing. They propose three strategies for optimizing it. (1) Since the relative positions of C, H and W are the same in both the layouts; they combine them in a single dimension CHW. This allows lowering the 4D transformation into a 2D transformation from [CHW][N] to [N][CHW]. For this, the matrix is flattened, and a 2D thread-hierarchy is used. Then, ShM based tiling is used for achieving memory coalescing. (2) In Kepler architecture, ShM allows both 4B and 8B accesses. For enjoying the bandwidth efficiency of 8B accesses, they group two successive floats into a single `float2` variable of vector type. (3) At the time of write-back, a tile is scattered into multiple successive rows depending on the tile shape, and every vector variable writes in a coalesced fashion. This boosts BW usage by doubling the number of GlM access transactions. This vectorization approach is used when N equals or exceeds 64.

Synergistic use of layout and CONV strategy: They further compare FFT-based CONV with GEMM-based CONV and note that FFT-based CONV gives superior performance when the batch size or filter kernel is large, or the number of channels is high. Otherwise, FFT-based CONV performs poorly since the overhead of kernel-padding, multiple kernel launches, and memory-streaming nullify its benefits. While selecting a data layout, their technique also selects the best implementation of CONV, since different implementations of CONV work best with different layouts. The combinations providing best results are shown in Table 7(c).

Optimizations to pooling layer: They further find that pooling layers achieve much lower bandwidth in both cuDNN and Caffe than what they achieve in cuda-convnet. The reason behind poor access efficiency is the use of "NCHW layout" in pooling layers which causes strided memory accesses. Further, in all three libraries, many GlM accesses are redundant. For example, on doing pooling operation on 12 elements with a stride of 2 and window size of 4, 20 accesses to GlM are required for producing five output elements. However, as shown in Figure 5, most of these accesses are redundant. The fraction of redundant accesses is even higher for 2D images than with 1D images. To improve GlM access efficiency, they propose using CHWN layout. Also, the number of output elements of pooling layer, which are to be computed by each thread, is increased by a factor. The input elements of each thread are cached in RF, and hence, they are fetched from GlM only once. The expansion factor is selected using the hill-climbing strategy with a view to striking a balance between data reuse and register usage.

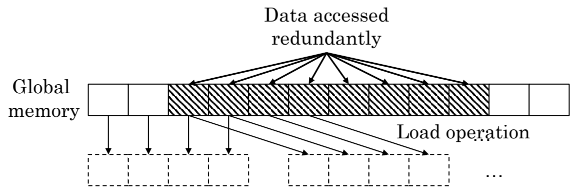


Fig. 5. Illustration of redundant GlM accesses in pooling operation [11]

Optimizations to softmax layer: The bandwidth attained by softmax layers is also much lower than the peak bandwidth. This is because, for meeting the inter-layer dependency constraints, Caffe and cuda-convnet use a different kernel for every step. Interim outputs of successive kernels are sent to and fetched from GlM which is inefficient. Further, the code has two nested loops: the outer loop over the batch of images and the inner loop over categories. The inner loop is not parallelized due to data dependency. The outer loop is parallelized using threads, but this is not sufficient for hiding GPU instruction latency since the batch size is usually small (e.g., 128). To address these challenges, they fuse the kernels, so that the data-transfer happens through ShM or register and not GlM. Also, to improve parallelism, they use threads for parallelizing the inner loop and use ShM for performing reduction operation. Overall, their techniques improve the performance of both individual layers and the overall network.

3.5 Optimizing data reuse

Chen et al. [14] note that lowering CONV into GEMM leads to wastage of memory bandwidth due to duplication of input features and limits possibility of data-reuse. Also, pruned CONV is not efficient on GPU due to irregular computations and memory accesses. They propose performing direct CONV which does not suffer from the above limitations. They further discuss several optimizations to direct CONV. To avoid the overhead of constructing the lowered matrix in memory, cuDNN [9] lazily loads the input matrix into cache at runtime, instead of creating it in the off-chip memory. They use a similar approach adapted for direct sparse CONV. Input fmaps are stored in a 1D array, and the element at a suitable index of the input array is loaded into on-chip memory at runtime. After completion of the computation, the output is stored at a suitable index position. This approach incurs overhead due to computing index values for both input and output arrays dynamically. Still, it is advantageous since it enhances the arithmetic intensity of sparse computation, which is memory-bound. Before performing CONV, the weight matrix is stretched, so it matches the input array dimension.

To reduce memory divergence, they use a dataflow which avoids irregular computations by computing the multiplications separately. The key idea is shown in Figure 6. CONV of a 3x3 filter (with only two non-zero entries) with a 6x6 input feature is divided into the summation of products of non-zero weights with 4x4 sub-matrices.

As for data-to-thread mapping, they note that if array elements with consecutive row/column indices are stored contiguously, then, the accesses to the input array are coalesced. In their technique, one

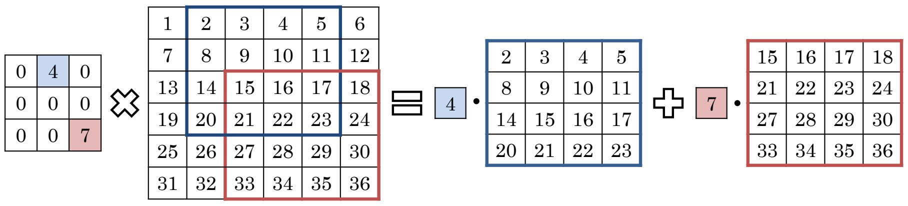


Fig. 6. Dividing a sparse CONV into summation of products of non-zero weights [14]

thread computes one element of output matrix. Since successive threads compute successive outputs, write-accesses to output array are contiguous. Thus, most accesses to GPU memory are coalesced which improves efficiency.

Their technique tries to exploit reuse of all data structures: inputs, weights, and partial sums. A thread-block handles computation of one output channel. The weights are stored in the cache, and input fmaps are streamed into the SM. To benefit from the overlap of input features across various sliding windows, input fmaps are also saved in cache for reuse. Figure 7 illustrates an example of exploiting "data reuse" during sparse CONV.

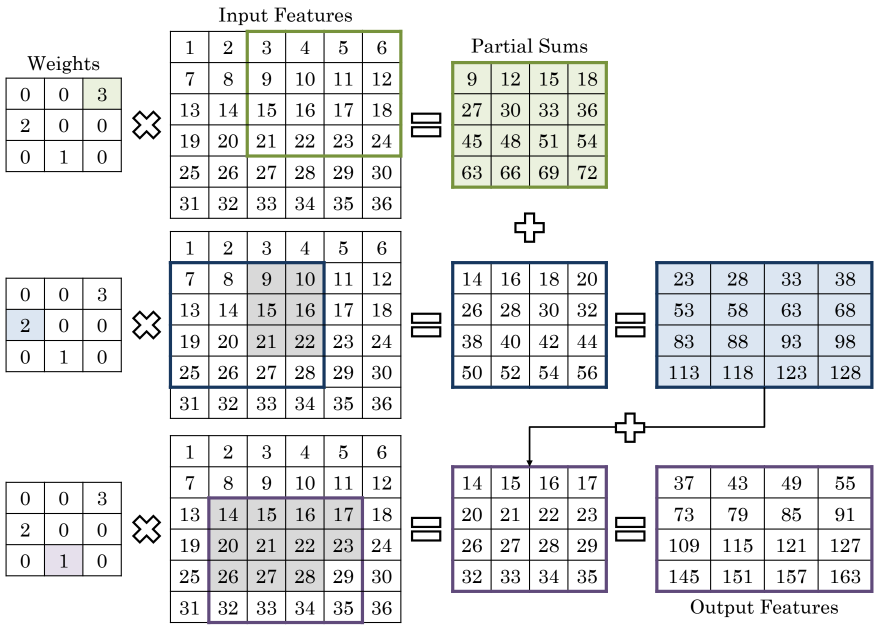


Fig. 7. Illustration of data reuse in sparse CONV [14]. Colored boxes show the data that are accessed repeatedly.

As for storage of different structures, (1) weights are loaded to ShM by threads in a block which leads to coalesced accesses to GlM. (2) Since the input data is not changed, it is stored in read-only cache to reuse it across thread-blocks on the same SM. (3) Partial sums are stored in RF to achieve efficient accumulation. Finally, they propose specific optimizations for specific values of parameters, such as stride, batch-size, filter size, etc. There technique provides higher performance than both cuBLAS and cuSPARSE libraries.

Mitigating bank-conflicts problem in ShM: Let the width of ShM be W_{ShM}. W_{ShM} equals 8 for Kepler GPU and 4 for older GPUs. Let the width of datatype be W_C, for example, W_C equals 4 for integer datatype. Let $Q = W_{ShM}/W_C$. If $Q = 1$, there are no bank-conflicts in ShM. However, if $Q > 1$, then multiple threads access the same bank, which leads to conflicts. Figure 8(a) shows this situation for $Q = 2$. In such cases, W_C can be increased by Q times, such that every thread retrieves Q elements in each access, which improves ShM BW. This case is shown in Figure 8(b). As an example, MAGMA is more efficient than cuBLAS on Fermi ($W_{ShM} = 4$) but is less efficient on Kepler ($W_{ShM} = 8$) due to the bank-conflicts. However, matching W_C with W_{ShM} improves MAGMA performance significantly. Such optimizations are especially important for workloads that show sensitivity to ShM BW.

Chen et al. [12] note that in CONV, input pixels are reused in both horizontal and vertical directions. For example, in Figure 9(a), pixels b, c, f, g, j, and k are reused in the horizontal direction and e, f, g, i, j and

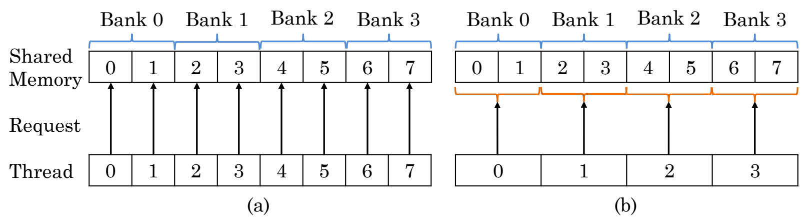


Fig. 8. (a) When $W_{ShM} = 2W_C$ ($Q = 2$), eight threads access four banks of ShM, which leads to bank-conflicts (b) On increasing W_C by Q times [12], each thread retrieves Q elements in each access. A bank is accessed by a single-thread, which avoids bank-conflicts.

k are reused in the vertical direction. Assuming F filters, each of size $K \times K$, an input pixel may be reused up to $F \times K \times K$ times. Chen et al. present a technique for exploiting data reuse for optimizing direct CONV on GPUs. They discuss their technique for two cases: (1) when there is only one input channel (e.g., grayscale images) and (2) general case of multiple input channels.

Case 1: single input channel: Here, every thread stores $K \times K$ pixels of the input in the register. The challenge in achieving reuse is that when a thread performs the next-horizontal CONV, some pixels required for next-vertical CONV are lost. They divide the input image into tiles of size $U \times V$ which allows data-sharing across vertical dimension, as each row of the input can be used in CONV of K rows. A block of V threads operates on each image tile, and thus, different tiles are processed concurrently by different thread-blocks. The V threads operate in parallel on one row at a time, until they reach the last row. This is illustrated in Figure 9(b).

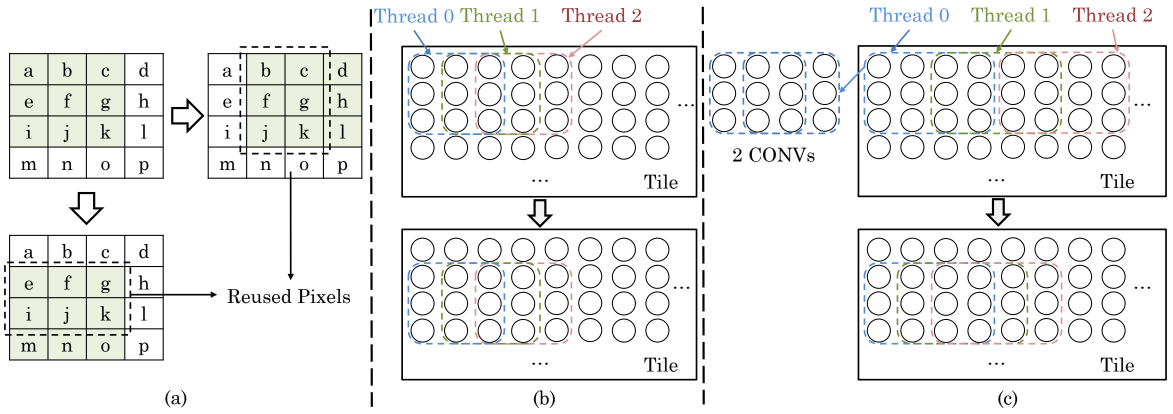


Fig. 9. (a) Data reuse in CONV. (b)-(c) CONV approach of Chen et al. [12] for case 1 ($K = 3$). (b) When $W_{ShM} = W_C$ (c) When $W_{ShM} = 2W_C$ (i.e., $Q = 2$). Here, each thread performs Q CONVs to obtain Q contiguous output pixels of a row.

In case of the single input channel, filters are generally small, and hence, they are stored in constant memory. All threads of a warp perform CONV using the same filter concurrently, so they access the same address. This improves the efficiency of constant memory.

The input is stored in GlM. For performing CONV, data sharing in the horizontal dimension is achieved using ShM and data reuse in the vertical dimension is achieved using registers. Thus, each input pixel is read only once from GlM, which improves GlM access efficiency. When $Q = 2$, which is the case for Kepler, each thread-block has V/Q threads. Every thread accesses and computes Q contiguous output pixels in every row and $Q \times U$ output pixels in the tile. When $Q = 2$, built-in datatype `float2` is used. Figure 9(c) shows their CONV approach for Kepler GPU ($Q = 2$). Their approach increases register-requirement of each thread slightly. Since successive threads access contiguous addresses, accesses to both ShM and GlM are optimized. For case 1, they find the optimal values as $U = 8$ and $V = 256$.

Case 2: multiple input channels: In this case, the pixels of single CONV cannot fit in the registers. Due to this, one CONV operation needs to be split into multiple parts, and the interim results are aggregated in the registers. Further, with an increase in the number of channels, the filter size also increases and hence, they need to be stored in GlM instead of constant memory.

Similar to case 1, the input image is divided into tiles of size $U \times V$. A 2D thread-block layout of size $B_X \times B_Y$ is used. In Y-direction, a thread-block handles C tiles at the same position of all C channels. In

X-direction, a thread-block handles F_B successive filters, where $B_X = \lceil F/F_B \rceil$, and F is the number of filters. Threads in a block are organized in the 2D layout of size $T_X \times T_Y$. Every thread handles V_T output pixels and F_T filters, where $T_Y = V \times U/V_T$ and $T_X = F_B/F_T$. Every thread uses $F_T \times V_T$ registers to store the interim CONV results for F_T filters and V_T pixels.

C_{ShM} channels of image tiles and filters are stored in ShM. When loading filters from GlM to ShM, padding is required to avoid bank-conflicts since transpose operation is performed. By comparison, for loading image tiles, padding is not required. A single thread operates on V_T contiguous output pixels, which reduces accesses to ShM compared to the case when different threads operate on these pixels. At a time, only one row of input pixels is stored in the register of every thread and accumulation of CONV results happens in multiple iterations.

Every thread reads Q contiguous filter values at a time from ShM along the horizontal dimension, which avoids bank conflicts. Reading of image tiles from ShM is also conflict-free since T_X contiguous threads in X direction read the same address which allows the use of the broadcast scheme of ShM. The limitation of their approach is that storing the output to GlM does not lead to coalesced accesses since nearby threads in X-dimension process distinct output fmaps. However, this has a negligible impact on performance since writing-back the output takes a small amount of time. Table 8 shows the optimal values of different quantities at various filter sizes for case 2. On Kepler GPU, their technique achieves higher performance than the cuDNN library for both the cases. The performance improvement is much higher when there is only one input channel.

TABLE 8
Optimal values of different quantities for various filter sizes for case 2 [12]

Filter size	3x3	5x5	7x7
U	4	8	4
V	32	32	64
F_B	64	32	32
V_T	16	8	8
F_T	4	8	8
C_{ShM}	2	1	1

The limitation of the technique of Chen et al. [12] is that since it assigns a fixed amount of data to every SM, it does not work well when the fmap size is less than 32. In state-of-the-art CNNs, most CONV layers operate on images of less than 32 size and hence, their technique is not effective for recent CNNs.

3.6 Optimizing tiling and batching schemes

Li et al. [5] note that tiling and batching are correlated. Data reuse can be increased by increasing the tile size, but this reduces TLP and thus, reduces the optimization space for the batching technique, and vice versa. Further, although cuBLAS provides a batched GEMM function (`cublasSgemmBatched`), it can batch the GEMMs with the same size only. However, in real-world DNNs, the dimensions of GEMMs being batched vary greatly [5]. They present a synergistic technique for tiling and batching for improving performance of GEMMs on GPUs. Their technique first performs tiling of GEMMs and then, performs batching, i.e., assigns the tiles to thread-blocks. Figure 10 presents the overview of their technique. Let C=A*B, where the dimension of A and B are MxK, and KxN, respectively. They note that the value of M*N affects TLP since when M and N are large, the number of tiles and the tile size is high. Similarly, the value of K decides the workload of each tile and thus, affects ILP.

They note that the tile size that is optimal for a single GEMM is not optimal on using batching since each GEMM in the batch may prefer different tile-size and the optimal tile size also depends on the number of GEMMs that are batched together. On using the same tile size for all GEMMs, some threads in a thread-block may remain idle.

A tiling scheme is characterized by the tile size (BY x BX x BK) and the total threads that execute the tile. Each thread executes one sub-tile of the tile and their size is shown in Table 9. The number of threads required by a tile is obtained by dividing (BY x BX) by the size of the sub-tile. Table 9, heading “Single GEMM”, summarizes the 6 tiling schemes of previous works. For example, the number of threads required by the ‘huge’ scheme is (128*128)/(8*8) = 256.

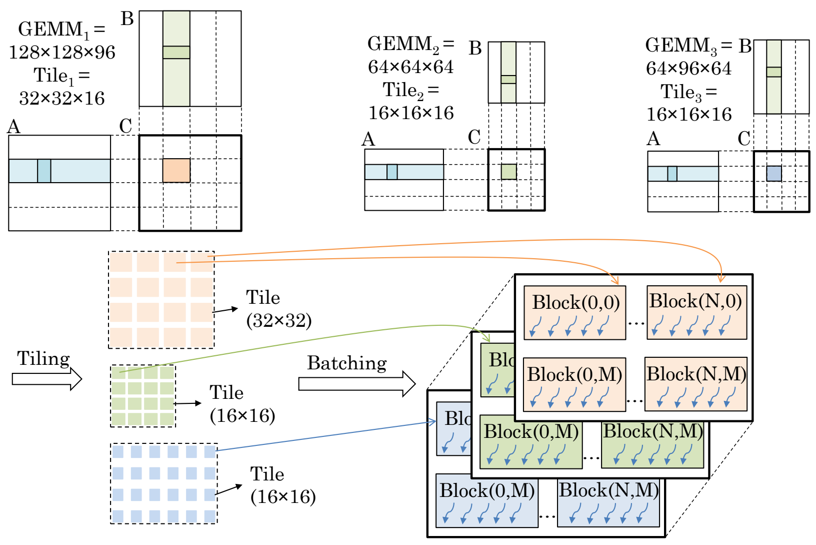


Fig. 10. Coordinated tiling and batching approach of Li et al. [5]

TABLE 9
Tiling schemes for single and batched GEMM [5]

Tiling scheme	BY	BX	BK	Single GEMM		Batched GEMM [5]	
				Sub-Tile Size	Threads	Sub-Tile (128-Threads)	Sub-Tile (256-Threads)
small	16	16	8	4×2	32	2×1	1×1
medium	32	32	8	4×4	64	4×2	2×2
large	64	64	8	8×8	64	8×4	4×4
tall	128	64	8	8×8	128	8×8	8×4
wide	64	128	8	8×8	128	8×8	8×4
huge	128	128	8	8×8	256	16×8	8×8

They propose a tiling scheme for batched GEMM, where sub-tile size is chosen such that the number of threads of a tile is the same (either T=128 or T=256). These tiling schemes are shown under the heading “batched GEMM” in Table 9. Use of same thread-block size avoids the thread-idling issue. With increasing T, there are more threads for exploiting TLP, whereas on decreasing T, the sub-tile size, and hence, the work to be performed by each thread increases. Hence, they provide two choices (T=128 or T=256), and of these, their technique selects one version for a certain GEMM. They keep BK as constant (8) and only change BX and BY.

For deciding the tiling scheme, they propose an algorithm which prioritizes TLP and then tries improving ILP. For all the GEMMs, their algorithm first tries to use sub-tile size corresponding to 256-threads since it has higher TLP than their 128-thread counterpart. The algorithm works in three phases, which are shown in Table 10.

TABLE 10
Algorithm for finding the tiling strategy [5]

Phase 1	For each GEMM, feasible tiling schemes, i.e., those with $BY \leq M$ and $BX \leq N$ are selected and kept in a priority queue such that smaller tiling scheme has higher priority. If the queues of all GEMMs have only one tiling scheme, the algorithm switches to schemes with 128-threads and performs phase 2.
Phase 2	A tiling scheme for every GEMM is popped from its priority queue, and the overall TLP is estimated for all the GEMMs.
Phase 3	If the TLP exceeds a threshold, a larger scheme is evaluated by repeating phase 2 for improving ILP at the cost of TLP. Else, the existing scheme is chosen as the final solution. The threshold is chosen based on the GPU architecture, e.g., the threshold is 65536 for V100 GPU.

After tiling, the batch of GEMMs is changed into the batch of tiles. Their technique assigns more than one tile to a thread-block for exploiting ILP, especially for small values of K. For finding the assignment, they design two heuristics: (1) threshold batching which prioritizes TLP but also ensures that the workload of a tile is more than a threshold (for example, 256 for V100 GPU) (2) binary batching which gives more priority to ILP and batches at most two tiles at a time. From these two heuristics, one is selected using "random forest" algorithm. They perform experiments using synthetic GEMM and GoogleNet and show that their technique outperforms the MAGMA library.

Oyama et al. [41] note that cuDNN works by selecting the best CONV algorithm out of eight possible CONV algorithms such that memory budget constraints are met. However, if the workspace required by an efficient algorithm exceeds the budget, cuDNN uses a slower algorithm with a smaller memory requirement. Due to this, fast algorithms such as FFT or Winograd may be used only if very large memory space is available.

They present a wrapper, termed "micro-cuDNN" to cuDNN which splits the mini-batch into multiple micro-batches and finds CONV algorithm to be used for each micro-batch. Then, it performs CONV on each micro-batch serially. A reduction in effective mini-batch size reduces memory requirement and hence, allows using a faster CONV algorithm. Figure 11 shows the comparison between the working of cuDNN and "micro-cuDNN". For utilizing the workspace, they present two schemes which are shown in Table 11.

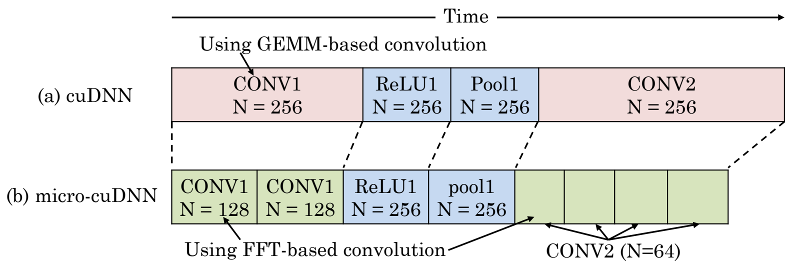


Fig. 11. Comparison between cuDNN and micro-cuDNN (N is the mini-batch size) [41]

TABLE 11
Schemes for utilizing the workspace in the technique of Oyama et al. [41]

Workspace reuse scheme	Workspace division scheme
One workspace is allocated to each layer, which is shared by different micro-batches.	A single workspace is allocated to the entire network from which, different portions are allocated to different layers. This allows small groups of CONV operations, such as those used in Inception module, to execute together with bigger workspaces. In this scheme, the workspace management is performed by their technique and not the DL framework since those frameworks do not view the workspace needs of the network as a whole.
The sizes of micro-batches are found using dynamic programming technique to minimize overall execution time.	Finding the mini-batch size in this scheme is more challenging than that in "workspace reuse" scheme because of the interdependency between the configurations of different CONV kernels. 0-1 ILP is used for solving this problem under memory constraint and with the goal of minimizing total execution time.

The optimal configurations and benchmarking results can be stored for later reuse by layers with similar characteristics on the same or other GPUs. They integrate micro-cuDNN in Caffe and TensorFlow. Their technique improves hardware usage efficiency without compromising on training accuracy. However, their technique does not provide improvement when the workspace size is too small or too large since in both cases, splitting into micro-batches provides no improvement over cuDNN. Further, the reduction in mini-batch size reduces reuse opportunity of weights.

Holmes et al. [8] compare three RNN frameworks/libraries on GPU, viz., cuDNN, TensorFlow, and TensorRT. For several model sizes, GPU implementation of TensorFlow is 90× slower than its CPU implementation. The GPU implementation of TensorFlow packs eight independent MMs of LSTM (long short term memory) into one MM for increasing the workload of each kernel for improving throughput. Still, it is inefficient since at least one kernel is launched for every iteration, and the barrier between the kernels leads to large overhead. Also, in every iteration, the whole weight matrix is loaded from GlM or L2 cache. TensorRT also suffers from a similar issue. While cuDNN achieves lower latency than DeepCPU [63] for a majority of configurations, it shows poor scalability to even small batch sizes (e.g., five) and thus, fails to benefit from the parallelism of GPU. Also, the fraction of GPU's peak performance achieved by cuDNN is lower than the fraction of CPU's peak performance achieved by DeepCPU. They present a GPU-library for enabling efficient implementation of RNN on GPU.

Since the weight matrix of typical RNN cannot fit in a single SM, it needs to be partitioned across SMs. There are two strategies for doing this: input-based tiling (shown in Figure 12(a)) and output-based tiling (shown in Figure 12(b)). For performing two independent MMs on a GPU with four SMs, input-based tiling runs every MM on two SMs. Each SM stores half of the weight matrix which mitigates RF capacity issue, but requires one "global synchronization" (GS) after MM and another GS after element-wise operators. Thus, if there are 8 MMs (e.g., in LSTM), then, 8 GSes are required after MM.

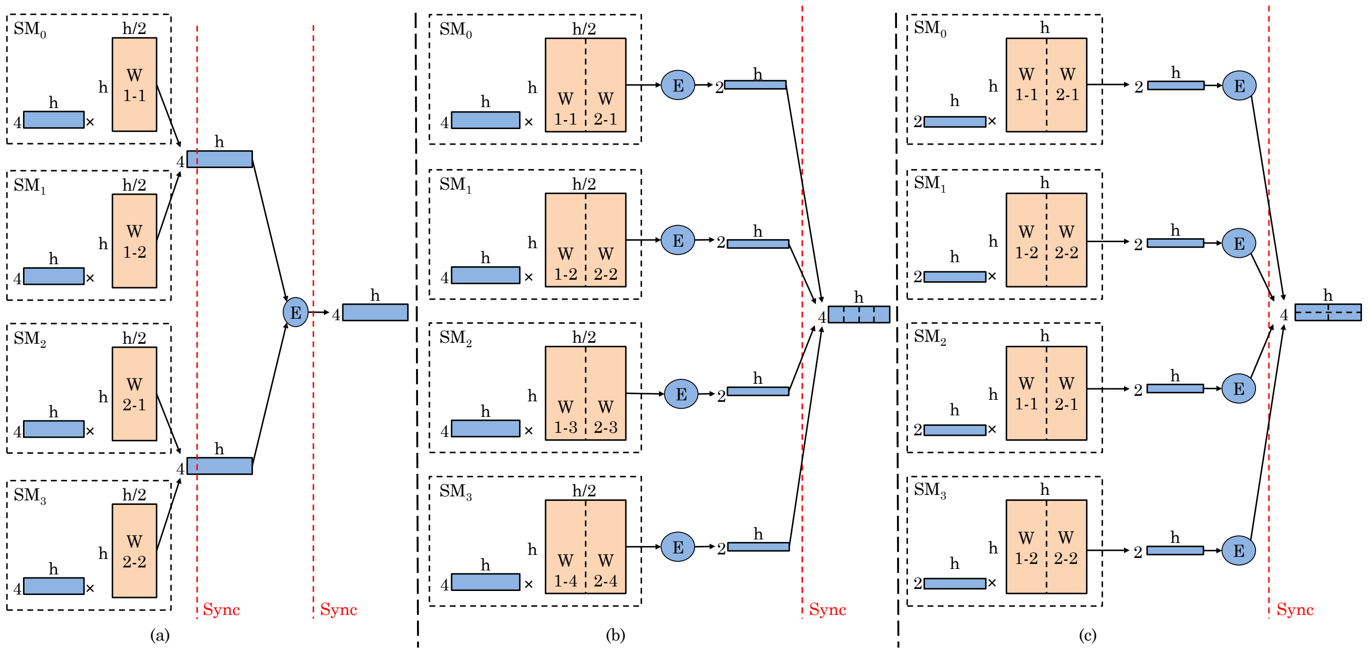


Fig. 12. (a) Input-based tiling (each MM is separately tiled) (b) Output-based tiling where output is vertically split (c) Output-based tiling where the output is split both vertically and horizontally

Their technique uses output-based tiling where every SM is responsible for computing one output tile, as shown in Figure 12(c). It co-locates those weights from different weight matrices on the same SM that are needed for producing the given output tile. In output-tiling, only one GS is required since element-wise operations happen inside a single SM only. Performing only one GS in every time step is efficient, which is the minimal number of GS required to make progress to the subsequent time-step. Since GRU has dependent MMs, a different strategy is used for this.

After this, the weights of every tile are loaded to RF which persist over different steps of the RNN calculations. After this, at each time step, the global hidden state H is replicated in the ShM of every SM and threads are mapped to the weight matrix. Then, optimized computations are performed to generate an updated local version of H. At last, all SMs are synchronized, and local versions of H are merged into the global copy. If the model cannot fit in the RF, their technique defaults to fusing different MMs since high data parallelism itself enables leveraging GPU resources.

As for the strategy to map work to threads, every warp computes a different output tile. Thus, every warp performs a partial MM, which requires loading state vector in RF and computing the partial sum. By performing the reduction on partial sums, the final output is produced. For optimizing the above steps, they propose three schemes (1) using an entire warp to sequentially generate the output elements (2) assigning NumThreads/NumOutputElements to every output element (3) using adaptive-size work units, each of which generates selected output elements in a sequence. Of these, they use scheme (3) due to its flexibility.

Different choices of mapping operators to SMs and threads to compute-elements leads to huge configuration space. Their technique uses a lightweight performance model to select the best M configurations. Then, M kernels, each corresponding to one configuration, are compiled and based on a profiling phase, the most performant kernel is finally selected. The performance model combines an analytical model with a low-overhead measurement approach and is very accurate. Overall, their technique outperforms the best CPU and GPU implementations and also improves resource utilization.

Note that of the works discussed in this section, Li et al. [5] perform tiling and batching in coordinated manner, whereas Oyama et al. [41] perform only batching and Holmes et al. [8] perform only tiling.

3.7 Optimizing pruning schemes

Yu et al. [4] evaluate "deep compression" technique using five CNNs on three processors: a processor with low-parallelism (microcontroller), modest-parallelism (CPU) and high-parallelism (GPU). They use a batch size of 1 on microcontroller and CPU, and 50 on GPU. They find that in spite of removing 80% of the weights, pruning harms the performance of eight configurations. On the rest of the configurations, performance improvement is much lower than the decrease in MAC operations. This happens because unlike dense matrices, sparse matrices are not regular. This is evident from Figures 13(a)-(b). Also, as shown in Figure 13(c), sparse matrices require additional memory and computations for storing and decoding the sparse format.

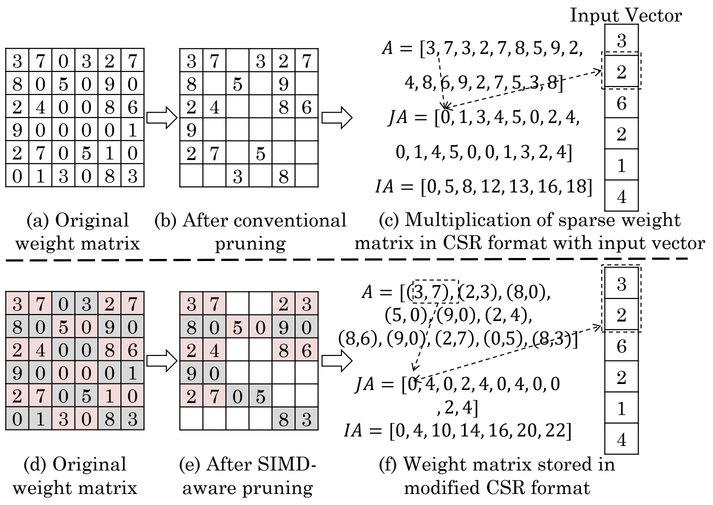

Input Vector
A = [3, 7, 3, 2, 7, 8, 5, 9, 2, 4, 8, 6, 9, 2, 7, 5, 3, 8]
JA = [0, 1, 3, 4, 5, 0, 2, 4, 0, 1, 4, 5, 0, 0, 1, 3, 2, 4]
IA = [0, 5, 8, 12, 13, 16, 18]
(a) Original weight matrix
(b) After conventional pruning
(c) Multiplication of sparse weight matrix in CSR format with input vector
A = [(3,7), (2,3), (8,0), (5,0), (9,0), (2,4), (8,6), (9,0), (2,7), (0,5), (8,3)]
JA = [0, 4, 0, 2, 4, 0, 4, 0, 0 , 2, 4]
IA = [0, 4, 10, 14, 16, 20, 22]
(d) Original weight matrix
(e) After SIMD-aware pruning
(f) Weight matrix stored in modified CSR format

Fig. 13. (a)-(b) Conventional pruning technique (e.g., deep-compression) (c) multiplication of a sparse weight matrix stored in CSR format with input vector (d) aligning the weights in groups of SIMD width (two) [4] (e) SIMD-aware pruning (f) storing a sparse weight matrix in modified CSR format allows SIMD-multiplication

When the unpruned network runs on the microcontroller, the memory latency cannot be hidden due to the simple design of microcontroller. Hence, a decrease in model size due to pruning compensates the inefficiencies of sparse MM. Hence, pruning improves performance on the microcontroller. On the GPU, pruning reduces the performance of all the networks, since sparse MM forgoes optimizations such as memory-coalescing and matrix tiling. On CPUs, the impact of pruning depends on the relative fraction of computations performed in FC and CONV layers. Pruning improves the performance of FC layers since a decrease in memory accesses boosts the performance of matrix-vector multiplication operations. However, since CONV layers perform MM, where the weights are reused several times, a decrease in memory footprint brings marginal benefit. Hence, pruning hurts performance of CONV layers on CPU.

Yu et al. present a technique for customizing pruning to the processor architecture. Figure 14 summarizes the working of their technique. They classify the processors into three types according to their parallelism. (1) Processors with low-parallelism have in-order cores with no cache and limited storage. For example, "ARM Cortex-M4" features a 2-way SIMD design and a 3-stage "inorder pipeline". (2) Processors with modest parallelism utilize ILP and MLP in addition to SIMD units. They also have a deep cache hierarchy and huge memory capacity. Examples of this are out-of-order CPUs. (3) Processors with high-parallelism such as GPU utilize TLP for further improving performance. Their throughput is sensitive to BW and not to the DNN model size.

On the low-parallelism processor, their technique performs "SIMD-aware weight pruning" of DNNs. First, the weights are classified into aligned groups of size same as the SIMD width. For example, on a 2-way SIMD, the weights are divided into groups of two. Then, the groups, whose "root-mean-square"

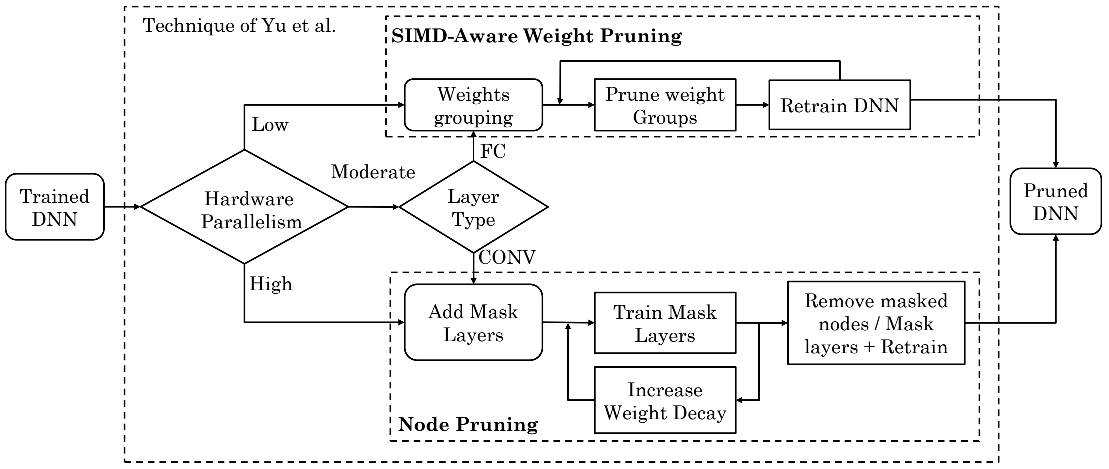


Fig. 14. Overview of the technique of Yu et al. [4]

value of weights is below a threshold, are pruned. These steps are shown in Figures 13(d)-(e). Then, the pruned matrix is retrained. Pruning and retraining are repeated until the accuracy of retrained DNN falls just below that of the original network. Layers with low redundancy or those showing no performance benefit from pruning are not pruned. Also, the dropout ratio is adjusted during training. The "sparse weight matrix" is stored using a modified CSR format, shown in Figure 13(f), where column index of only the first element in every group is stored. This reduces the model size. As the inputs are present in contiguous locations, a single SIMD instruction is sufficient to load them, and this reduces the number of instructions.

For the highly-parallel processor such as the GPU, their technique performs node pruning. In FC layers, one neuron is taken as one node, and in CONV layers, one fmap is taken as a node. Node pruning uses mask layers for selecting nodes which are not important so that their output can be blocked. After mask layers are trained, the blocked nodes are eliminated, and after all redundant nodes are eliminated, mask layers are also removed. Finally, retraining is performed for obtaining the pruned network. Node pruning does not make the network sparse, and thus, the overheads of the sparse network are not incurred. Node pruning reduces the layer-size but not as much as the weight pruning. Still, on GPUs, node pruning leads to larger throughput than the weight pruning technique.

On moderately-parallel processors, they first apply node pruning to CONV layers and then, they apply SIMD-aware weight pruning to FC layers. On different processor platforms, their pruning technique brings larger improvement in performance than naive pruning, while reducing the model size and without losing prediction accuracy.

Hill et al. [13] note that hardware-unaware pruning such as deep-compression technique harms performance on GPU due to increasing branch divergence and uncoalesced memory accesses. Figures 15(a)-(b) compare the computations in the baseline and pruned DNN. Further, improving off-chip BW provides only limited benefit in DNN execution on GPU. By comparison, many DNNs are bottlenecked by on-chip memory (i.e., ShM) BW since on applying loop tiling at register level, pressure on off-chip BW is reduced, however, ShM BW continues to be a bottleneck due to the limited capacity of RF.

They propose two techniques for improving DNN execution on GPUs, which work on the observation that DNNs are resilient to both weight/neuron pruning and the use of low-precision weights. In the baseline approach, a $Q \times K$ weight matrix is multiplied with a $K \times N$ output matrix from the last layer. This is shown in Figure 16(a). Their first technique termed "synapse vector elimination", preprocesses these matrices in 2 steps. In step 1, the synapses are reordered so that unimportant synapses can be discarded without losing regularity of data structure. Taking the example of the weight matrix, since the number of discarded synaptic weights (P) is known beforehand, column swaps are performed to bring the retained columns towards left side and discarded columns towards right side of the matrix. Then, the matrix is partitioned at column $K-P$, so that the left side $K-P$ columns and the right side P columns show retained and discarded (respectively) synapse groups. This is shown in Figure 16(c). This step has minimal overhead and is more efficient than naively copying the retained synapses into an extra buffer.

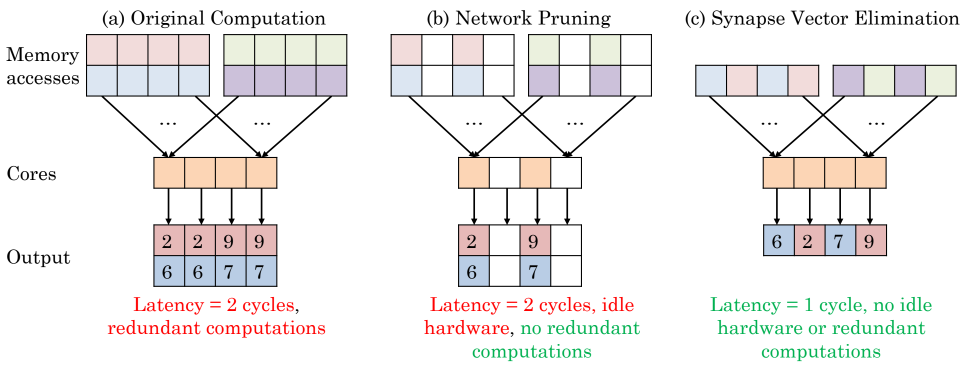

(a) Original Computation
(b) Network Pruning
(c) Synapse Vector Elimination
Memory accesses
Cores
Output
Latency = 2 cycles, redundant computations
Latency = 2 cycles, idle hardware, no redundant computations
Latency = 1 cycle, no idle hardware or redundant computations

Fig. 15. (a) Naive DNN execution leads to redundant computations (b) pruning technique leads to hardware-underutilization (c) “synapse vector elimination” improves performance and resource utilization [13]

In step 2, to reduce matrix-dimension, multiplication between weight and input is stopped at $(K-P)^{th}$ column instead of stopping at the K^{th} column. Thus, their technique reduces the number of operations in MM from $Q \times K \times N$ to $Q \times (K-P) \times N$, as shown in Figure 16(b). This technique improves the efficiency of both storage and computation, as shown in Figure 15(c). For compensating for the discarded synapses, the magnitude of retained synapses is increased to ensure that the expected value of optimized and baseline output is close.

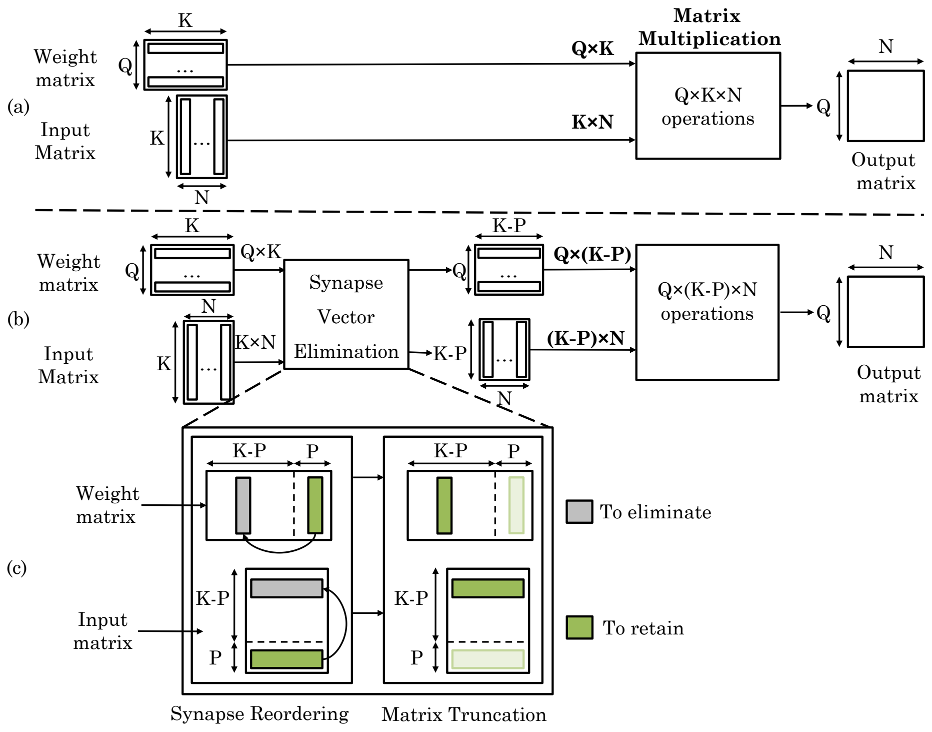


Fig. 16. (a) Precise MM in the baseline DNN (b) “synapse vector elimination” reduces [13] (c) internal functioning of “synapse vector elimination”

The procedure for finding non-contributing synapses is shown in Table 12. By selecting the value of threshold α, a tradeoff between accuracy and performance can be exercised.

Their second technique seeks to mitigate the issue of limited ShM BW. Here, before writing the register values to ShM, multiple values are packed into one element. Similarly, before reading from ShM, the values are unpacked, and then, computation is performed on them. Figure 17 shows the steps in their second technique. Since off-chip BW is not a bottleneck, this technique is not applied on the off-chip link. Since the original data is single-precision (32b), they study 3 low-precision formats for allowing packing. These formats seek to remove the unimportant bits from the representation of a value. These formats are described in Table 13 and illustrated through Figure 17.

TABLE 12
Steps for finding non-contributing synapses [13]

Step 1	Find the correlation $\rho_{i,j}$ of each synapse vector i with each remaining vector j
Step 2	If $\rho_{i,j}$ exceeds a threshold (α), vector i is said to represent j
Step 3	Compute R_i as follows $R_i = \sum_{j=1}^{N} F \mid F = 1$ if $\rho_{i,j} \geq \alpha$ and $F = 0$ otherwise
Step 4	The vector i with largest R_i is retained during "synapse vector elimination", whereas those represented by the retained one are eliminated.
Step 5	Repeatedly apply above steps on the rest of the vectors. Stop when all the vectors are either discarded or retained.

TABLE 13
Three low-precision formats used by Hill et al. [13]

IEEE 754 "half-precision format"	It uses a CUDA scheme for converting between half and single precision which uses custom hardware for bit-conversion. However, the use of the half-precision format leads to slight performance loss due to data-reformatting operations.
Deft-16	It is simply the 16 MSBs of single-precision format and conversion between the two formats requires only bitwise and shift operations. By virtue of using seven mantissa and eight exponent bits, "Deft-16" achieves reasonable dynamic range and precision.
Deft-16Q	It allows MSBs of one value to be stored in LSBs of another value. Compared to Deft-16, Deft-16Q reduces one "logical AND" instruction in the unpacking process, still, it provides much higher performance than Deft-16.

To further reduce the reformatting overheads, they also discuss a specialized hardware unit which performs unpacking in just one cycle. They perform experiments over a range of DNNs. Since modern GPUs have many more 32b ALUs than 16b ALUs, naively using 16b storage and computation leads to large performance loss. By comparison, their packing/unpacking based technique improves performance by virtue of utilizing 32b ALUs. Overall, their techniques provide large speedup.

3.8 Coalescing and scheduling GPU kernels

Jain et al. [15] note that since interactive queries have strict deadlines, DNN inference for them requires small batch-sizes. Since hardware resources are provisioned for dealing with the worst-case demand, use of small batch sizes leads to low resource-usage efficiency and hence, financial losses. For example, at interactive latency values, ResNet-50 achieves less than 25% of the peak throughput of V100 GPU. As the ratio of peak compute throughput to peak memory-bandwidth increases, the problem of GPU underutilization will exacerbate even further.

They first discuss two naive approaches for addressing this issue. (1) Time-multiplexing of CUDA contexts on GPU allows interleaving of kernel execution. Kernels execute serially and not in parallel,

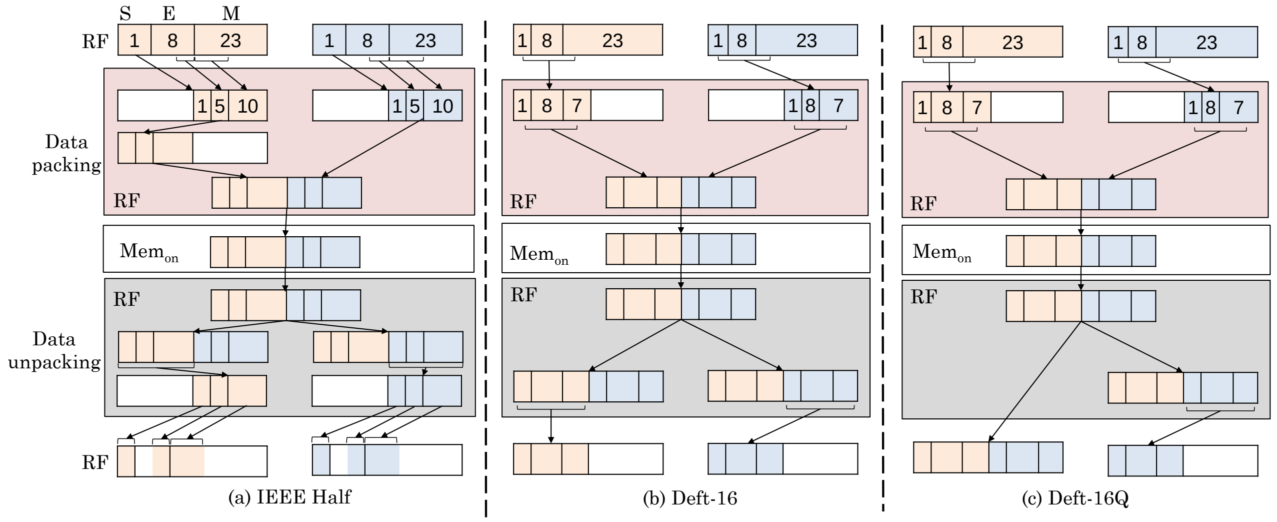


Fig. 17. Packing-unpacking technique of Hill et al. [13] and three low-precision formats used by them

and processes are preempted periodically. Context-switching involves flushing of execution pipeline and hence, incurs large overhead. Due to this, they observe that with an increasing number of interleaved processes, inference latency rises linearly. (2) Spatial multiplexing allows concurrent-overlapping kernel execution. However, it cannot ensure performance-isolation and leads to unpredictability in execution time. Further, kernels optimized for solo-execution provide lower performance when running concurrently with other kernels.

Their proposed technique performs just-in-time coalescing of GPU kernels across multiple streams and over time. Figure 18 illustrates working of their technique. Their approach is similar to how VLIW (very large instruction word) compilers pack kernels for better utilization of system resources. Specifically, conventional GPU programing (e.g., using CUDA) uses an early-binding approach where the thread-block dimension is specified in the source-code itself. They propose using a late-binding approach where for a kernel, only high-level parameters such as operators, the inputs, and latency constraints are specified. Then, a just-in-time (JIT) compiler executes the kernels based on runtime information about the number and requirements of concurrent kernels, GPU context, etc.

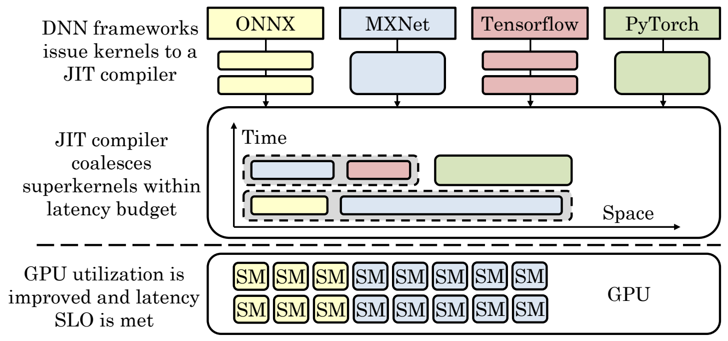

DNN frameworks issue kernels to a JIT compiler
ONNX
MXNet
Tensorflow
PyTorch
JIT compiler coalesces superkernels within latency budget
Time
Space
GPU utilization is improved and latency SLO is met
SM
GPU

Fig. 18. Out-of-order just-in-time compiler performs coalescing and reordering of kernels from different streams [15].

Their compiler also reorders queued kernels for (1) prioritizing streams having tight margin from the deadline and (2) delaying kernels that may provide better packing efficiency when coalesced at a later time. Coalescing latency is overlapped with computation latency of other streams. For ensuring isolation and performance-predictability, the latency of each kernel is monitored and based on it, resource-allocation of kernels is adjusted.

While VLIW compilers perform only ahead-of-time modification, their technique performs both ahead-of-time and runtime tuning. Tuning involves adjusting the parameters of GPU programs. During runtime, several small kernels are packed into a macrokernel which may be further optimized by applying parameters obtained from an autotuning phase. For example, they cluster MM kernels having similar dimension into one macrokernel. Running a macrokernel achieves higher resource utilization than time-multiplexing and is effective also because the operations performed in DNN kernels are very limited. Their technique improves throughput without violating latency constraints.

3.9 Mitigating GPU memory limitations by offloading data to CPU Memory

DNN training requires a significant amount of memory, which may exceed the memory capacity of a single GPU. For example, VGG-16 (BatchSize=256) requires 28GB memory, which is larger than the 12GB memory capacity of Titan X. This forces the user to train the network with smaller batch-size or parallelizing the training over multiple GPUs. Several researchers have proposed techniques for mitigating memory capacity issues of GPUs by utilizing the memory resources of the host. These techniques and their distinguishing features are shown in Table 14. For example, if the cost of computation is much lower than the cost of storing/accessing its output, then repeating the computation provides benefit over storing its results.

Rhu et al. [24] propose vDNN, a runtime memory management scheme for virtualizing the memory utilization of DNNs over both CPU and GPU memories. They note that DNNs trained with “stochastic gradient-descent” algorithm are designed using multiple layers. In DNN training, layer-wise computations are performed such that their order remains fixed and is repeated for billions of training iterations. While

TABLE 14
Works that offload data from GPU memory to CPU memory during training

Transferring intermediate data or parameters to CPU memory	[7, 17, 24, 26, 29, 32, 35, 39, 57]
Use of CUDA stream	[24, 36, 39, 57]
Unique features	technique proposed for non-linear DNNs [28], use of high-BW stacked memory [29], use of unified virtual memory [40], mitigating memory fragmentation issue [26], selecting sub-batch size [26], repeating the computation to avoid saving its output [32, 35], using a single "tensor storage object" for error-terms with same value [32]

training a network, current machine-learning frameworks allocate the memory for accommodating the needs of all the layers because, in the back-propagation algorithm, gradient update is performed in a layer-wise manner. The intermediate output of one layer produced during the FWP phase is termed as feature map (fmap). These fmaps are later reused during BWP phase of the same layer for gradients computation. Hence, all fmaps need to be present in GPU memory until the completion of the BWP phase. This network-wide allocation approach (called baseline) avoids the need for costly page-transfer between CPU and GPU over PCIe.

Although the baseline approach reduces data-transfer overheads, it increases the memory requirement greatly. With an increasing number of layers, the memory reserved for fmaps grows even further. Since the training is performed in a layer-wise manner, only a portion of allocated memory is required at a time and thus, with baseline scheme, up to 80% of the allocated memory may not be used at a time.

They note that "feature-extraction layers" consume a much higher fraction of memory than the "classifier layers", for example, "feature-extraction layers" consume 81% and 96% of the memory in AlexNet and VGG-16, respectively. Further, in "feature-extraction layers", intermediate fmaps and workspace (a temporary buffer required for FFT-based CONV) require much higher memory than the weights. Hence, vDNN focuses on fmaps of "feature-extraction layers". In DNNs, fmaps of layer K, computed during the FWP phase are reused only on reaching the same layer in the BWP phase. Since the reuse of fmaps happens after tens to hundreds of milliseconds (e.g., 1200ms for the first layer of VGG-16(BatchSize=64)), fmaps of DNNs stay in GPU memory for a long time without being reused.

In the vDNN technique, memory allocation is done for the layer that is currently being processed, and the fmaps not required by the current layer are offloaded to CPU memory via PCIe. Then, the offloaded fmaps are released to reduce the GPU memory usage, as shown in Figure 19(a). In non-linear networks, multiple layers may consume the output fmaps of a previous layer. Hence, vDNN tracks these dependencies using a dataflow graph and offloads fmaps of layer-K only when all consumer-layers have used fmaps of layer-K. While performing BWP for a layer, vDNN releases the tensors that are not required for training the BWP of the remaining layers. For example, output fmaps and output gradient maps of layer $K+1$ are not needed during BWP of layer K, since the gradient updates of layer K have already finished. This is shown in Figure 19(b). Input fmaps and input gradient fmaps are not released since they will be required in BWP of the previous layer.

vDNN prefetches offloaded fmaps of layer K and overlaps this with BWP phase of layer L ($L > K$) to hide prefetch latency and bring the offloaded fmaps of a layer before its BWP phase starts. The layer for prefetching is selected to ensure that its data comes neither too early nor late. vDNN works on top of cuDNN and uses two CUDA streams for overlapping DNN computations with memory management operations. Before offloading, a pinned memory portion is created in host memory, and then, the fmaps of a layer are transferred over PCIe in a non-blocking manner, such that this is overlapped with FWP phase of cuDNN. Prefetching of fmaps is performed in the opposite order of the offloading in FWP phase.

The choice of the best layer to offload depends on multiple factors, e.g., CONV algorithm and memory needs of each layer, GPU memory size, and DNN throughput. Since the CONV algorithm which consumes less memory (e.g., GEMM-based) also gives a lower performance, to balance these tradeoffs, they propose two static policies and one dynamic policy for choosing the layer to offload. The motivation behind designing the dynamic policy is that the static policies do not take into account the characteristics of DNN and GPU. These policies are shown in Table 15.

Their technique lowers the GPU memory requirement of DNNs greatly and allows training a network

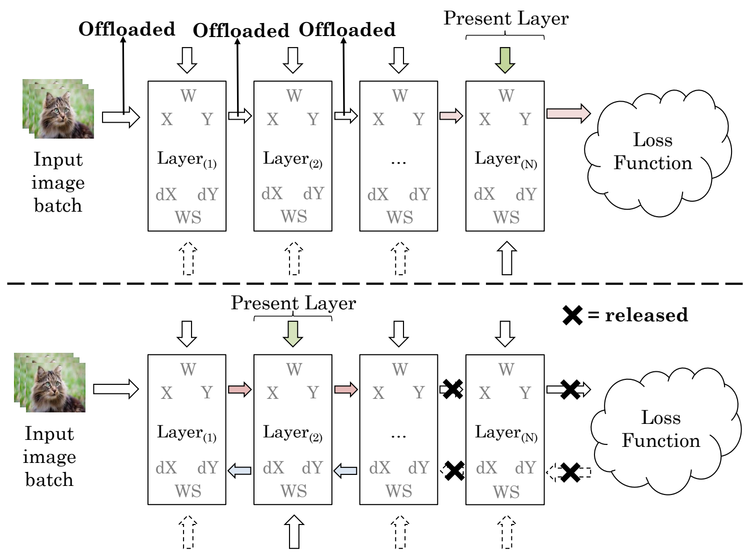


Fig. 19. (a) Offloading previous layer's input fmaps to CPU memory in FWP [24] (b) releasing input fmaps of previous layers in BWP since they will not be reused

TABLE 15
Policies for choosing the layer to offload [24]

Policy	Working
Static 1	This is a memory-efficient scheme which offloads and releases fmaps of all feature-extraction layers
Static 2	This is a performance-efficient policy that offloads fmaps of CONV layers and leaves fmaps of pooling/activation layers in the GPU memory. Since CONV layers account for upwards of 70% latency in FWP/BWP phase of DNNs, the offloading/prefetching latency can be easily hidden.
Dynamic	Existing machine-learning frameworks use a profiling phase where the best CONV algorithm for each layer is found using a cuDNN API. The dynamic policy enhances this profiling phase with extra passes for finding the best layers for offloading and best CONV algorithm such that the performance is optimized and the total memory requirements can be met.

with higher memory than the memory capacity of the GPU. Also, it incurs only small performance loss compared to an oracle GPU having sufficient memory to store the whole DNN.

Chen et al. [26] note that the technique proposed by Rhu et al. [24] offloads data of either all the layers or all the CONV layers, and thus, does not choose the layers wisely. They propose a technique which performs data-transfer scheduling and CONV-algorithm selection intelligently. While decreasing the batch size reduces memory consumption, it may affect accuracy. To avoid losing accuracy, their technique divides a batch into several sub-batches. Due to this, training of a batch is completed in several passes, and the gradients are aggregated from all the sub-batches.

The memory requirement is minimum (M_{min}) when the sub-batch size is 1, and the CONV-algorithm with least memory requirement is chosen. This CONV-algorithm is implicit GEMM since it does not require any workspace. They present a technique for DNN training when the available memory is larger than M_{min}. Since the memory consumption is increased on both increasing the sub-batch size and using more performance-efficient CONV-algorithm, they find an optimal value of them by performing iterative profiling. During profiling, the execution time of a task and the transfer latency of its data are found.

The technique of Rhu et al. overlaps one data-transfer operation with one computation, and hence, incurs larger latency and synchronization overheads. To avoid these, their technique overlaps one data-transfer operation with several computations or vice versa. Their technique decides about the layers to be offloaded. However, if the GPU memory becomes fragmented, then a suitable offloading scheme cannot be found. In such a case, their technique offloads all data and re-loads the required data. Their technique schedules offload/prefetch operations and selects a CONV algorithm for each layer with a view to minimize the overall execution time without exceeding the memory budget. For every fast CONV-algorithm, cost and benefit analysis is done, where the cost is the latency of offloading/prefetching, and the benefit is the latency-reduction due to faster CONV. The CONV algorithm, with the largest value of benefit minus cost is selected. For the same memory budget, their technique incurs smaller performance loss than the technique of Rhu et al. and also works well in the case of training on multiple GPUs.

Wang et al. [28] propose a memory management runtime for enabling training of non-linear DNNs on GPUs whose memory requirement far exceed the memory capacity of GPU. Figure 20 shows examples of non-linear connections. Let forward, and backward memory usage of layers i be M_i^f and M_i^b, respectively. Let peak memory requirement of the network be M_{peak}. The i^{th} tensor is shown as E_i and total number of layers is L. For the baseline network-wide memory-allocation approach, $M_{peak} = \sum_1^L M_i^f + \sum_1^L M_i^b$. Clearly, the theoretical minimum memory requirement of the CNN is max(M_i) for $i \in [1, L]$.

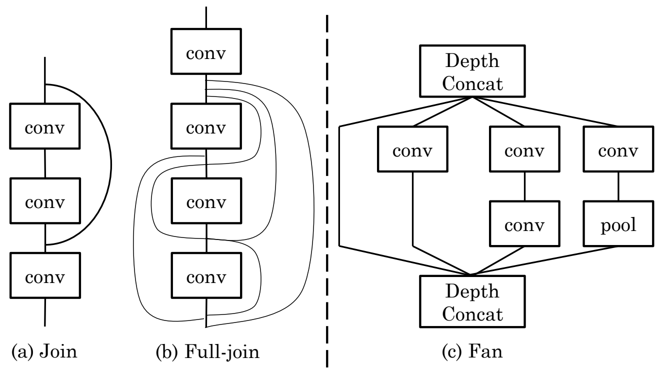


Fig. 20. Non-linear connections in (a) ResNet (b) DenseNet and (c) Inception v4 [28]

In a non-linear DNN, they first identify the execution-order of different layers by taking into account the interconnection (dependencies) between them. Table 16 shows the working of the first technique which reduces M_{peak} to $\sum_1^L M_i^f + M_L^b$, thus achieving 50% saving in memory consumption. The challenge in this technique is that since during training, millions of iterations of FWP/BWP phases are executed, allocating/deallocating memory using `cudaMalloc/cudaFree` incurs large overhead. To avoid this overhead, they allocate a large heap-based memory in the beginning and then, merely acquire/release memory from this pool, which is equivalent to allocation/deallocation, respectively.

TABLE 16
Summary of the techniques of Wang et al. [28]

No.	Technique	Working	M_{peak}
	Baseline		$\sum_1^L M_i^f + \sum_1^L M_i^b$
1	Deallocation	While doing BWP phase for layer-k, it deallocates M_i^f and M_i^b where $k+1 \leq i \leq L$. This is because these are not required for the execution of successive layers.	$\sum_1^L M_i^f + M_L^b$
2	Offloading	It offloads data of selected layers from GPU memory to CPU memory and later prefetches them. Due to the insignificant memory consumption of FC, softmax, and dropout layers, they are not offloaded. Further, LRN, BN, activation, and pooling layers together account for 50% of the memory needs, but their computations are a much smaller fraction of the total computations. This prohibits effective overlapping of computing and data-transfer operations. Hence, it offloads data of CONV layers only. Specifically, forward outputs of CONV layers are transferred to pre-allocated "pinned CPU memory" and then, the corresponding GPU memory is released.	$\sum_1^L (M_i^f \mid i \notin$ offloaded$) + M_L^b$
3	Release and recompute	Since LRN, BN, pooling and activation layers account for 50% of memory but a much lower fraction of computations, this technique frees them after FWP phase and recomputes them during BWP phase. This is done only if the highest memory needs of any layer cannot be met by the present memory budget. This helps in exercising a balance between performance and memory efficiency.	$max(M_i)$
4	Faster CONV	If after applying the above three techniques, some memory is still left, a faster implementation of CONV is selected as long as its memory requirement can be met.	

The working of the second technique is shown in Table 16. Both offloading and prefetching operations are overlapped with useful computations. Since data-transfer over slow PCIe bus may create a bottleneck, they use a "tensor cache" which stores tensors on GPU DRAM to reuse them maximally. Due to the "head-to-tail" and then, "tail-to-head" execution pattern of BWP phase, most recently generated data are reused first and hence, the tensor-cache is managed using "least-recently used" replacement policy. On using these, data-transfers on using offloading/prefetching happen only when the GPU memory is not sufficient. Table 16 also shows the working of third and fourth technique. For the same memory budget, their techniques allow training with much larger batch size than that allowed by Torch, Caffe, TensorFlow, and MXNet. Using their techniques, ResNet2500 CNN with 10,000 basic layers can also be trained on a GPU with 12GB memory.

Jin et al. [32] present a technique for enabling use of higher batch size in CNN training, which reduces data-communication during distributed training. Figure 21 presents an overview of their technique. In phase one, their technique divides the input image into multiple patches and processes them independently before entering subsequent stages of CNN. This is illustrated in Figure 22. The splitting is applied in R CONV layers. With increasing R and decreasing patch size, the overall model accuracy reduces and hence, the values of R and patch size are chosen carefully. In phase two, topological sorting is performed on compute nodes of the dataflow graph for serializing the computations. Analogously, the computation graph for BWP is also serialized.

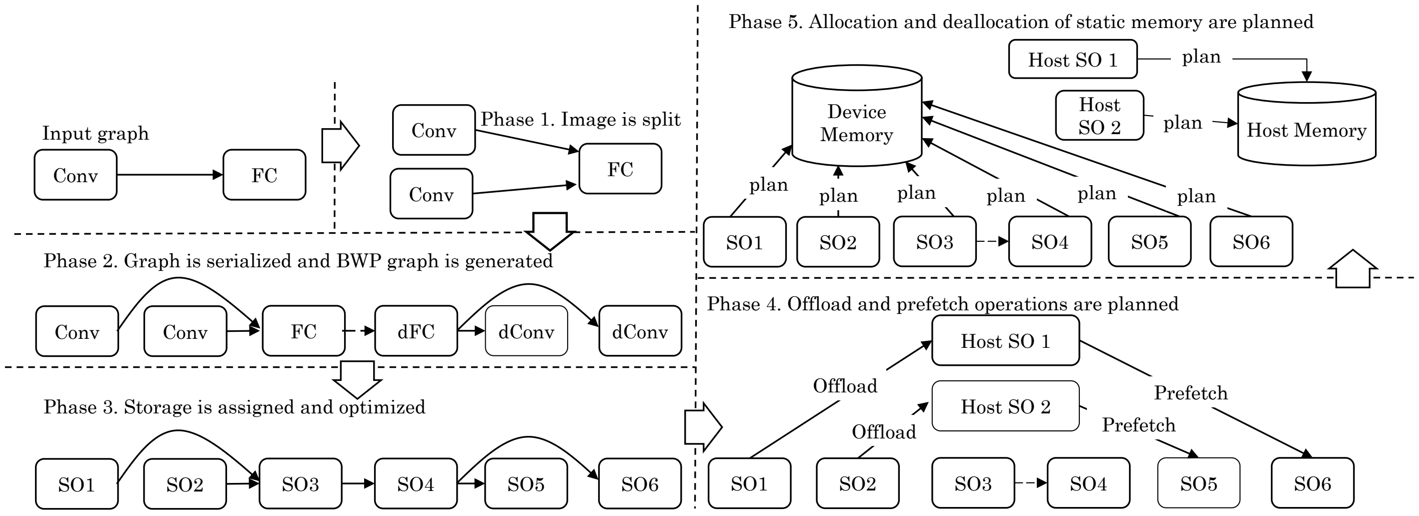

Input graph
Conv
FC
Phase 1. Image is split
Conv
Conv
FC
Phase 2. Graph is serialized and BWP graph is generated
Conv
Conv
FC
dFC
dConv
dConv
Phase 3. Storage is assigned and optimized
SO1
SO2
SO3
SO4
SO5
SO6
Phase 4. Offload and prefetch operations are planned
Host SO 1
Host SO 2
Offload
Offload
Prefetch
Prefetch
Phase 5. Allocation and deallocation of static memory are planned
Device Memory
Host Memory
Host SO 1
Host SO 2
plan

Fig. 21. Memory management approach in the technique of Jin et al. [32]

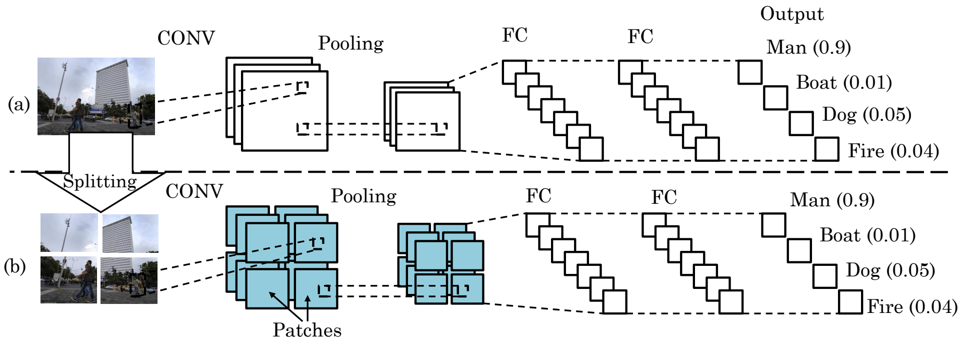


Fig. 22. (a) Traditional CNN (b) Split CNN having four patches [32]

In phase three, every tensor in the computation graph is mapped to a "storage object" (SO) which refers to a contiguous memory region where one or more tensors are used. For every SO, a "reference counter" is maintained. Since the input of "rectified linear unit" (ReLU) layer is not required during BWP phase of this layer, if no other tensor references the SO of this layer's input tensor, the input tensor's SO is replaced with the output tensor's SO. Further, on applying chain rule for obtaining back-propagation error, the value of all the error terms is same, and hence, the same SO is used for them.

Notice that there is no need to plan the beginning of offload operation since it happens right after computation in FWP phase. Similarly, since the prefetch operations end right before the computation in BWP phase, no planning is required for the end of prefetch also. Hence, their technique plans only the end of offload operations and beginning of prefetch operations. The planning is done such that communication should not slow-down computation, which happens when the product of estimated execution time and CPU-GPU link (PCIe or NVLink) bandwidth is more than the total data size to offload. Similarly, prefetch starts only when the product of layer execution time and link bandwidth exceeds the data size to prefetch. Their technique increases the trainable batch size of ResNet to 2X and that of VGG to 6X, with an only small reduction in throughput. Usually, only a few layers prohibit trainability of CNN due to their high memory requirements. Image-splitting in their technique reduces the memory requirements of these layers and also allows different patches to reuse the same workspace. By virtue of this, their technique improves the trainability of CNNs.

"High-bandwidth memory" (HBM) is a DRAM variant which is designed with multiple vertical memory dies. The memory bus of HBM is much wider (4096-bit) than that of GDDR5 memory (384-bit). Due to this, HBM provides high bandwidth (up to 512 GB/s) despite working at a low frequency (1GHz). AMD Radeon Fury GPU has four stacks of memory chips with a total in-package global memory capacity of 4GB. Thus, compared to other GPUs such as NVIDIA K80 with 24GB memory, the capacity of HBM is much smaller due to thermal and technological constraints.

Zhu et al. [29] note that on using minibatch-size of 256, training of AlexNet becomes very slow on the HBM-enabled GPU since the fmaps cannot be stored in HBM due to its limited capacity. When GPU memory becomes insufficient, OpenCL runtime allocates memory buffers in the host memory. Afterward, read/write operations are performed to this buffer over PCIe, which provides much lower bandwidth than HBM.

They present a technique to mitigate this issue. Due to the layer-wise training, in BWP phase, the output of only one layer is required when the next layer is being processed. Hence, their technique offloads data of layers that are required in the future to CPU memory and prefetches their data before they are required. Since the training of a CONV layer takes larger time than the data-transfer between CPU and GPU, the data-transfer latency is easily hidden. Data-transfer is handled by a DMA engine and the double-buffering scheme is used. Their technique works using two command queues, which are shown in Figure 23. At the time of training layer k, the data of layer $k-1$ is copied from GPU to CPU memory and that of layer $k+1$ is copied from CPU to GPU memory. When the data of layer $k+1$ arrives, the second queue notifies the first queue to begin training layer $k+1$. When layer $k+1$ has been processed, the first queue notifies the second queue to begin data-transfer. In this way, their technique reduces memory requirement to the size of the double-buffer. They apply their technique on all CONV layers of AlexNet. They evaluate their technique on AMD Fury X GPU and show that it leads to higher performance than that achieved by NVIDIA K40 GPU, which has 12GB memory.

Awan et al. [40] propose a technique to address memory capacity limitations in DNN training. They note that if a batch of training samples requires 1GB memory, one needs to allocate 1GB of CPU memory as the "staging area" and 1GB of GPU memory for the training itself. Further, host-to-device copies of 1GB of data need to be handled. Also, since `cudaMalloc` and `cudaFree` calls are blocking, their use during training for saving GPU memory leads to further overhead.

In baseline design, communication happens between file-system $\Longleftrightarrow$ host-memory, between host $\Longleftrightarrow$ device memory and between device $\Longleftrightarrow$ device memory. They propose primitives for communication between file-system $\Longleftrightarrow$ unified-memory and within unified-memory itself. Figure 24 contrasts baseline design with their proposed approach. They further design an "interception library" which redefines the behavior of CUDA memory allocation and management calls.

They use `cudaMallocManaged` to allocate a unified memory buffer which can be accessed by both the file system and CUDA kernels that need to process this data. This saves GPU memory and allows overlapping part of the computations with driver-managed copies. CUDA kernels work directly on the "managed buffers". Since the size of trained DNN is small, transferring it to file-system at the end of training does not incur overhead. Still, the use of `cudaCpuDeviceId` hint for setting the preferred location of this buffer through `cudaMemAdvise` improves the performance of some large models.

They further note that a naive "managed-to-managed" (M2M) transfer takes higher latency than a "device-to-device" (D2D) transfer on the same GPU. For `cudaMemcpy`, prefetching the destination

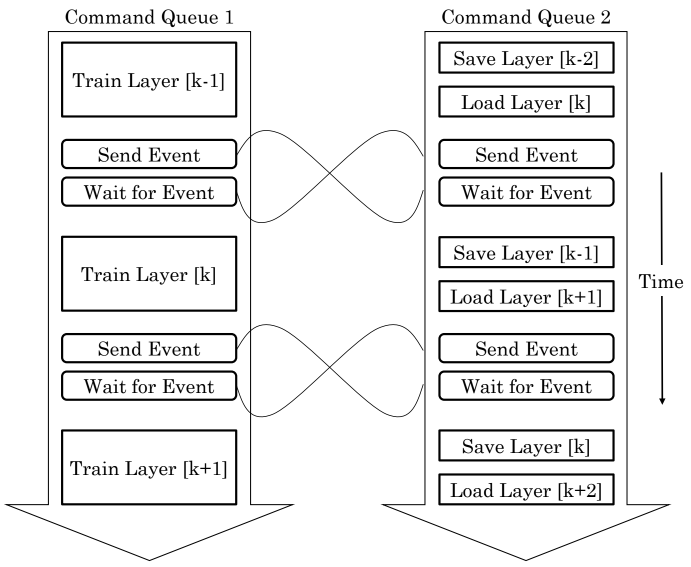


Fig. 23. Working of two queues in the technique of Zhu et al. [29]. Here "training" a layer means executing its FWP or BWP phase.

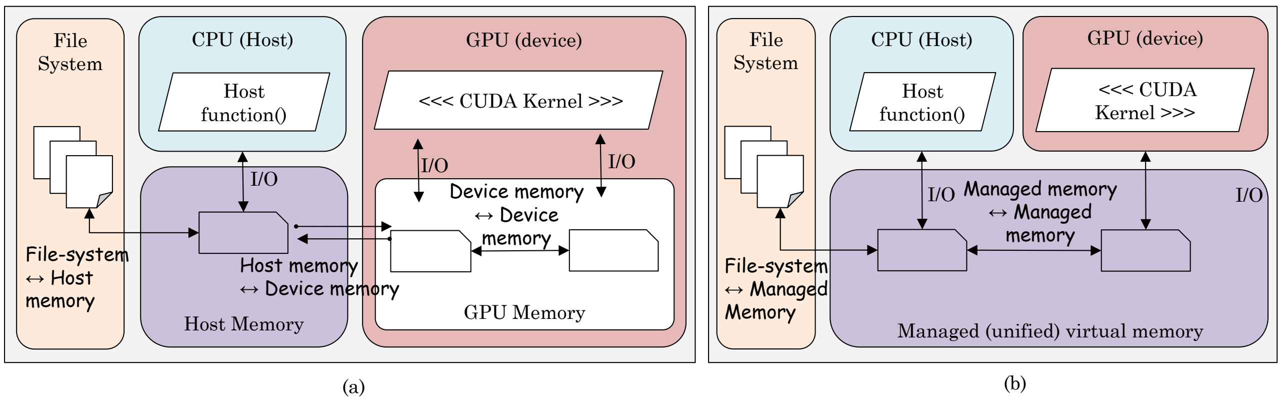


Fig. 24. (a) Traditional primitives for CPU and GPU memory allocation and associated data-transfer (b) primitives proposed by Awan et al. [40] for unified memory allocation

buffer offers higher performance for an M2M transfer but prefetching the source buffer has no impact on performance. This is because the source buffer results from a previous kernel and hence may be already residing in the GPU memory. Further, undue use of `cudaMemPrefetch` calls can create overhead. Since the source buffer of one layer becomes the destination buffer of previous layer during FWP phase, prefetching the destination buffer of copy operation improves performance by decreasing GPU page faults. To improve performance of M2M copy operations, they propose marking the source buffer as "consumed" and offloading it at the time of low GPU memory. For this, `cudaMemAdviseSet-PreferredLocation` is used and `cudaCpuDeviceId` is passed as the "device" option to the `cudaMemAdvise` call on the "source buffer". Since buffers offloaded during the FWP phase are required during the BWP phase, these buffers are prefetched for any M2M copy operations.

Use of unified (managed) memory improves productivity by virtue of offering a simplified programming abstraction. It also reduces redundant code since separate versions of each function for CPU and GPU are not required. For CNN training workloads whose memory requirements exceed GPU memory capacity, their technique provides a large improvement in performance. The limitation of their technique is that the features of CUDA 8 and 9 exploited by them such as unified memory, prefetching and automatic page-migration work only with Volta and Pascal GPUs.

4 Optimizing DL on distributed GPUs

Data/model-parallelism and pipelining: There are three broad approaches for performing distributed training of DNNs [70].

1) *Data-parallelism:* Here, same DNN model runs on each system but is fed with different portions of the training data. The challenge in data-parallel training is that it requires the weights and activations to fit inside the limited GPU memory.
2) *Model-parallelism:* Here, the DNN computations are divided across multiple systems, and same training data is fed to all the systems. This allows training of very large models since the entire CNN is not stored in one system. However, it leads to extra communication after each layer.
3) *Pipelining:* In the pipelining scheme, one or more consecutive layers of a CNN form a chunk. The layers in a chunk are executed on one system, and thus, different systems compute the CNN in a pipelined manner.

Gradient communication topologies for parameter synchronization: In data-parallel DNN training, synchronization of model parameters can be implemented using two approaches.

1) *Parameter server (PS):* PS approach, shown in Figure 25(a), uses the master-slave paradigm where at least one server node manages the parameter updates centrally. In each iteration, every worker computes its gradients and sends them to the PS node for accumulation and model update. Then, the PS node sends updates parameters to every worker. PS facilitates the use of flexible synchronization schemes and allows training very-large models. However, when the number of workers (clients) become large, BW of PS becomes a bottleneck.

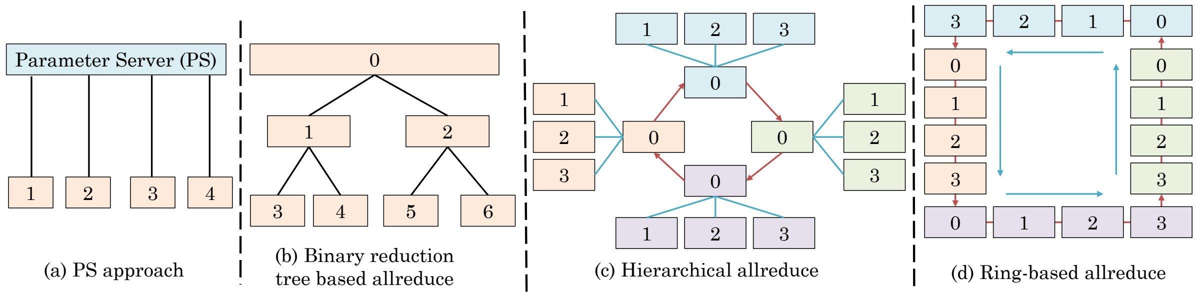


Fig. 25. (a) Parameter server (b) binary tree based allreduce (c) hierarchical allreduce (d) ring-based allreduce

2) *Allreduce approach:* Allreduce is an example of "collective communication operation". Here, all workers communicate between each other using an allreduce operation for exchanging the local gradients. This provides accumulated gradients to each worker which are used for updating the parameters locally. By virtue of using peer-to-peer communication, allreduce-based approach distributes the communication cost between all the workers. Allreduce can be implemented in different ways, some of which are shown in Figure 25(b)-(d). The challenge with the allreduce approach is that it requires physical connections between many more pairs of workers than what the PS approach requires.

Synchronization approaches: In data-parallel training, multiple strategies can be used for aggregating the results of different workers. At the two extremes are the following modes:

1) *Synchronous mode:* Here, PS waits till it has received the gradients from all the workers. Then, gradients are applied to the current weights, and the updated weights are sent to all the worker nodes. By virtue of using accurate gradient estimates, this approach leads to fast convergence. However, this approach may become bottlenecked by the slowest worker, and this problem becomes worse in case of network congestion.
2) *Asynchronous mode:* Here, PS updates the parameters each time it receives the gradients from a worker. This improves the throughput compared to the synchronous mode. However, the parameters used by every worker may be different, which degrades the quality of gradient estimates [71] and increases the number of iterations required for convergence.

Table 17 shows the projects that optimize GPU-based distributed DNN training. The challenge in distributed training is that especially for large-sized models, the communication latency far exceeds the computation latency and hence, effective approaches for hiding/minimizing communication overheads are essential. The strategies used for these are shown in Table 17. For example, "double buffering scheme" uses two buffers such that one buffer is filled with data, and the data of the second buffer is used for computation. In every phase, the role of buffers is reversed, which helps in hiding the communication latency.

TABLE 17
A classification of works that perform DNN training using distributed GPUs

Category	References
Distributed DNN training	[7, 21, 22, 26, 37–39, 41, 42, 44–49, 51, 54, 57]
Data-parallelism or model-parallelism	
Data-parallelism	[7, 25, 26, 37–39, 42, 44–49, 51, 53, 54, 56, 57]
Model parallelism	[44, 54, 72]
Pipelining	[44]
Use of parameter server or allreduce	
Parameter server	[7, 44, 46, 49, 51, 53, 57]
Allreduce	[37, 42, 45, 47, 48, 56]
Mitigating data-communication overheads or BW bottlenecks	
Mixed/low-precision	[13, 21, 37, 42, 43]
Double buffering	[5, 7, 20, 59]
Performing synchronization . . .	when the total size of data exceeds a threshold [37, 43], after multiple layers [39], with stale parameters [30, 49]
Avoiding PS from becoming a bottleneck	Improving PS access efficiency by using pre-built indices, which avoids the need of accessing arbitrary parameter values on the GPU PS [7], allowing communication between clients to remove bottleneck due to centralized PS [49]
Reducing amount of data-transfer	Transferring only required data [38, 47], prioritizing communication of items that are most important for convergence [49], storing parameter cache in GPU memory [7], using sufficient-factor communication instead of transmitting weight gradient tensor [49], by intelligent partitioning of CNN layers which are mapped to different machines [44, 51], Storing weights persistently [8, 35, 46]
Others	Choosing CNNs with lower number of parameters (e.g., NiN and GoogleNet) than others (e.g., AlexNet) [48], processing multiple blocking send/receive operations by a tree-reduction operation [46], reducing GPU memory consumption by performing gradient aggregation on the host [39], packing smaller-width values to improve BW utilization [13], using GPUs located in the same data-center and not at different geographical locations [30], use of high-BW stacked memory [29]

In this section, we discuss works that perform parameter synchronization using PS approach (Section 4.1). A variant of this is where gradient-accumulation is performed on the host and thus, no GPU acts as the PS (Section 4.2). We then discuss works that use allreduce approach (Section 4.3). Then, approaches for using mixed-precision training (Section 4.4), optimizing pipeline-parallel training (Section 4.5) and scheduling techniques for DL jobs (Section 4.6) are discussed. Finally, a technique for reducing financial cost of training (Section 4.7) and comparison of distributed training on GPUs with that on CPU/Xeon-Phi (Section 4.8) are summarized.

Several of these works overlap gradient-accumulation of upper layer with BWP of a lower layer [37, 39, 46, 47, 49]. In CNNs, initial (CONV) layers have much lower parameters than the last (FC) layers. As BWP computations progress from "tail-to-head" direction, gradient-communication and aggregation of FC layers get over by the end of BWP phase even if it takes a large amount of time. For CONV layers, this takes much less time and hence, happens quickly after BWP phase. Hence, the layer-wise overlap scheme is especially effective for CNNs.

4.1 Using parameter server approach

Cui et al. [7] present optimizations to PS design for enabling training on GPUs attached to different servers. In the baseline design, parameter cache is stored in the CPU. They propose storing it in GPU memory, which allows the PS to overlap data-movement with GPU computations. Also, read/update functions can happen inside GPU memory itself, and updating of the parameter cache can be parallelized.

Due to the SIMD nature of GPU, accessing arbitrary parameter values is inefficient. Since training happens in iterations, they propose using pre-built indices for the whole batch which allows parallel gather/update operations on the parameters referenced in a batch. This improves the efficiency of accessing PS. Furthermore, currently-unused data are moved from GPU memory to CPU memory. Due to the iterative nature of training, the same parameter data are accessed in each iteration and based on this, data transfer can be done proactively in the background to avoid its performance impact. Lastly, the data that will not be reused, such as intermediate results, is deallocated. Their technique reduces the stalls due to data-movement and synchronization.

They also observe that although performing parameter updates asynchronously avoids synchronization overheads, it increases the time to reach convergence due to lower quality of training. Their technique achieves more than 80% strong scaling efficiency on 16 GPUs. Also, the training throughput of their technique with four GPUs is higher than that achieved using 108 CPUs.

Zhang et al. [49] present techniques for scaling DNN training on networks with low bandwidth, e.g., Ethernet connection. They note that in the client-server architecture, communication happens only between the client and the server. They add a hierarchy in every worker node, which allows multiple client threads to co-exist inside one worker machine. This allows using both CPU and GPU workers by mapping every worker thread to a CPU or a GPU. Further, communication is allowed both between "client and server" and between two clients. This allows using dedicated data-transfer schemes for synchronizing the parameters. It also avoids the situation where PS becomes a bottleneck in communication. Figure 26 shows an overview of their technique.

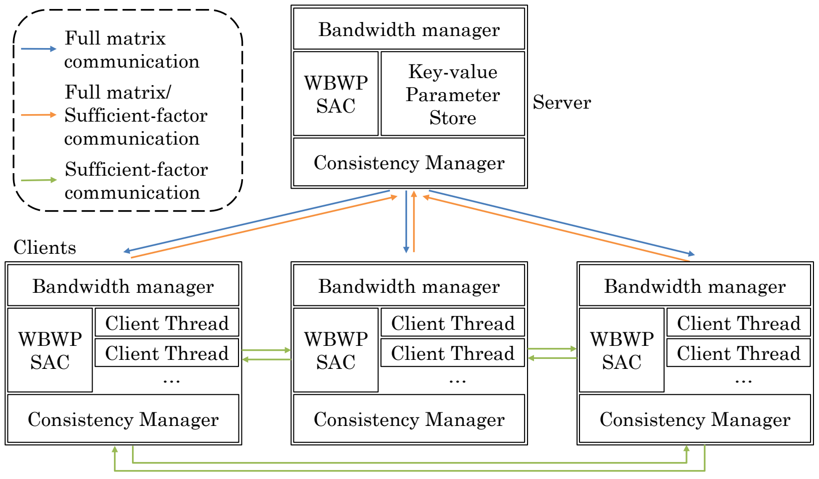


Fig. 26. Overall architecture of technique of Zhang et al. [49] (WBWP= Wait-free BWP, SAC= structure-aware communication)

Their technique overlaps communication of gradients of upper layers with BWP of lower layers and this is referred to as "wait-free BWP" in Figure 26. They discuss the "sufficient factor communication" approach, where instead of directly sending weight gradient tensor to the master node, it is decoupled in two vectors. These vectors are broadcast to other workers, and others' vectors are received from them. Then, the weight gradient tensor is reconstructed from them, and updates are applied locally on each worker. Compared to transmitting the whole matrix, this approach reduces the communication overhead significantly. Also, since computation is much cheaper than communication, the extra computations for finding significant factor is negligible. However, the cost of "sufficient-factor broadcasting" increases quadratically with the rising number of nodes due to use of the peer-to-peer data-exchange approach. Hence, they use an adaptive communication scheme, which works as follows: (a) Since parameters of CONV layers are sparse, parameters are communicated directly via the centralized PS (b) Since the parameters of FC layers are dense and have "low-rank property", a choice is made between PS and "sufficient-factor broadcasting" depending on the batch size and the number of workers.

For data-parallel CNNs, iterative-convergence based algorithms can converge even when their parameters see synchronization delays, as long as the delays are bounded. Based on this "stale synchronous parallel" (SSP) consistency model is used which saves communication costs by allowing updates of parameters with bounded staleness. They also use a bandwidth manager which sends dirty model parameters and model updates as soon as possible without exceeding the network BW budget and allots network BW based on the contribution of a message to the convergence. Their technique reduces the time and number of machines required for achieving the same level of accuracy as previous techniques.

4.2 Using the host for performing gradient accumulation

There are five steps in data-parallel training over multiple GPUs:

S1: Every GPU reads a minibatch and executes FWP phase.
S2: Every GPU executes BWP phase for computing the gradients based on its learnable parameters
S3: The gradients of the other GPUs are collected at the master GPU (GPU 0), and their average value is computed.
S4: Computed gradients of master GPU are used for updating its "learnable parameters".
S5: Master GPU broadcasts its learnable parameters, and the remaining GPUs update their parameters to these values.

Le et al. [39] present a technique for utilizing the host to accelerate training on the GPUs. They seek to alleviate the impact of communication steps (S3 and S5), which bottleneck the scalability of parallel training. They note that gradients of a layer do not change after computation during BWP phase. Hence, gradient collection need not be postponed until the end of the BWP phase. Further, gradient collection can be done with the help of the host CPU. Based on these, they overlap S3 (gradient collection) and S5 (broadcasting of parameters) with S2 (BWP).

For this, they use 3 GPU streams. Computation of loss function in FWP and gradients in BWP and updating of parameters happen in the "default stream". The "deviceToHost stream" is utilized for sending local gradients to the host and then issuing a function on the host for collecting the gradients. The "hostToDevice stream" is utilized for broadcasting the global gradients to GPUs. Data-transfers happen sequentially since at most one transfer in one direction can happen at any time. Their technique works in 3 major steps:

SP1: Every GPU reads a minibatch and executes FWP phase.
SP2: Every GPU executes BWP phase.
SP3: Every GPU updates its parameters.

Gradient collection during SP2 happens on the host, and after accumulation, gradients are broadcast to all the GPUs. Thus, on completion of BWP, each GPU has the same gradients and updates its parameters concurrently with other GPUs. Hence, their technique does not require any PS. When the CPU is aggregating the gradients, GPU goes on to calculate the gradients of subsequent layers.

They further note that in the baseline scheme, communication with host happens after every layer using a synchronization barrier. This, however, incurs large overhead for deep CNNs, such as ResNet-152 which has 152 layers. To mitigate this issue, they propose performing synchronization after multiple layers, called a group. The limitation of this approach is that gradient collection for all the layers in a group happens only when the last layer in the group completes its BWP phase. This increases the overhead of gradient aggregation.

The number of layers in different groups can be different, although, in practice, they use the same layer-count in each group to shrink the search-space. This layer-count is found heuristically by running a few initial training iterations and choosing a value that minimizes the execution time of the BWP phase. Compared to baseline Caffe, their technique improves the training speed of AlexNet, GoogleNet, VGG-16, and ResNet-152 and achieves weak-scaling efficiency above 90% on four GPUs. Further, performing gradient aggregation on the host reduces memory consumed on GPU, which allows increasing the batch size.

4.3 Using allreduce approach

Iandola et al. [48] accelerate DNN training on a cluster of GPUs. They perform experiments using 32 to 128 GPUs. With 32 nodes, the interconnect speed approaches that obtained with connecting all nodes in a portion to a single Infiniband-class switch. They choose NiN and GoogleNet for training since these CNNs have a much lower number of parameters than other networks such as AlexNet. During training, no communication is required during FWP. During BWP, every GPU computes a sum of the weight gradients for its mini-batch, and finally, the weight gradients of different GPUs are added. Accumulation of local gradients can happen using a PS or an allreduce operation. The allreduce operation is performed using a binomial reduction tree (refer Figure 25(b)) where the communication latency increases as the log of the number of GPUs. By comparison, with PS, the communication latency increases linearly with the number of GPUs due to serialization of communication with PS. Thus, reduction trees incur lower latency of communication than PS. Compared to single-GPU, on 128 GPUs, their technique provides 47× and 39× speedup while training GoogleNet and NiN, respectively.

Awan et al. [47] present optimizations to Caffe to scale it to clusters with multiple GPUs. For avoiding unimportant data-transfer between CPU and GPU, they transfer only essential data-items across the processes by utilizing existing GPU-based buffers. For optimizing file access operations, each process runs a thread and maintains separate queues. These threads can concurrently read data from a parallel file system such as Lustre. This is more efficient than transferring images with MPI-level data-exchange. The parallel reader approach can be utilized along with ImageDataLayer and "lightning memory-mapped database" (LMDB) of Caffe, which scales well till 160 GPUs. Further, CUDA-aware MPI broadcast is used in data-propagation phase, and MPI reduce is used for the gradient aggregation.

To increase the overlap between computation and communication, the weights and gradient tensors are transmitted on-demand instead of transmitting all the tensors in a single operation. Then, "non-blocking collective operations" of MPI-3 are used for overlapping data-transfer with compute operations. Specifically, all the non-blocking broadcast operations are performed at the beginning and the `Wait` operation of l^{th} broadcast operation is placed right before l^{th} FWP of a layer that requires those data. This is shown in Figure 27(b) and is contrasted from the naive approach shown in Figure 27(a).

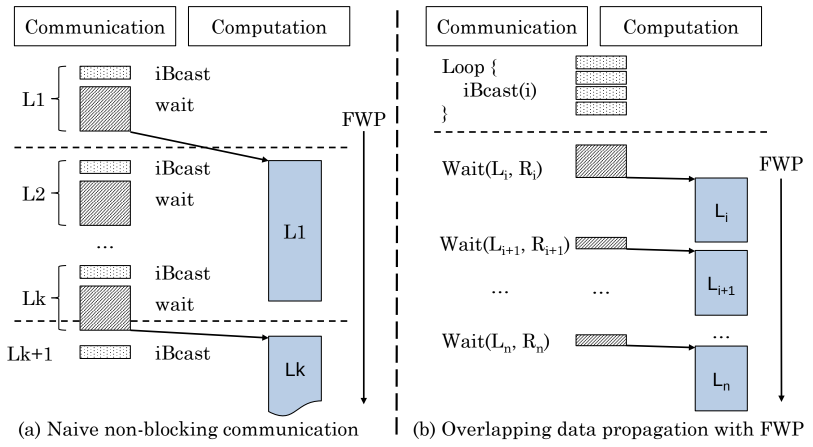


Fig. 27. (a) Naive non-blocking communication for data-propagation (b) Overlapped data-propagation with FWP [47]

They further note that inter/intra-node hierarchical collective communication is not efficient in case of GPU-based communication since a node has at most two to four GPUs. To enable efficient reduce operation, they propose a hierarchical design where the "lower-level communicator" may be spread over multiple nodes. Figure 28 shows a hierarchical communicator where the lower-level communicator spreads over two nodes. The algorithms for communicators at lower-level and upper-level can be selected at runtime. For example, when process-count is large (e.g., $>$ 64), binomial tree-based reduce is used in "upper-level communicator," and chunked-chain scheme is used in the "lower-level communicator". The chunked-chain scheme works as follows: The last process of the chain splits the buffer into multiple chunks and forwards the chunk to its left-side process. This process performs a reduction of the chunk with its own chunk and sends it to its left-side process. Continuing in this manner, the chunk arrives at the root. Thus, the chunked chain approach seeks to overlap communication and reduction of consecutive chunks in a large buffer. Their optimized Caffe implementation offer 133× and 2.6× improvement over OpenMPI and MVAPICH2 (respectively) on 160 GPUs.

You et al. [46] mitigate scaling limitations of "elastic averaging stochastic gradient descent" on a cluster of GPU. They find that communication of DNN parameters between CPU-GPU (i.e., CPU sending global weight to GPUs and receiving local weights) incurs much higher overhead than communication of a batch

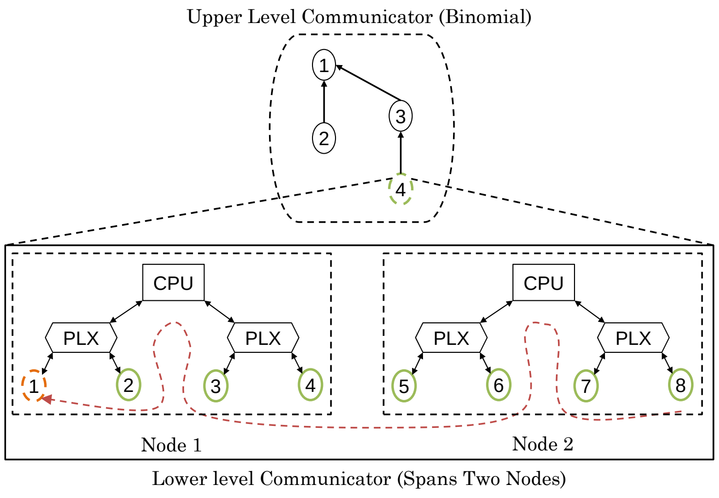

Upper Level Communicator (Binomial)
1
2
3
4
CPU
PLX
PLX
1
2
3
4
Node 1
CPU
PLX
PLX
5
6
7
8
Node 2
Lower level Communicator (Spans Two Nodes)

Fig. 28. A hierarchical communicator with chain-binomial combination [47]

of training data between CPU-GPU in each iteration. This is because the weights of a CNN are much larger than the training data. To reduce the communication overhead, they process multiple blocking send/receive operations by a tree-reduction operation. Further, weights are stored persistently in GPU memory since they are used in every iteration and their total size is less than the memory capacity of GPU. For larger datasets such as ImageNet, the total training data is much larger than the GPU memory capacity, and hence, training data are not stored in GPU memory but are communicated on-demand. Finally, since inter-GPU communication is independent of the FWP/BWP on GPU, they are overlapped. In the baseline design, communication overhead was a bottleneck, but their optimizations remove this bottleneck.

Mikami et al. [42] perform training of ResNet-50 over a cluster of GPUs, where the GPUs are connected in 2D torus topology, which is shown in Figure 29. In distributed DNN training, the use of too small mini-batch size increases the frequency of synchronization between workers. This increases the communication overhead and the time required for completing a fixed number of epochs. Increasing the mini-batch size decreases the variance of gradients by averaging the gradients in mini-batch and thus, estimates the gradients with higher accuracy. It increases the progress rate of the algorithm but reduces the inference accuracy. They employ strategies for using large batch-size without loss of accuracy. Further, they propose a 2D-torus implementation of allreduce operation which works in three steps: (1) horizontal reduce-scatter (2) vertical allreduce and (3) horizontal all-gather. Figure 30 shows these steps for a cluster with 4 GPUs organized in 2D torus topology with a 2x2 grid.

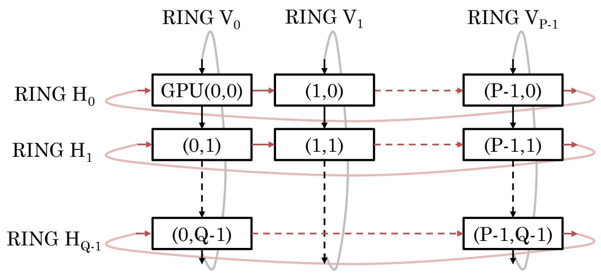


Fig. 29. 2D torus interconnection between the GPUs [42].

If there are P and Q GPUs in the horizontal and vertical dimension respectively and a total of G GPUs, then, the number of inter-GPU operations are 2(P-1) for 2D-torus-based allreduce and 2(G-1) for ring-based allreduce. While the number of inter-GPU operations is the same in both hierarchical and 2D-torus allreduce, the data size of step (2) of the 2D-Torus allreduce is P times less than that of the hierarchical allreduce. On a cluster of 3456 V100 GPUs, their technique completes training of ResNet-50 in 2 minutes and achieves the top-1 accuracy of 75.29%

Zhao et al. [45] present linear pipelining based collective communication operations viz., broadcast,

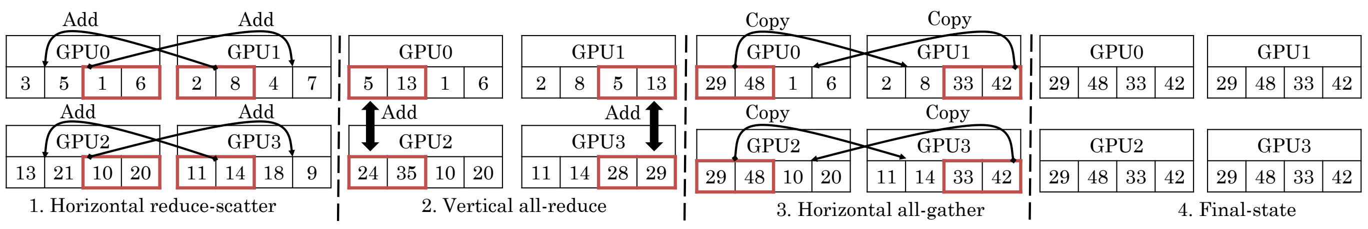


Fig. 30. Allreduce operation in a cluster of 4 GPUs, organized in 2x2 grid in 2D torus topology [42].

reduce, and allreduce. It works by dividing a long message into small chunks. A GPU obtains a chunk from the previous GPU over DMA1 and also sends a chunk to the next one over DMA2. Communication of chunks happens over different links, and once the pipeline reaches a steady state, the whole network is utilized.

Figure 31(a) shows broadcast operation assuming that GPU0 is the source. In step 1, chunk 'P' is copied from GPU0 to GPU1. After this, in every step, GPU1 obtains a chunk from GPU0 and sends a chunk to GPU2, over different links. Figure 31(b) shows the reduce operation assuming that aggregation is performed on GPU2. In step 1, chunk P0 is written to a buffer on GPU1. Then, reduction operation is performed between P0 (received) and P1 to obtain P'. In step 2, GPU1 receives Q0 from GPU0 and reduces to Q', and also sends P' to GPU2 which reduces it to P", and so on.

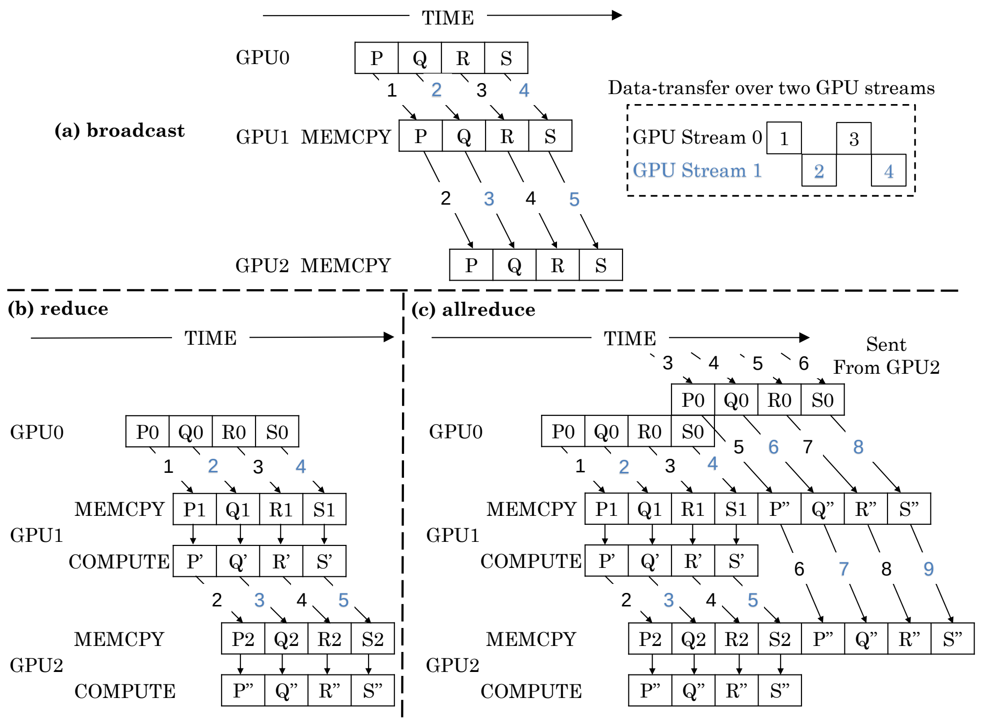


Fig. 31. Working of (a) broadcast , (b) reduce and (c) allreduce on three GPUs [45]

Although allreduce can be implemented as a sequence of reduce and broadcast operations, implementing it as a single instruction is more efficient since it requires filling the pipeline only once. Figure 31(c) shows the workflow of the allreduce operation. First, P0, P1, and P2 are reduced to P" as discussed above. Then, P" is broadcast to GPU1 at step 5 and GPU2 at step 6. Since the outgoing DMA of GPU0 is utilized for sending S0 in step 4, communication of P" can happen in step 5 only. Processing of Q", R", S" happens similarly.

Although PCIe allows full-duplex transfer between two ends, every PCIe end device possesses only one input and output port. Due to this, contention happens when multiple GPUs send to a single GPU. Likewise, every PCIe switch has only input, and the output port and inter-switch communication in the same direction lead to contention on the PCIe bus. It is well-known that the data-transfer delays interrupt pipelining. In minimum-spanning tree based collective communication, transfer between parent to children leads to competition for the same PCIe bus, which interrupts pipelining by creating data-transfer delays. Similarly, in bidirectional exchange, inter-switch data-transfer leads to contention in one direction. By comparison, in their technique, GPUs are connected in a chain, and dataflow happens in one direction. Hence, their technique utilizes network bandwidth optimally and also does not interrupt pipelining. Further, in their technique, the cost of collective operations is independent of the number of GPUs. For training of large CNNs, their technique outperforms "minimum-spanning tree" and "bidirectional based communication".

4.4 Using mixed-precision approach

Jia et al. [43] present techniques for improving the efficiency of distributed training on GPU clusters. "Layer-wise adaptive rate scaling" (LARS) approach uses a local learning rate for every layer which is multiplied with the gradients. It enables the use of larger mini-batch size in DNN training. They note that on using FP16 data with LARS, the learning rate goes beyond the dynamic range of FP16 format. Then, due to the "vanishing gradient" problem, the training process is stalled. Hence, they propose mixed-precision training where the FWP/BWP phases operate on FP16, whereas LARS is performed on FP32 weights and gradients. This enables using large mini-batch size (e.g., 64K) without losing accuracy. Further, to improve accuracy, they remove weight decay on the bias and BN; and add BN layers after Pool5 layer in AlexNet.

The reason for poor scalability of ring-based allreduce is that with M GPUs, it divides the data on every GPU into M portions and performs the reduction in $M-1$ iterations. With increasing M, both the message-size and BW usage efficiency are reduced. Also, the gradient tensor size varies across layers, and the gradient tensors of CONV layers are much smaller than those of FC layers. Their technique packs the gradients in a pool and performs allreduce operation only when the pool size exceeds a threshold (γ). This improves throughput at the cost of latency since tensor-packing prohibits parallelization of gradient accumulation of last few layers and BWP of initial layers. Reducing γ lowers the latency but degrades the efficiency of allreduce operations. They propose "hierarchical allreduce" operation where Q GPUs are grouped and then, all-reduce operation is performed in three steps, as shown in Figure 32: (1) reduce across GPUs of a group and save the interim-results at the master-GPU of the group (2) perform ring-based allreduce between the groups (3) broadcast the final result in every group. By virtue of reducing computation steps, this approach is more efficient in latency-sensitive scenarios, where there are many GPUs, and tensor-size is small. For the cluster of 1024 GPUs, they use $Q = 16$.

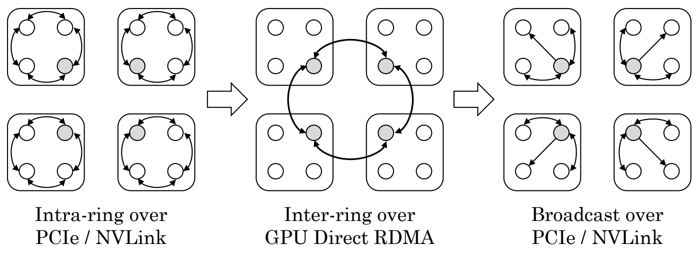


Fig. 32. Three-step allreduce approach for performing gradient accumulation in a distributed system with GPUs [43] (RDMA = remote direct memory access)

They observe that ring-based allreduce performs better for FC layers since they have many more weights, whereas hierarchical allreduce works better for CONV layers which have only a few weights. Hence, they propose hybrid all-reduce, where one of the two allreduce strategies are intelligently selected. They use a cluster with 256 machines, each with eight P40 GPUs connected using PCIe. Each machine has a Mellanox ConnectX-4 100Gbps card. Machines communicate with each other using RDMA. For direct communication between GPUs of different nodes, GPUDirect RDMA (GDR) is used. On 2048 P40 GPUs, their technique completes AlexNet training on ImageNet in 4 minutes with a top-1 accuracy of 58.7%.

Sun et al. [37] present techniques to accelerate distributed training of DNN on GPU clusters. They use two clusters: a cluster with 16 machines, each having 8 Pascal GPUs and a cluster with 64 machines, each having 8 Volta GPUs. The GPUs of a machine connect using PCIe. The machines of a cluster connect using 56Gbps InfiniBand and also have a distributed file system. For distributed training, two frameworks are used: PyTorch-0.4 and their in-house I-system which use Gloo and OpenMPI-3.1

(respectively) as communication backend. Inter-node communication is supported using RDMA in both Gloo and OpenMPI.

Figure 33 shows the traditional approach for distributed training of a DNN. Here, an allreduce operation is performed on the gradients of all the layers. NCCL is a library which provides primitives for multi-GPU collective communication such as allreduce. They find that for allreduce operation, NCCL provides much higher bandwidth than OpenMPI. But even with NCCL, for both PyTorch and I-System, the speedup remains low due to communication latency.

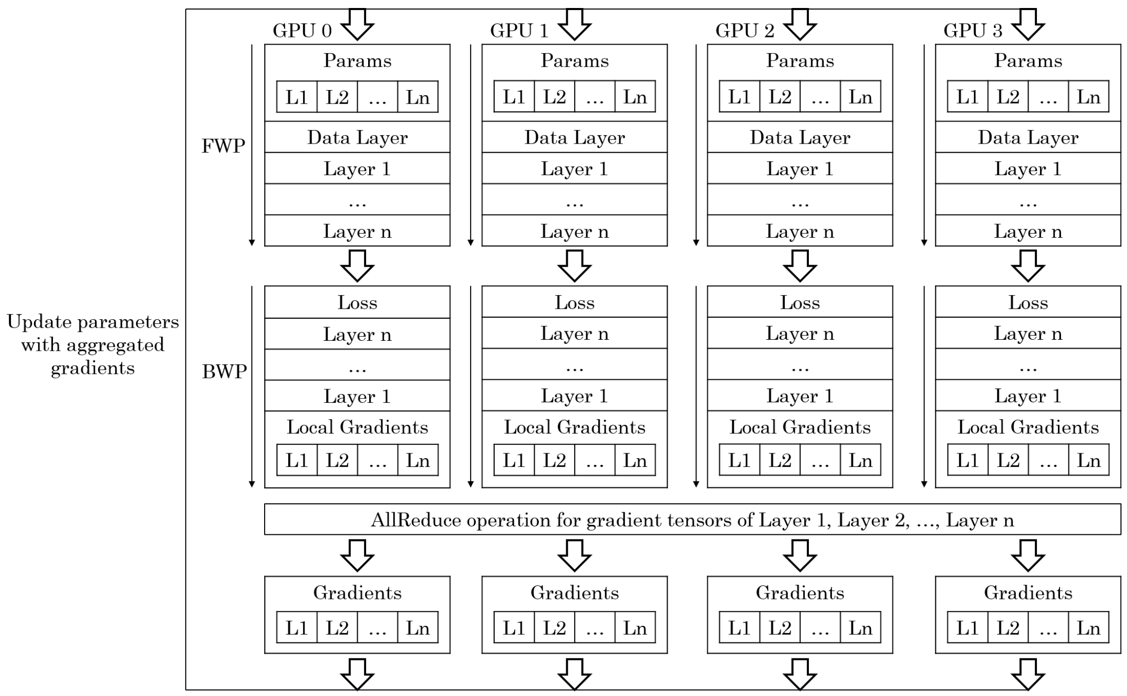

GPU 0
GPU 1
GPU 2
GPU 3
Params
L1
L2
…
Ln
Data Layer
Layer 1
…
Layer n
FWP
Loss
Layer n
…
Layer 1
Local Gradients
BWP
Update parameters with aggregated gradients
AllReduce operation for gradient tensors of Layer 1, Layer 2, …, Layer n
Gradients

Fig. 33. Traditional approach for training DNN on a distributed system [37]

They further explore mixed-precision training, shown in Figure 34, which uses FP32 values for "model update" phase and FP16 parameters and gradients for FWP/BWP phases. NVIDIA's tensor cores perform mixed-precision "matrix multiply and accumulate" computations in a single operation. For FP16 operations, tensor cores can provide between two to eight times speedup over FP32 operations. Although mixed-precision approach reduces data-transfer overheads and allows doubling the batch-size, the speedup is much lower than the ideal speedup. Mixed-precision training also incurs the overhead of conversion between FP16 and FP32.

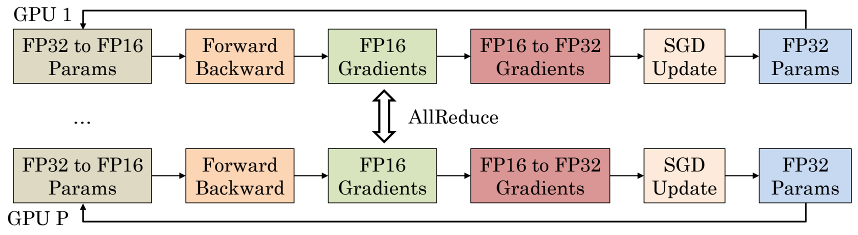


Fig. 34. Mixed-precision training which performs allreduce operation on FP16 gradients [37]

Further, they overlap the transfer of gradient of layer-k with BWP phase of layer-j ($j < k$), as shown in Figure 35. Even on adding this "communication-computation pipelining" optimization, the speedup remains lower than the ideal speedup. This is because on doing allreduce on small-size gradients, the network is not fully utilized. Also, due to their huge number of parameters, CNNs such as AlexNet lead to high network traffic.

To tackle these challenges, they propose a communication backend for the I-system, along with some optimizations. Firstly, it uses "lazy allreduce" which combines several allreduce operations into one

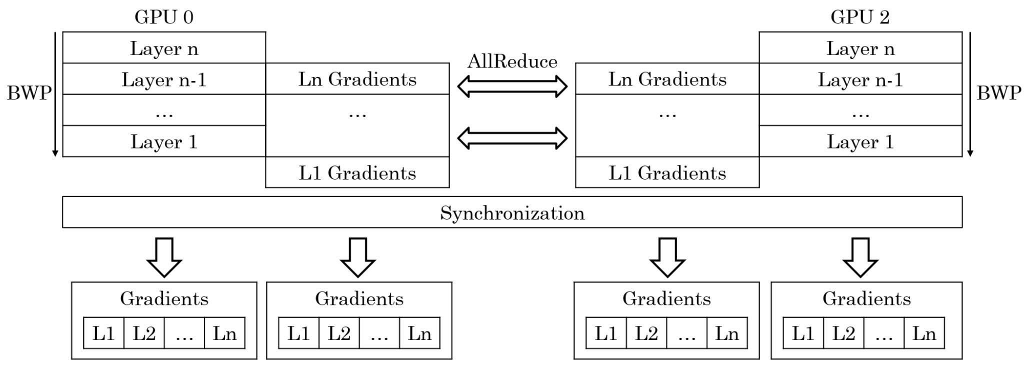


Fig. 35. Layer-wise overlap of communication and computation [37]

operation while incurring only small GPU memory copy overhead. Specifically, as the gradients of a layer are computed in the BWP phase, the baseline design allocates a separate GPU memory space for every gradient. In "lazy-allreduce" scheme, the gradients get placed in a memory pool in the order that they are produced. Then, the allreduce operation is actually performed only when the pool is filled to more than a threshold size. Using lazy-allreduce along with above-mentioned optimizations, they achieve near-ideal performance for the training of ResNet. However, the performance of AlexNet still remains low due to huge network traffic generated by it.

They note that transmitting only those gradients that exceed a threshold can reduce network traffic without harming accuracy. However, it requires "sparse allreduce" which leads to poor network utilization. This is because every GPU transmits different gradients and stores/transmits these values in "index-value" format which is inefficient due to overheads of memory-copy and irregular memory accesses. Hence, this approach does not improve performance.

Instead, they propose "coarse-grained sparse communication" which is illustrated in Figure 36. Here, the gradients in the memory pool are partitioned into chunks having the same number (e.g., 32000) of gradients. After every iteration, a fraction of chunks is chosen as important. Only important chunks are copied to a buffer, and when the buffer size exceeds a threshold, the (lazy) allreduce operation is performed. Each GPU selects the same important chunks which allows using NCCL for accumulating and broadcasting those chunks with high BW usage efficiency. They also use some strategies to avoid losing accuracy. Combining above optimizations, they obtain 410× speedup on 512 Volta GPUs and train AlexNet on ImageNet dataset in just 1.5 minutes, while achieving top-1 accuracy of 58.2%.

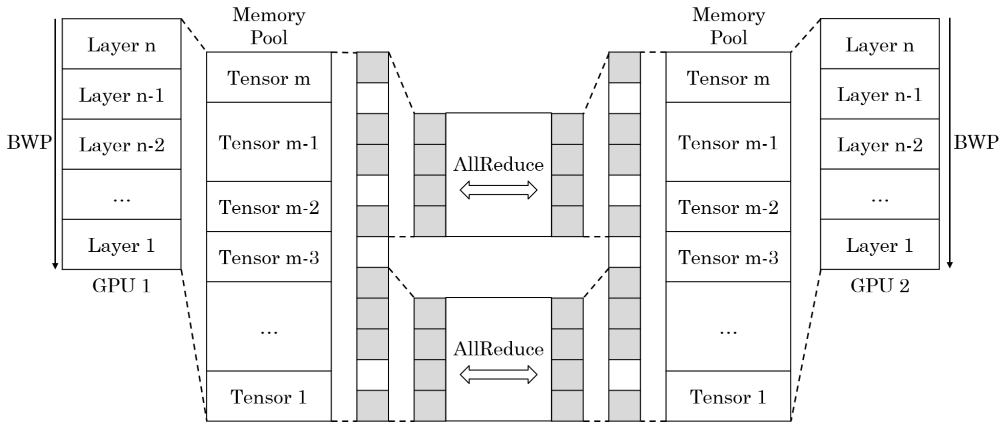


Fig. 36. "Coarse-grained sparse communication" approach [37]

4.5 Using pipeline-parallel training scheme

Harlap et al. [44] propose "pipeline parallel training" for distributed training of DNNs which brings together the best of data-parallel, model-parallel training and pipelining. The layers of DNN are divided into groups of successive layers, called a chunk. Both FWP and BWP of all the layers in a chunk are processed by a single GPU, and in this way, different GPUs handle different chunks.

For each minibatch, a GPU performs its FWP for the layers in its chunk and sends it to the next GPU. The GPU processing the last chunk calculates the loss for the minibatch. Then, every GPU executes BWP and sends the loss to the GPU processing the previous chunk. With only one minibatch, at most one GPU is active at a time. To improve utilization, multiple mini-batches are injected in a pipelined manner. After performing FWP for a minibatch, each GPU asynchronously sends the output activations to the GPU processing the next chunk and starts processing another mini-batch. Likewise, upon performing BWP for a minibatch, every GPU asynchronously sends the gradient to the previous GPU and starts processing another mini-batch.

While the "bulk synchronous parallel" approach requires sending all the parameters of a DNN, their pipeline-parallel approach requires sending only the output data of one layer. This reduces the volume of communication significantly. Further, their approach overlaps transmission of forward output activations and backward gradients with useful computation and also improves GPU utilization efficiency. Their technique finds partitioning of layers of the DNN on the GPUs for minimizing the training time. Assuming that all the GPUs are identical, the processing load of each chunk should be similar, and the amount of data communicated should be minimal. Assuming that the communication and computation time is nearly the same for different mini-batches, their technique profiles each layer of the DNN on one GPU. Then, it uses a "dynamic-programming" based algorithm for grouping layers into chunks while also finding out the replication factor and optimal minibatch count for keeping the pipeline fully-utilized. Figure 37 shows an example of how their technique may divide layers of a CNN across chunks of eight GPUs.

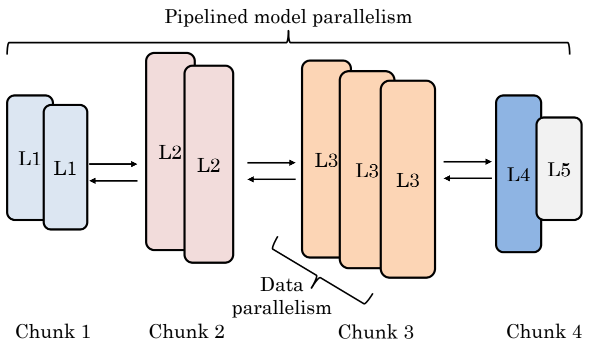


Fig. 37. Illustration of pipelined parallel training technique [44] (L1 to L5 are five layers of a CNN)

At any point in time, a GPU can perform either FWP or BWP, and both of them are important for ensuring forward progress of the training. Their scheduling scheme has a starting phase and a steady phase. In the starting phase, U mini-batches are injected where U=NumberOfMachines/NumberOfMachinesInFirstStep. In the steady phase, every GPU alternately executes one FWP pass and one BWP pass for a mini-batch. This avoids GPU idling, as shown in 38(b). By contrast, the naive pipelined training leads to GPU idling, and this is shown in Figure 38(a).

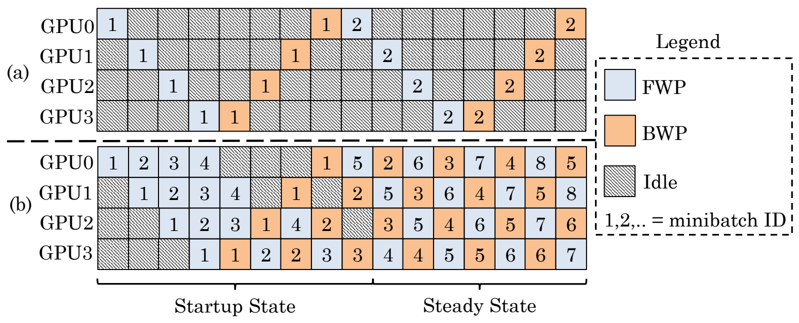


Fig. 38. (a) Naive pipelining scheme (b) "Pipelined parallel training" approach of Harlap et al. [44]

An advantage of their technique is that it does not require FWP and BWP phases to take the same amount of time. When a chunk is replicated on multiple GPUs, a "round-robin load-balancing" scheme is used for allocating work from the earlier chunks to the replicas. It allocates BWP of a minibatch to the same GPU which performed its FWP.

A limitation of the pipelined design is that the FWP and BWP for every minibatch happen with different parameters. For instance, as shown in Figure 38(b), on machine 2, the FWP for minibatch 5 happens after applying updates from minibatch 1 and 2, but the BWP for minibatch 5 happens after applying updates from mini-batches 3 and 4. Hence, due to differences in the weights used, the model may not converge. Also, the amount of staleness differs in different chunks. Due to these factors, the accuracy of the above pipelined design is much lower than that of data-parallel training. They propose two schemes to mitigate these issues: (1) For every active minibatch, a separate version of weights is maintained. The FWP is performed using the most up-to-date weights available, which are stored, and the same version is used for performing BWP. Thus, in a chunk, for a minibatch, the same weights are used in both FWP and BWP. (2) Consistent values of weights are used across the chunks. They note that scheme (1) is important for meaningful training, but scheme (2) has a negligible impact. Experiments show that their technique incurs much lower "time-to-accuracy" than the data-parallel training. Also, its benefits are higher in a cluster which has faster GPU but a slower network.

4.6 Designing schedulers for DL training

DL training involves a trial-and-error approach where multiple configurations of a job are explored, of which some are aborted or prioritized based on quick feedback. Conventional schedulers do not provide early feedback on all the jobs since they focus on running some jobs to completion and place other jobs in waiting queues. Thus, providing exclusive GPU access to some jobs leads to low utilization of GPU and high latency for other jobs. Further, DL training jobs have varying compute/memory/communication resource requirements and are impacted differently by these factors. For example, even for the same number of total GPUs, the number of GPUs per server decides the communication latency, which has a high impact on the performance of the CNN with larger model size. Similarly, different DL training jobs show different sensitivity to interference from co-running jobs. DNN training involves multiple mini-batch iterations, and during every mini-batch, the memory consumption increases during FWP reaches its peak and then decreases during BWP.

Xiao et al. [38] propose adapting the scheduling technique to DL training workload for improving the cluster utilization. When multiple training jobs are present, their technique enables time-sharing of GPUs by incoming jobs by using a suspend-resume scheme. Specifically, when a suspend command is issued, their technique waits till the memory consumption of the job becomes minimum, transfers the GPU data to CPU, frees GPU memory and calls the CPU "suspend scheme." Later, when the job is to be resumed, GPU memory is allocated, data is copied from CPU to GPU memory, and the job is resumed. Although waiting for the memory usage to reach its minimum value may delay the suspend operation by the duration of a mini-batch interval, it reduces CPU memory utilization and amount of data-transfer. When resumed, a job may run on a different GPU than the one it was originally running on. As an alternative to suspend-resume, multiple jobs can be packed and allow GPU to time-share the jobs. The limitation of job-packing is that it may lead to resource contention and interference between jobs.

Further, they perform job-migration to change the group of GPUs allocated to a training job using functions that enable automatic checkpointing and restoration of a model. The migration overhead can be reduced by warming up the destination before migration, and only moving the required training context. Migration enables the use of idle GPUs or consolidating jobs on few GPUs to achieve de-fragmentation.

Finally, their technique increases the number of GPUs allotted to a job at a high amount of load and vice versa. Based on the periodicity of training jobs, profiling is done to find the impact of a scheduling decision on mini-batch progress speed. Based on this, for example, the GPUs allocated to those jobs can be increased that show rising performance with an increasing number of GPUs. Their technique improves cluster utilization and reduces time to early feedback.

4.7 Using transient servers in the cloud for reducing the financial cost of training

Cloud providers offer two kinds of servers: on-demand that remain available until they are released by the customer and transient that may be revoked anytime by the cloud provider. Transient servers are offered as preemptible VMs (virtual machines), and spot instances and are as much as 90% cheaper than the dedicated servers[30]. In the "Google compute engine", the maximum lifetime of a transient server is 24 hours. In general, different cloud providers have different policies governing the lifetime of transient servers.

Li et al. [30] note that distributed deep learning incurs significant overheads and financial costs due to sublinear scaling. They propose use of transient servers to mitigate these costs. They note that distributed training on transient servers provides comparable accuracy and speed like that on on-demand servers while providing significant monetary savings. Also, compared to training on a single-GPU, distributed training on transient servers provides speedup and cost saving with an only small loss in accuracy. The loss in accuracy comes due to the use of stale parameters in distributed asynchronous training. A training cluster is adjudged to be a failure if the master worker gets revoked before completion of training. Hence, they propose redesigning existing distributed training frameworks so they can continue to work even if the master server, which performs checkpointing, fails.

They note that revoking of servers has a minor impact on accuracy and cost. The increase in training time due to revoking of servers can be mitigated by using a larger cluster. They further study whether within a fixed budget of training, using a more powerful GPU (e.g., P100 or V100) or using a cluster of GPUs provides any benefit over using an on-demand K80 GPU. They observe that the use of more powerful GPU increases vulnerability to server revocations whereas the use of GPU cluster improves the speed at the cost of accuracy. Overall, the use of four GPUs provides a balance between training speedup and accuracy loss.

Clusters with a large number of transient servers are more resilient to revocations since every server performs less amount of work in a large cluster and hence, the loss of time due to its revocation is lower. Apart from the number of revocations, the timing of revocations also has a crucial impact on the cost and training time. They also study heterogeneous training, which uses GPUs of different types (e.g., K80 and V100) or GPUs located at different geographical locations. On using GPU servers located in the same data-center, the saving in cost is nearly proportional to the slowdown. By comparison, using servers at geographically-different locations leads to huge slowdown since the communication between GPU workers and parameter-server happens over a slow network connection.

4.8 Comparison with CPU/Phi

Awan et al. [25] compare single-node and multi-node training using CPU, Phi, and GPU. They use Intel-Caffe and NVIDIA-Caffe for CPU/Phi and GPU, respectively. They find that compared to default Caffe, use of MKL2017 engine in Caffe improves the performance of CPU greatly, and the performance improvement in Phi is even higher. Then, they compare the three memory modes of Knights Landing (KNL): (1) use of "multi-channel DRAM" (MCDRAM) as L3 cache and (2) allocating entire data in DDR-memory or (3) MCDRAM memory. Of these, option (1) provides the highest performance and hence, is used for subsequent experiments.

They further compare overall training time of AlexNet and ResNet and find that MKL2017-optimized Broadwell CPU and KNL Phi provide much higher performance than K40 and K80 GPUs. KNL Phi provides comparable performance as P100 GPU, which outperform Broadwell and Haswell CPUs. On studying the time taken by individual layers, they find that FC layers and most CONV layers run faster on P100 than on KNL, whereas one CONV layer runs faster on KNL.

For multi-node training using CPU, they use Intel-Caffe which uses MPI via "Intel Machine Learning Scaling Library" (MLSL). Since NVIDIA-Caffe itself does not provide support for multi-node training, they use OSU-Caffe, which is built on top of NVIDIA-Caffe. On moving from one to two nodes, Intel-Caffe shows a sudden increase in training time due to communication overheads. OSU-Caffe does not show such a sharp increase. The throughput increases with the increasing number of nodes, and the throughput of OSU-Caffe remains higher than that of Intel-Caffe.

5 Concluding Remarks

This paper reviewed the techniques for accelerating DL applications on GPUs. We discussed the works that use GPUs for both training and inference, in both single-system and distributed systems. We organized the work based on key parameters for bringing out their similarities and differences. We close this paper with a discussion of future challenges.

In the last nine years, the compute performance of GPU has increased by 32 times, but the memory bandwidth has increased by 13 times only [31]. On the other hand, memory-requirements of DNNs are growing at a rapid pace. Evidently, data-movement is likely to become the bottleneck in training and testing of DNNs on GPUs in the future. Going forward, substantial improvements in memory capacity and bandwidth are required for keeping GPUs a platform of choice for large-sized DNNs. This may require using novel approaches, for instance, emerging non-volatile memories and computing-in-memory approach [73, 74]. Similarly, data-movement overheads can be reduced using data-encoding techniques [75].

Most works have evaluated optimization techniques such as pruning, tiling, suitable data-layout, etc. in isolation. Since in real-life applications, these techniques will be used in conjunction, a thorough evaluation is required to ensure that these techniques integrate synergistically. A crucial challenge in the use of GPUs is their large power consumption [65], which obstructs their use in power-constrained application scenarios such as autonomous driving. However, nearly all the works have ignored the evaluation of the energy efficiency of their solutions. Future works need to evaluate the performance benefits of their techniques, vis-a-vis their energy overheads. Future works also need to evaluate recent GPU models (e.g., Volta and Turing) and low/mixed-precision techniques for achieving high efficiency.

In recent years, several custom-made AI accelerators such as Google's tensor processing unit (TPU) have been introduced in the market. While the general-purpose nature of GPU makes it useful for a broad range of applications, it also precludes thorough optimization of GPU architecture for AI applications. It remains to be seen whether the future trajectory of GPU architecture will see revolutionary or evolutionary changes [76]. It will be also interesting to see how well the next-generation GPU strikes a balance between the conflicting goals of special-purpose and general-purpose computing, and how well it competes with the other AI accelerators [77–79].

As CNNs become progressively used in mission-critical applications, ensuring their security has become important. However, GPUs do not guarantee a trusted execution environment and have many security vulnerabilities [80]. Further, outsourcing DNN's training on GPUs in remote clouds increases the vulnerability to attacks. Moving forward, security concerns need to be addressed in the design of both CNNs and GPUs as the first principle, instead of retrofitting for it.

REFERENCES

[1] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[2] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "100-epoch imagenet training with alexnet in 24 minutes," *ArXiv e-prints*, 2017.

[3] S. Mittal, "A Survey of Techniques for Architecting and Managing GPU Register File," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2016.

[4] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, 2017, pp. 548–560.

[5] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 2019, pp. 229–241.

[6] S. Mittal, "A Survey on Optimized Implementation of Deep Learning Models on the NVIDIA Jetson Platform," *Journal of Systems Architecture*, vol. 97, pp. 428 – 442, 2019.

[7] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *European Conference on Computer Systems*, 2016, p. 4.

[8] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-Latency and Scalable RNN Inference on GPUs," in *Proceedings of the Fourteenth EuroSys Conference*, 2019, p. 41.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[10] H. Park, D. Kim, J. Ahn, and S. Yoo, "Zero and data reuse-aware fast convolution for deep neural networks on GPU," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2016, pp. 1–10.

[11] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 633–644.

[12] X. Chen, J. Chen, D. Z. Chen, and X. S. Hu, "Optimizing memory efficiency for convolution kernels on Kepler GPUs," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[13] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "DeftNN: Addressing bottlenecks for DNN execution on GPUs via synapse vector elimination and near-compute data fission," in *International Symposium on Microarchitecture*, 2017, pp. 786–799.

[14] X. Chen, "Escort: Efficient sparse convolutional neural networks on GPUs," *arXiv preprint arXiv:1802.10280*, 2018.

[15] P. Jain, X. Mo, A. Jain, A. Tumanov, J. E. Gonzalez, and I. Stoica, "The OoO VLIW JIT Compiler for GPU Inference," *arXiv preprint arXiv:1901.10008*, 2019.

[16] B. Van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra, "Optimizing convolution operations on GPUs using adaptive tiling," *Future Generation Computer Systems*, vol. 30, pp. 14–26, 2014.

[17] A. Zlateski, K. Lee, and H. S. Seung, "ZNNi: maximizing the inference throughput of 3D convolutional networks on CPUs and GPUs," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, p. 73.

[18] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory CNN across GPU microarchitectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 1–12.

[19] X. Lu, H. Shi, R. Biswas, M. H. Javed, and D. K. Panda, "DLoBD: A Comprehensive Study of Deep Learning over Big Data Stacks on HPC Clusters," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 4, pp. 635–648, 2018.

[20] A. Lavin, "maxDNN: an efficient convolution kernel for deep learning with Maxwell GPUs," *arXiv preprint arXiv:1501.06633*, 2015.

[21] R. Xu, F. Han, and Q. Ta, "Deep Learning at Scale on NVIDIA V100 Accelerators," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 23–32.

[22] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 27–40.

[23] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance analysis of GPU-based convolutional neural networks," in *International Conference on Parallel Processing (ICPP)*, 2016, pp. 67–76.

[24] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *International Symposium on Microarchitecture*, 2016, p. 18.

[25] A. A. Awan, H. Subramoni, and D. K. Panda, "An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures," in *Machine Learning on HPC Environments*, 2017, p. 8.

[26] X. Chen, D. Z. Chen, and X. S. Hu, "moDNN: Memory optimal DNN training on GPUs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 13–18.

[27] S. Dong, X. Gong, Y. Sun, T. Baruah, and D. Kaeli, "Characterizing the microarchitectural implications of a convolutional neural network (CNN) execution on GPUs," in *International Conference on Performance Engineering*, 2018, pp. 96–106.

[28] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "SuperNeurons: Dynamic GPU memory management for training deep neural networks," in *ACM SIGPLAN Notices*, vol. 53, no. 1. ACM, 2018, pp. 41–53.

[29] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie, "Performance evaluation and optimization of HBM-Enabled GPU for data-intensive applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, pp. 831–840, 2018.

[30] S. Li, R. J. Walls, L. Xu, and T. Guo, "Speeding up deep learning with transient servers," *arXiv preprint arXiv:1903.00045*, 2019.

[31] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "DeLTA: GPU Performance Model for Deep Learning Applications with In-depth Memory System Traffic Analysis," *arXiv preprint arXiv:1904.01691*, 2019.

[32] T. Jin and S. Hong, "Split-CNN: Splitting Window-based Operations in Convolutional Neural Networks for Memory System Optimization," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 835–847.

[33] S. Li, Y. Zhang, C. Xiang, and L. Shi, "Fast convolution operations on many-core architectures," in *International Conference on High Performance Computing and Communications*, 2015, pp. 316–323.

[34] Q. Chang, M. Onishi, and T. Maruyama, "Fast convolution kernels on Pascal GPU with high memory efficiency," in *High Performance Computing Symposium*, 2018, p. 3.

[35] C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu, "Training deeper models by GPU memory optimization on TensorFlow," in *Proc. of ML Systems Workshop in NIPS*, 2017.

[36] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," *arXiv preprint arXiv:1412.7580*, 2014.

[37] P. Sun, W. Feng, R. Han, S. Yan, and Y. Wen, "Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes," *arXiv preprint arXiv:1902.06855*, 2019.

[38] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 595–610.

[39] T. D. Le, T. Sekiyama, Y. Negishi, H. Imai, and K. Kawachiya, "Involving CPUs into Multi-GPU Deep Learning," *International Conference on Performance Engineering*, pp. 56–67, 2018.

[40] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, and D. K. Panda, "OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training," in *International Conference on High Performance Computing (HiPC)*, 2018, pp. 143–152.

[41] Y. Oyama, T. Ben-Nun, T. Hoefler, and S. Matsuoka, "Accelerating deep learning frameworks with micro-batches," in *International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 402–412.

[42] H. Mikami, H. Suganuma, P. U-chupala, Y. Tanaka, and Y. Kageyama, "Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash," in *Arxiv*, 2018.

[43] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in Four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[44] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "PipeDream: Fast and

efficient pipeline parallel DNN training," *arXiv preprint arXiv:1806.03377*, 2018.
[45] Y. Zhao, L. Wang, W. Wu, G. Bosilca, R. Vuduc, J. Ye, W. Tang, and Z. Xu, "Efficient communications in training large scale neural networks," in *Thematic Workshops of ACM Multimedia*, 2017, pp. 110–116.
[46] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on GPU and Knights Landing clusters," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, p. 9.
[47] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters," in *Acm Sigplan Notices*, vol. 52, no. 8, 2017, pp. 193–205.
[48] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.
[49] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing, "Poseidon: A system architecture for efficient GPU-based deep learning on multiple machines," *arXiv preprint arXiv:1512.06216*, 2015.
[50] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training ResNet-50 on ImageNet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.
[51] J. H. Park, S. Kim, J. Lee, M. Jeon, and S. H. Noh, "Accelerated Training for CNN Distributed Deep Learning through Automatic Resource-Aware Layer Placement," *arXiv preprint arXiv:1901.05803*, 2019.
[52] M. Song, Y. Hu, Y. Xu, C. Li, H. Chen, J. Yuan, and T. Li, "Bridging the semantic gaps of GPU acceleration for scale-out CNN-based big data processing: Think big, see small," in *International Conference on Parallel Architectures and Compilation*, 2016, pp. 315–326.
[53] V. Campos, F. Sastre, M. Yagües, J. Torres, and X. Giró-i Nieto, "Scaling a convolutional neural network for classification of adjective noun pairs with TensorFlow on GPU clusters," in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 677–682.
[54] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.
[55] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," in *International Conference on Parallel Processing*, 2018, p. 1.
[56] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.
[57] J. Guo, W. Liu, W. Wang, Q. Lu, S. Hu, J. Han, and R. Li, "AccUDNN: A GPU Memory Efficient Accelerator for Training Ultra-deep Deep Neural Networks," *arXiv preprint arXiv:1901.06773*, 2019.
[58] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating on-node GPU interconnects for deep learning workloads," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2017, pp. 3–21.
[59] K. Zhou, G. Tan, X. Zhang, C. Wang, and N. Sun, "A performance analysis framework for exploiting GPU microarchitectural capability," in *International Conference on Supercomputing*, 2017, p. 15.
[60] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs," in *International Conference on Supercomputing*, 2017, p. 5.
[61] M. Jorda, P. Valero-Lara, and A. J. Peña, "Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs," *IEEE Access*, 2019.
[62] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.
[63] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving RNN-based deep learning models 10x faster," in *USENIX ATC*, 2018, pp. 951–965.
[64] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 69:1–69:35, 2015.
[65] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," *ACM Computing Surveys*, vol. 47, no. 2, pp. 19:1–19:23, 2015.
[66] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 578–594.
[67] S. Mittal, "A Survey of FPGA-based Accelerators for Convolutional Neural Networks," *Neural computing and applications*, 2018.
[68] L. Meng and J. Brothers, "Efficient Winograd Convolution via Integer Arithmetic," *arXiv preprint arXiv:1901.01965*, 2019.
[69] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *International Conference on High Performance Computing*, 2016, pp. 21–38.
[70] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv preprint arXiv:1802.09941*, 2018.
[71] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.
[72] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *International conference on machine learning*, 2013, pp. 1337–1345.
[73] S. Umesh and S. Mittal, "A survey of spintronic architectures for processing-in-memory and neural networks," *Journal of Systems Architecture*, vol. 97, pp. 349 – 372, 2018.
[74] S. Mittal, "A survey on applications and architectural-optimizations of micron's automata processor," *Journal of Systems Architecture*, vol. 98, pp. 135 – 164, 2019.
[75] S. Mittal and S. Nag, "A survey of encoding techniques for reducing data-movement energy," *Journal of Systems Architecture*,

vol. 97, pp. 373 – 396, 2018.
[76] S. Shead, "Nvidia's got a cunning plan to keep powering the AI revolution," https://www.wired.co.uk/article/nvidia-artificial-intelligence-gpu, 2019.
[77] M. Wielomski, "The GPU: Powering The Future of Machine Learning and AI," https://phoenixnap.com/blog/future-gpu-machine-learning-ai, 2018.
[78] A. Sharma, "GPU Vs CPU: The Future Think-Tanks Of AI," https://www.analyticsindiamag.com/gpu-vs-cpu-future-think-tanks-ai/, 2018.
[79] G. Anadiotis, "AI chips for big data and machine learning: GPUs, FPGAs, and hard choices in the cloud and on-premise," https://www.zdnet.com/article/ai-chips-for-big-data-and-machine-learning-gpus-fpgas-and-hard-choices-in-the-cloud-and-on-premi
2018.
[80] S. Mittal, S. B. Abhinaya, M. Reddy, and I. Ali, "A Survey of Techniques for Improving Security of GPUs," *Hardware and Systems Security Journal*, vol. 2, no. 3, pp. 266–285, 2018.