BrickDL: Graph-Level Optimizations for DNNs with Fine-Grained Data Blocking on GPUs

Mahesh Lakshminarasimhan
Mary Hall
{maheshl,mhall}@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Samuel Williams
Oscar Antepara
{swwilliams,oantepara}@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, California, USA

ABSTRACT

The end-to-end performance of deep learning model inference is often limited by excess data movement on GPUs. To reduce data movement, existing deep learning frameworks apply graph-level optimizations such as operator fusion to exploit data reuse across operators in deep learning graphs. Such optimizations are limited and cannot optimize arbitrary chains of compute- and time-intensive operations, including convolutions. To address these limitations, this paper presents BrickDL, a deep learning inference library that implements *merged execution* of a sequence of layers as an orthogonal approach to existing graph-level optimizations. BrickDL additionally employs fine-grain blocking using a *brick* data layout that further optimizes data locality on GPUs. We implement merged execution with the abstraction of bricks using two approaches – *padded bricks* and *memoized bricks*, and develop a performance model to choose between them using static analysis. Merged execution with bricks demonstrates performance gains on well-known deep learning models as compared to PyTorch JIT, TensorFlow XLA, and cuDNN baselines on NVIDIA A100 GPU. We also characterize the performance of the proposed optimizations with microbenchmarks and gain insights into their applicability and tradeoffs.

CCS CONCEPTS

• **Computing methodologies** → *Parallel computing methodologies*; *Machine learning*; • **Software and its engineering** → *Software notations and tools.*

KEYWORDS

Graph-level Optimizations, Deep Learning, Data Layout, Memoization, Runtime, Performance Modeling, GPU

ACM Reference Format:
Mahesh Lakshminarasimhan, Mary Hall, Samuel Williams, and Oscar Antepara. 2024. BrickDL: Graph-Level Optimizations for DNNs with Fine-Grained Data Blocking on GPUs. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673046

ICPP '24, August 12–15, 2024, Gotland, Sweden
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673046

1 INTRODUCTION

Modern GPUs have evolved with phenomenal throughput, but their memory performance lags behind their exceptional computational efficiency. Thus, the end-to-end performance of deep learning models, with the ever-increasing number of layers, is significantly bottlenecked by data movement across the memory subsystem. It is quite challenging to achieve proper memory bandwidth utilization and high on-chip data locality in GPUs for deep neural networks (DNN) due to the complex memory access patterns of DNN computations and high-dimensional tensors they operate on (e.g., 5D tensor for a 3D convolution operation). Performance is further impeded by suboptimal data layouts and inefficient scheduling of the order of execution on the compute elements of GPUs, which miss data reuse opportunities.

The computational pattern of a DNN can be modeled as a data flow graph, with each node denoting an operator and the edge representing the dependence between operators. Existing deep learning frameworks (e.g., TensorFlow, PyTorch), compilers (TVM [6], XLA [29], Triton[36]), and libraries (cuDNN, TensorRT [1]) primarily focus on optimizing the performance of standalone operators in DNN graphs. While operator-level optimizations improve computational efficiency, it is necessary to utilize spatial and temporal data reuse across layers to improve memory performance. Specialized graph compilers [20, 24, 27, 43, 49, 56] perform graph rewriting and operator/kernel/loop fusion based on a restricted set of rules and a limited number of patterns that can be combined together. Due to complex data dependences between operators, operator fusion is not applicable to arbitrary chains of compute-intensive operations involving convolutions, which constitute most of the execution time of a DNN pipeline, affecting the overall performance.

In this paper, we present BrickDL, a graph-level optimizing library and runtime for DNN inference. In contrast to existing frameworks, BrickDL performs *merged execution* of a sequence of DNN layers, including chains of compute-intensive convolution operations. BrickDL employs fine-grained data blocking with *bricks* – a data layout of small, fixed-sized blocks of contiguously-packed data that represent logically adjacent multi-dimensional data. Bricks enhance on-chip data residency on GPUs by reducing access strides and avoiding extraneous address streams. Using *merged execution* with *bricks*, BrickDL thereby optimizes the end-to-end memory performance of DNN pipelines.

BrickDL uses the abstraction of bricks as a unit of data and parallelism for applying inter-layer optimizations on DNN graphs. BrickDL partitions DNN graphs into subgraphs, and layers within the subgraphs are decomposed into bricks. With merged execution, bricks within subgraphs are executed asynchronously in a modified order so that the output brick from one layer is still resident in cache for the next layer's computation. The presence of data-dependent boundaries in DNN computations, such as halo regions in convolutions similar to stencils, complicates the merged execution of layers. We develop two optimizations — *padded bricks* and *recursive memoized bricks*, to facilitate inter-layer merged execution by avoiding data dependences. The former performs data copies of halo regions in bricks, while the latter involves a recursive approach with memoization to keep track of data-dependent bricks.

Our approach to merged execution for DNN workloads is related to well-known space-time tiling, cache-oblivious, wavefront, time skewing approaches for stencil computations [13, 25, 35, 38]. However, these techniques are not directly applicable to DNN graphs since computations vary across layers, as opposed to stencil applications that involve identical operations between time steps. BrickDL, on the other hand, combines the concepts of *execution reordering* and *data reorganization* to facilitate inter-layer merged execution with bricks, and memoization using dynamic runtime.

The performance of BrickDL is assessed with seven prominent DNN models on NVIDIA A100 GPUs, demonstrating speedups over PyTorch JIT, TensorFlow XLA, and cuDNN baselines. We believe it makes a strong case to integrate the brick data layout and merged execution optimizations into state-of-the-art deep learning frameworks, compilers, and libraries for improved performance.

The contributions of this paper are the following:

- We present the BrickDL library that applies fine-grain data layouts and graph-level optimizations for DNN inference.
- We propose the *padded bricks* and *memoized bricks* optimizations for the *merged execution* of layers, along with performance models to guide optimizations at compile time.
- Using microbenchmarks, we perform a comprehensive evaluation of the applicability and tradeoffs associated with these optimizations and gain insights.
- We demonstrate the efficacy of the proposed optimizations on well-known deep learning models on NVIDIA GPUs.

2 MOTIVATION: DATA MOVEMENT

Convolutions are inherently compute-intensive operations, and generally, over 80% of the computation time of convolutional neural network (CNN) pipelines is spent performing convolutions. Convolutions are most commonly implemented using the sliding window approach in which a kernel or filter slides over the input data to perform localized element-wise multiplications, and the output is written to a feature map.

Existing deep learning frameworks typically store data in memory in canonical row-major order. Convolutions thus access *logically* neighboring but *physically* distant data along the non-contiguous dimensions, involving long reuse distances. The accessed data is hence spread across multiple address streams, causing large memory access strides that place enormous pressure on register files, cache, TLB, and hardware prefetchers. Optimizations such as tiling enable reuse in cache but do not alleviate and may even exacerbate the immense strain on the number of cache lines and TLB entries. Convolutions are thus often affected by vertical data movement, even though they are typically compute-bound.

The end-to-end CNN performance is often bottlenecked by data movement on GPUs. Deep learning frameworks apply graph-level optimizations to utilize data reuse across CNN layers. Operator fusion is one such common optimization that combines multiple elementary operations or computational kernels in a graph into a single, large, complex optimized kernel. This could be fusion of a chain of a compute-intensive operation (e.g., convolution, matrix multiplication) and a memory-intensive operation (e.g., ReLU, pooling, softmax), or fusion of a chain of memory-intensive operations [6, 11, 24, 27, 37, 49, 56].

However, research on the fusion of back-to-back compute-intensive operations is limited, due to strict dependences between such operations. There exists loop-carried dependences along the input and output channel dimensions, and operator fusion requires the spatial dimensions and kernel size of the operators in the chain to be the same. The pseudocode in Figure 2(b) demonstrates operator fusion of two convolutions from Figure 2(a) under these conditions, with the ability to fuse only along the batch dimension n. Tiled execution of convolutions produces data-dependent *halo regions* surrounding each tile, which requires synchronization of tiles with a reduction after each operation's execution before proceeding to the next, impeding fusion.

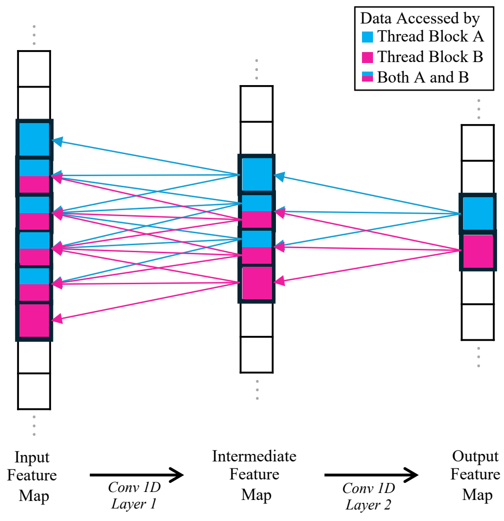

Figure 1: A subgraph with two 1D convolution operations demonstrating the redundant computation of dependent chunks of data in two thread blocks.

Performance is further hampered by inefficient scheduling of the execution of convolutions on GPUs, causing redundant computation of halo regions. Consider a simple subgraph with two 1D convolutions as shown in Figure 1 tiled along the spatial dimension. The two different thread blocks A and B respectively produce two tiles shaded red and blue from the second convolutional layer. Each thread block would redundantly compute the neighboring tiles in the first convolutional layer to cover halo regions (three tiles in this simple example). There would be an even more significant duplication of computations for deeper subgraphs with multidimensional convolutions and large filters.

```
for n,c1,h1,w1,x1,y1,m1
//Tiled 7D Loop
{ //Compute First Conv2D }
for n,c2,h2,w2,x2,y2,m2
//Tiled 7D Loop
{ //Compute Second Conv2D }
```

(a) Naive

```
for n { //fuse batch dimension
  for c1,h1,w1,x1,y1,m1 { //Tiled
  { //Compute First Conv2D}
  for c2,h2,w2,x2,y2,m2 { //Tiled
  { //Compute Second Conv2D}
}
```

(b) Operator Fusion

```
for Bh, Bw { //each brick in parallel
  for nlayer{ //each subgraph layer
    //bricks padded with halo: px, py
    for n, c, Bh+px, Bw+py, r, s, m
    {//Compute conv2D}}}
reduce() //Aggregate all bricks
```

(c) Merged Execution: Padded

```
compConv2D (layers, bricks) {
  for Bh, Bw { //each brick
    getStatus() //Check brick status
    //recursively compute bricks
    compConv2D(nlayer, brick)
  } reduce() //aggregate bricks
}
```

(d) Merged Execution: Memoized

Figure 2: Comparison of naive version, operator fusion, and merged execution with padded and memoized bricks for two convolutions. The naive version is tiled and sequentially computes both convolutions. Operator fusion fuses along the batch (n) dimension. Padded bricks and memoized versions copy halo regions and recursively backtrack between layers, respectively.

3 BRICKDL: MERGED EXECUTION WITH FINE-GRAINED DATA BLOCKING

This section describes how BrickDL supports graph-level optimizations that span a sequence of compute-intensive convolutional layers sidestepping the limitations of operator fusion. BrickDL is a library that combines: (1) brick data layout for representing data; (2) a dynamic runtime that supports merged execution; and, (3) static analysis to guide optimizations.

3.1 Brick Data Layout

To address the challenges associated with suboptimal memory accesses for convolutions, we apply the fine-grain blocked *brick* data layout to computations on DNN graphs. Prior work on optimizing computations on higher-order stencils applied this approach to improve data locality on CPUs and GPUs [50, 52]. The brick data layout decomposes high-dimensional input data and feature maps into small, fixed-size multi-dimensional blocks stored contiguously in memory. Each brick represents a unit of data moving through the memory hierarchy as well as an aggregate unit of parallelism.

While a cache line naturally captures spatial locality in only one dimension of a structured grid, bricks can exploit spatial locality in three or more dimensions. Fine-grained blocking reduces the number of strides necessary when accessing neighboring elements. By comparison, tiling in the non-contiguous dimension significantly increases the number of strides taken within the tile. The brick data layout thus achieves high performance by exploiting data reuse across multiple axes using a single address stream, reducing TLB pressure, minimizing cache line and register reuse distances, and improving memory bandwidth utilization.

3.2 Inter-Layer Merged Execution With Bricks

We now describe *merged execution* of layers as a graph-level optimization for chains of compute-intensive operations using bricks. Figure 3(b), a three-operator subgraph, depicts the merged execution of bricks marked red that perfectly overlap with the execution of other bricks and hence need not synchronize after each operation. This approach merges the execution of computations asynchronously on fine-grained bricks across operations in a subgraph partitioned from a given DNN graph. Instead of generating fused kernel code, we invoke each operator's kernel at the fine-grained granularity of bricks and reschedule their order of execution, in contrast to the standard execution order of operators. This optimization is aimed at exploiting data reuse across operators and improving data locality at the higher levels of the GPU memory hierarchy. Merged execution accomplishes this by eliminating data dependence between operators across layers, thus minimizing redundant sweeps over large, multi-dimensional activation layers for the execution of each operation (Figure 3(a)). The BrickDL library implements merged execution by applying bricks with fine-grained blocking along the batch and spatial dimensions of activations (height, width, depth), avoiding the channel dimension.

Beyond a sequence of convolutions, merged execution is applicable to any operation for which a block of the input data of size X_i along dimension i yields an output data block with size of the form $\alpha_i X_i + \beta_i$, for constants α and β. Operations compatible with this form include localized operations such as non-linear functions (e.g., ReLU, softmax, sigmoid), linear activation functions, and different types of convolutions (strided, dilated, depthwise/spatially separable, deconvolution, etc.).

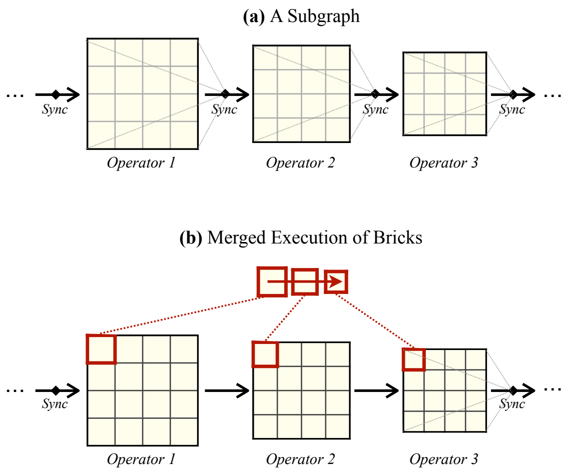

Figure 3: Comparing graph optimizations for conventional and brick data layouts: (a) is a subgraph of 3 layers partitioned from a graph, requiring synchronization of tiles after each operation; (b) shows fine-grained blocking into bricks and merged execution, with bricks synchronized only at the end.

To address cross-layer data dependences of halo regions between bricks, we implement merged execution using (i) *padded bricks*, that copies halo regions from neighboring bricks (section 3.2.1), and (ii) *memoized bricks*, that recursively tracks and computes dependent bricks at runtime, avoiding redundancies (section 3.2.2). Figure 2-(c),(d) demonstrates merged execution of bricks with these approaches, perfectly overlapping computations between layers, with a reduction across bricks only at the end of the subgraph.

3.2.1 Padded Bricks. BrickDL enables merged execution of layers in operation chains by expanding the boundaries of bricks; it makes copies of data-dependent halo regions from neighboring bricks from the same operation. This approach is complementary to padding in convolutions, where the boundaries of feature maps are padded with extra data before performing the convolution operation to preserve spatial dimensions and enable access to boundary artifacts.

Figure 4 demonstrates merged execution with padded bricks for two convolution operations and their corresponding feature maps. In merged execution, computation on each 2D brick yields a fine-grained 2D block of the entire activation (assuming blocking along height and weight dimensions only). To produce a $B_h \times B_w$ brick as an output for the second convolution operation (shaded green in Figure 4), it requires an enlarged brick of size $(B_h+2p_x)\times(B_w+2p_y)$ as input (green and red shaded regions) with padded halo region $p_x = \frac{X-1}{2}$ and $p_y = \frac{Y-1}{2}$ along the height and width dimensions of the brick, respectively, for an $X \times Y$ kernel. Correspondingly, to output a $(B_h + 2p_x) \times (B_w + 2p_y)$ brick from the first convolution operation, a $(B_h + 4p_x) \times (B_w + 4p_y)$ (blue, green and red shaded area) is needed as an input. For ReLU, softmax, and other element-wise operations, no padding is required ($p_x, p_y = 0$). For a max pool operation, the padding factor is the pooling stride S_p, which is multiplied by the brick dimension (i.e., $B_h \times p_x$, where $p_x = S_p$).

BrickDL employs a static analysis algorithm to determine the size of bricks with padding in a given subgraph of operations. The subgraph is traversed in reverse order, and the padding factor is computed and added to the size of each brick, corresponding to each node in the subgraph, tracked using a queue data structure. The padding factor varies depending on the position of bricks in the data grid (i.e., corner, edge, and central position in a 2D activation).

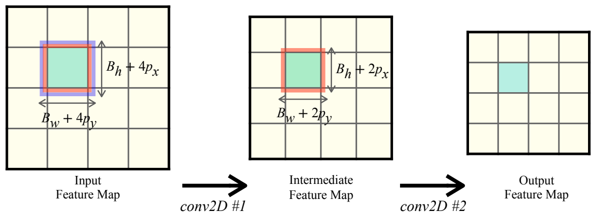

Figure 4: Padding for the merged execution of bricks in a subgraph with 3 layers. Padding factors are $p_x = \frac{X-1}{2}$ and $p_y = \frac{Y-1}{2}$ for brick with height B_h, width B_w, and kernel $X \times Y$.

3.2.2 Memoized Bricks with Recursion. Memoized bricks is a second approach for merged execution in BrickDL. Memoized bricks is a recursive, dynamic runtime approach inspired by top-down dynamic programming with memoization. Instead of redundantly computing data-dependent bricks entirely or padding the corresponding bricks with halo regions, this approach recursively computes those bricks with an asynchronous order of execution. This forms a data dependency chain between bricks across layers, eliminating conflicts between thread blocks and avoiding duplicate computation of dependent bricks. For the example in Figure 1, consider the data chunks shaded blue and red as bricks. With the memoized bricks approach, the execution of blue-shaded bricks is merged by backtracking the dependent blue bricks from previous layers and executing them in a modified order in thread block A. Similarly for red bricks with thread block B, thus averting redundant computations of bricks in thread blocks A and B.

To avoid duplicate computation of dependent bricks, bricks corresponding to the operators in a subgraph are memoized or cached for fast lookup. This is facilitated by tagging each brick with an auxiliary data structure based on three states shown in figure 5: (0) *Not started* – the brick has not been computed yet, (1) *In progress* – the brick is being computed, and (2) *Complete* – the brick has been computed. A thread computing a brick turns its status from 0 to 1. When it backtracks and accesses a dependent brick tagged (1), it yields to the other thread to complete its execution, and stalls until the status updates to (2) by issuing an atomic CAS (Compare-and-Swap) operation. In another case, when a thread encounters a dependent brick tagged (0), it moves on to accessing other bricks and revisits this brick later instead of stalling until it turns to (2). The best case scenario is when a brick tagged (2) is accessed in the first attempt without requiring additional atomic operations. In any case, accessing each brick involves two *compulsory* atomic operations – one to acquire the thread lock and the other to release the lock. It may incur additional *conflicting* atomic operations depending on the state of the brick. The overhead due to atomic operations is minimal – the time for each atomic operation is modeled as 87.45ns on NVIDIA A100 GPU using the method described in section 4.5. With the point of synchronization typically being L2 cache on NVIDIA GPUs, it enables high bandwidth access to the memoized bricks written to and read from atomically by different threads.

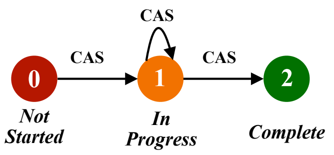

Figure 5: Three-state system for tagging bricks in the memoized bricks approach.

3.3 Performance Modeling and Implementation

BrickDL uses static analysis to generate subgraphs and to identify the optimal brick size and the best merged execution strategy among padded and memoized bricks, using performance models.

3.3.1 DNN Graph Partitioning. BrickDL performs static analysis to partition an input DNN graph into subgraphs, with all operators within a subgraph merged for execution with bricks. During this analysis, a reverse traversal of the entire graph is performed, computing the data footprint associated with the sequence of operators for the padded and memoized bricks approaches, factoring in the additional data from padding and the auxiliary data structures in memoized bricks. Based on this, operators are grouped together, and the graph is partitioned to obtain subgraphs. The analysis ensures that the data footprint associated with a subgraph from merged execution can be resident on chip in a GPU, e.g., 40 MB L2 cache on an NVIDIA A100 GPU.

The analysis typically places the last node in a subgraph as a reduction operation (e.g., pooling) or global operations (e.g., batch normalization). The former case is to reduce the overhead from excess padded data and atomic operations as the layer size shrinks within a subgraph due to, for example, reduction with large pooling strides. The latter case is to minimize the number of synchronizations between bricks for global operations.

3.3.2 Modeling Merged Execution Approaches. The *padded bricks* approach performs additional computations on the padded halo regions of bricks and, in turn, reduces the necessity of extraneous, expensive synchronization points in DNN graphs. For deeper subgraphs and smaller layer sizes, padded data and redundant computations cause overhead and affect performance. In those cases, we employ the *memoized bricks* approach, which avoids redundancies at the expense of atomic operations. For large layers, atomics cause excess overhead, in which cases padded bricks is preferred since the overhead of padding is minimal.

BrickDL chooses between the padded and memoized bricks approaches during the static analysis for DNN graph partitioning, which analyzes the cost of padded data for each subgraph. When the percentage of data growth due to padding (Δ) exceeds a certain threshold for a given subgraph, memoized bricks is used, avoiding the excess overhead of padding bricks. So when $\Delta > 15\%$ for a subgraph, BrickDL chooses to apply memoized bricks instead of padded bricks. This value of Δ has been validated on multiple NVIDIA and AMD GPU architectures.

3.3.3 Modeling Brick Size. We choose the most performant brick size for each subgraph using a performance model based on the amount of parallelism obtained for a given activation layer. For n number of blocked dimensions of feature maps of size $D_1, D_2, ..., D_n$, the amount of parallelism with ρ threads is given by $\rho = (D_1 \times D_2 \times .... \times D_n)/B^n, \forall B \in \{4, 8, 16, 32\}$. While maximum ρ can offer high performance theoretically, performance deteriorates beyond a certain threshold $\tau = 2^{12}$. The model thus picks B for a maximum value of ρ such that $\rho \leq \tau$.

Towards the end of a DNN graph, tiny layer sizes do not benefit from merged execution with fine-grained blocking due to insufficient parallelism. For those few cases, when $\rho < B^n$, we leverage cuDNN library instead of merged execution with bricks.

3.3.4 Implementation. BrickDL is implemented as a C++ template library. It uses three primary data structures to represent the brick data layout: `Brick`, `BrickMap`, and `BrickInfo`. Each `Brick` is a small, fixed-size block of data, and elements within each `Brick` are contiguously stored in the conventional row-major multi-dimensional data layout. `Brick` is the access interface that overloads operators with brick indices such that values can be retrieved from individual elements within a brick using array-based accesses.

The second data structure `BrickMap` maps the logical location of a brick to its physical location in memory. It contains the allocation information to manage memory regions accessed by a `Brick`. Even though elements within each brick are contiguously packed in memory, the blocks of bricks need not be physically stored in the conventional row-major/column-major order. Instead, the logical ordering of bricks is preserved using adjacency information that allows flexibility in how bricks are organized in memory. The third data structure `BrickInfo` is an array of adjacency lists that provides adjacency information for each Brick with the indices of its logical neighbors and their direction on a single data stream.

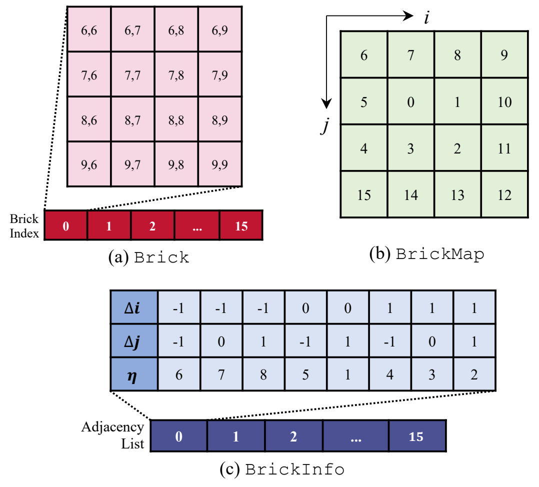

Figure 6: Representation of a 4×4 brick from a 16×16 array with the three corresponding data structures in BrickDL.

Figure 6 shows the representation of a 4×4 brick data layout from a 16×16 2D array. Figure 6(a) shows the array elements stored in the `Brick` structure at index 0. Figure 6(b) is `BrickMap`, a layer of indirection, mapping the `Brick` at logical index $(1, 1)$ to its physical location at index 0. Figure 6(c) shows `BrickInfo`, an adjacency list storing the physical indices of neighbors η to the brick at index 0. BrickDL expresses DNN operations using templates and operator loading to automatically translate [*Brick Index*, *Index In Brick*] tuples to the corresponding offset in memory, leveraging the adjacency list to access neighboring bricks.

BrickDL performs fine-grained blocking of activation layers along all spatial and sample dimensions. Brick size is kept constant for operators in a subgraph, while brick size can vary between subgraphs. A brick's size is set greater than the kernel/filter size for a given operation. A brick is masked accordingly with zero-valued elements when the layer size is not a multiple of the brick size.

Each brick makes a fine-grained invocation to the cuDNN library API for the corresponding operations. CUDA kernels are launched from the device using dynamic parallelism and CUDA graphs with the CUDA runtime API. BrickDL also leverages the operator fusion capabilities of cuDNN to fuse DNN primitives (e.g., convolution) with point-wise operations (e.g., ReLU, softmax). This is done with the NVIDIA cuDNN Backend API by configuring an engine with operation graphs of fused operators.

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

All experiments are run on the LBNL NERSC Perlmutter supercomputer. Each Perlmutter GPU node has one AMD EPYC 7763 GPU and four NVIDIA A100 GPUs. Each GPU includes 108 streaming multiprocessors (SM), and each SM contains 192 KB shared memory and shares a 40 MB L2 and 40 GB of HBM accessible at 1.5 TB/s.

The BrickDL code is compiled with CUDA 12.1.10 and we used PyTorch 1.12, TensorFlow 2.14.1, Python 3.8.8, cuDNN v8.8.2. The performance metrics are collected from NVIDIA Nsight Compute 2023.2.1. The fundamental kernels in BrickDL and baselines use single-precision floating point types. Each experiment is run 20 times, and the mean is reported.

4.2 End-to-End Model Inference Performance

Evaluated Models. The efficacy of BrickDL is evaluated with seven well-known CNN models with varying depth, convolution types, and model architectures: (i) VGG-16 [32], (ii) ResNet-50 (with identity and projection skip connections) [16], (iii) DarkNet-53 from YOLOv3 [28], (iv) 3D ResNet-34 with 3D convolutions [15], (v) DRN-26 (DRN-C) [48] with residual blocks involving 2D dilated, strided convolutions. (vi) DeepCAM [22], based on encoder-decoder architecture with deconvolutions and asymmetric spatial pyramid pooling (ASPP) layers, and (vii) InceptionNet-v4 [33].

Baselines. The achieved performance of these models with BrickDL is compared against three baselines: (i) **cuDNN**, (ii) **TorchScript**, and (iii) **TensorFlow+XLA**. The cuDNN baseline is a set of C++ benchmarks implemented with tiled cuDNN API calls for the evaluated models. This comparison analyzes the benefits of merged execution with bricks in BrickDL, which also calls cuDNN at the brick level. Additionally, BrickDL is compared against the PyTorch and TensorFlow implementations of these models, which are respectively optimized for end-to-end inference with the TorchScript PyTorch JIT compiler and the XLA compiler. Graph-level optimizations including operator fusion are enabled in both of them. TorchScript and TensorFlow XLA are used as baselines for BrickDL to form a comparison against highly optimized versions of these models with these frameworks.

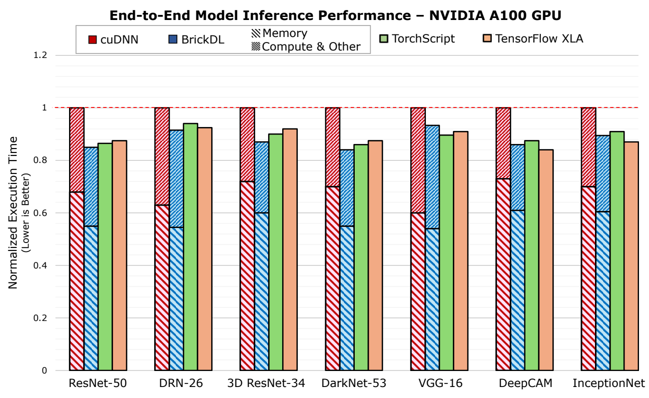

Figure 7: Performance of BrickDL relative to cuDNN, TorchScript, and TensorFlow XLA for the seven models. BrickDL attains a 9-17% speedup over the cuDNN baseline. The vertical bar for BrickDL is partitioned into fraction of time spent on compute and data movement.

Performance Evaluation. Figure 7 shows the execution time of BrickDL compared to TorchScript and TensorFlow XLA, relative to the cuDNN baseline. The performance of BrickDL is plotted for the best merged execution strategy among padded and memoized bricks for all subgraphs in each model, guided by the performance model. It also factors in the cost of creating bricks, which is minimal. The vertical bars of cuDNN and BrickDL are partitioned into the fraction of execution time for DRAM transfers, and compute and other tasks. The DRAM transfer time is calculated using the memory bandwidth and the number of DRAM transactions measured from Nsight. The execution rate for DRAM transactions (txn) is given by

$$R_{txn} = \frac{Bandwidth\ (TB/s)}{Transaction\ Size\ (B/txn)} txn/s$$

where $Bandwidth$ is the measured HBM bandwidth. $TransactionSize$ is 32 bytes for DRAM read and write transactions on NVIDIA A100 GPU. The total time for DRAM transactions is calculated as $T_{DRAM} = N_{txn}/R_{txn}$ where N_{txn} is the number of DRAM transactions collected from the hardware counter, and an R_{txn} of 46M txn/s. The `Compute & Other` time in Figure 7 is calculated as the difference of T_{DRAM} from the total execution time.

For all seven models evaluated, BrickDL consistently outperforms the cuDNN baseline and, in most cases, TorchScript and TensorFlow XLA. All models present reduced DRAM movement with BrickDL compared to cuDNN, which primarily contributes to the observed speedups. Deeper models benefit even better from BrickDL, with the ability to merge layers in more subgraphs. The highest performance gains are observed with DarkNet-53, with a speedup of 17.4% over cuDNN, 5.2% against TorchScript, and 6.7% versus TensorFlow XLA. BrickDL reduces DRAM transfer time by 16.5% from cuDNN, for DarkNet-53. The results demonstrate the benefits and applicability of merged execution with bricks across a breadth of CNN models with different types of convolutions (2D, 3D, dilated, strided, transposed), and diverse model structures (residual blocks, Inception modules, encoder-decoder layers).

4.3 Modeling Atomics and Compute Time

To isolate the cost of atomic operations and computation of bricks from the total execution time in Figures 8, 10, and 11, we model them using synthetic microbenchmarks.

4.3.1 Modeling Atomic Execution Time. Nsight Compute does not include the metric for the execution time of atomic operations. It only has a counter for the number of atomic transactions. We hence implement a simple synthetic microbenchmark to estimate the execution time of atomic transactions. This microbenchmark creates a 32×64K sized array (i.e., 64K number of 32-byte sized cache lines), which is partitioned into thread blocks, and each individual thread per word in a cache line executes a Compare-and-Swap (CAS) atomic operation. The reason for allocating each thread to different cache lines is to ensure there are no conflicts, assuming that the GPU hardware does not automatically optimize for atomic operation coalescing. This allocation creates a large one-to-one mapping between the threads in the kernel and cache lines in this array. A loop executes, where every thread performs 10^6 atomic operations, each to its own private cache line. The total execution time T of this loop structure is measured. For a number of threads N, the rate of execution of atomic operations is given by $R = \frac{N \times 10^6}{T}$ *atomics per second.* The time per atomic can be deduced from this as $T_{atomic} = R^{-1}$ *seconds per atomic.* The number of atomic transactions N_{atomic} is obtained from the hardware counter, and the total execution time for the atomic operations is calculated as $N_{atomic} \times T_{atomic}$. For the A100 GPU, the benchmark suggests the time for one atomic operation to be $T_{atomic} = 87.45\ ns$.

4.3.2 Modeling Compute Time. We implement a second synthetic microbenchmark to determine the time required for computing a brick, assuming perfect overlap with DRAM transactions. This microbenchmark creates an array of independent bricks allocated in shared memory. In the CUDA kernel, every thread block gets one brick, and it does 10^6 3D convolution cuDNN calls to this array, iterated over 1000 times in a loop (so a total of 10^9 cuDNN calls). The execution time of this benchmark is measured, which yields the execution rate for 10^9 cuDNN invocations per second. This metric is inverted to obtain the time per cuDNN call (T_{brick}). The total number of bricks called is multiplied by T_{brick} to get the total compute time, assuming the DRAM transfer time is hidden. Using this benchmark, we obtain $T_{brick} = 6.72\ \mu s$ as the execution time for an $8 \times 8 \times 8$ brick with a $3 \times 3 \times 3$ convolution filter.

4.4 Case Study: ResNet-50

Figure 8 shows the execution time of seven subgraphs from ResNet-50 with merged execution using *padded bricks* and *memoized bricks*, relative to cuDNN. The analysis for these subgraphs is performed with a consideration that compute time perfectly overlaps with DRAM transfers. Hence, `memory` and `computation` time are plotted side-by-side for each case. `Idle` time (shaded pink) is the time the memory subsystem is idle, calculated as the difference between the total execution time and the measured DRAM time (as described in Section 4.2). The `compute` and `atomic` execution time (modeled as in Section 4.3), are stacked on bars for `computation` in each case. The `Other` time (shaded gray) is the difference between the total execution time and the modeled compute + atomic time, which denotes time spent on tasks such as recursion, synchronization, data manipulation, stalls waiting for memory, etc. Inspired by the 3C's cache model [18], the atomic time is further split based on the number of compulsory atomic operations and conflicting atomic operations. Two compulsory atomics are required per brick, and the number of conflicting (failed) atomics is the difference between total atomics executed and compulsory atomics. The cuDNN baselines in Figures 8–11 are implemented with tiled calls to cuDNN, and the time to compute each tile is modeled using the same methodology as for bricks described in Section 4.3.2.

In Figure 8, we observe both padded and memoized bricks outperforming the cuDNN baseline for each subgraph. Both merged execution approaches considerably cut down DRAM movement compared to cuDNN in all subgraphs. Merged execution is more profitable in initial subgraphs with large layers, which offer opportunities to improve data locality on GPUs. Padded bricks performs better than memoized bricks in subgraphs 1 and 2, due to extra overhead from atomic operations on large layers in those subgraphs. However, in the subsequent subgraphs 3-7, the growth in net data due to padding, $\Delta > 15\%$, and it is favorable to use memoized bricks for those cases where a lesser fraction of time is spent on atomics.

To understand the data movement effects in Figure 8, Figure 9 compares the number of global memory, L2 cache, and DRAM transactions for padded and memoized bricks relative to cuDNN, for each subgraph. Merged bricks with padded bricks in subgraph 1 reduces

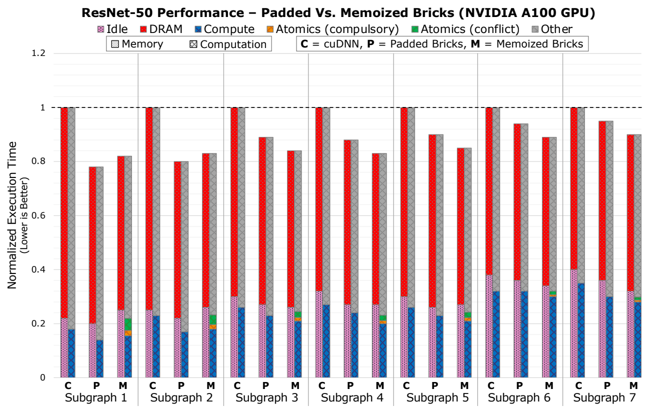

Figure 8: Comparing the speedup of padded bricks and recursive memoized bricks over the cuDNN baseline for the 7 subgraphs from ResNet-50. Padded bricks outperforms memoized bricks for Subgraphs 1-2, while memoized bricks performs better in the other subgraphs. All versions assume perfect overlap between memory and compute.

DRAM transactions the most, by 21%. Global memory transactions are a proxy for L1 cache transactions. Generally, techniques that improve data locality should decrease DRAM transactions at the cost of increased L2 transactions. Ideally, L1 transactions should remain constant, assuming there is no redundancy or overfetch. We observe merged execution decreases DRAM transactions and increases L2 transactions as expected, but also tends to increase L1 transactions due to overfetch from padding bricks — unlikely to be a performance impediment as L1 bandwidth is far greater than DRAM bandwidth on GPUs. Reduction in the net number of DRAM transactions at the expense of increased L1 and L2 transactions yields performance gains with merged execution of bricks.

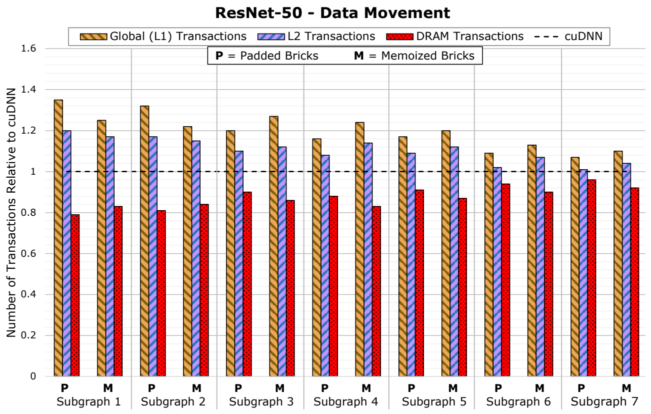

Figure 9: Comparing the data movement metrics of different subgraphs for ResNet-50. Both padded and memoized bricks show reduced DRAM transactions by trading them with faster L1 and L2 transactions.

In many cases, the ratio between the three data movement metrics for the different subgraphs shown in Figure 9 are of comparable magnitude (one reads as much data from DRAM as the L2). This suggests that these implementations basically stream the data through the memory hierarchy – from global memory → L2 cache → L1 cache → register files, which gives the benefit of reduced latency but at the cost of bandwidth filtering. This implies there is more scope for improving L1 and L2 cache locality.

4.5 Microbenchmarking Merged Execution Optimizations with Bricks

To analyze the performance characteristics (DRAM data movement, atomic operations, and compute time) that can incentivize or penalize padding and memoization, we construct two microbenchmarks to evaluate the effect of varying graph partitions and brick sizes.

4.5.1 Performance With Varying Subgraph Size. Partitioning DNN graphs into subgraphs of different sizes can have a considerable effect on the performance of padded and memoized bricks approaches. We characterize this with a six-layer microbenchmark as a proxy for a CNN graph with a chain of six convolution operations. The first layer is a $112 \times 112 \times 112$ 3D convolution operation with 64 channels and stride = 0, padding = 0, dilation rate = 1, and the subsequent five layers are computed accordingly. We use $8 \times 8 \times 8$ bricks and block only along the spatial dimensions. We implement both the padded bricks and memoized bricks optimizations by merging a sequence of 2, 3, 4, and 6 layers in this microbenchmark.

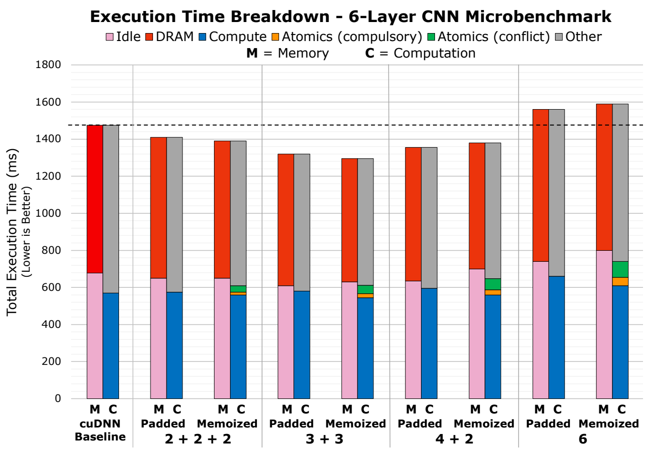

Figure 10: Breaking down the execution time of the different merged execution configurations for the six-layer proxy benchmark. For example, $2+2+2$ denotes the six-layer graph partitioned into 3 subgraphs, each with 2 merged layers; 6 implies that all the six layers are merged. The dotted horizontal line indicates execution time of the cuDNN baseline.

Figure 10 compares with cuDNN baseline the performance of different graph sizes partitioned from the 6-layer CNN graph. Using the same methodology described in Section 4.4, we measure DRAM time and model atomic and compute time. Among all merge configurations, $3+3$ with memoized bricks optimizations performs the best with a 12% speedup over the cuDNN baseline, reducing DRAM transfer time by 16.2%. Merged execution with all six layers (or more) leads to a significant slowdown with both padded bricks and memoized bricks due to excess data movement with padding and the overhead from many atomic operations in a large merged region of six layers. Merged execution in a subgraph of 2 layers is not beneficial due to frequent synchronizations and reduced parallelism. The compute time for padded bricks is higher than that for memoized bricks due to additional computations performed on the enlarged halo regions. The overhead from padded data, as well as compulsory and conflict atomic operations, increases with more layers in the subgraph, which could be due to clustered thread blocks that get totally overlapped with more merged layers.

4.5.2 Performance with Varying Brick Size. The microbenchmark in section 4.5 was implemented with a fixed brick size of 8^3. However, brick size is adjustable and can impact performance based on varying parallelism, additional data movement due to padding, and the overhead of atomic operations. To characterize this effect, we implement a CNN proxy microbenchmark with three back-to-back convolutional layers. The first layer is a $224 \times 224 \times 224$ 3D convolution operation with 64 channels, and the subsequent layers are computed accordingly. We evaluate this microbenchmark with brick sizes of 4^3, 8^3, 16^3, and 32^3 for padded bricks and memoized bricks, and we block along the spatial dimensions only. The three convolutional layers are always merged in each of these cases.

Figure 11 shows the execution time breakdown of the three-layer microbenchmark for varying brick sizes with padded and memoized bricks. Implementations with bricks of size 4^3 perform the worst due to the overhead of additional padded data and increased atomic operations for padded bricks and memoized bricks, respectively. The 32^3 bricks perform poorly due to coarse-grained parallelism with large bricks, which are unsuitable for GPUs. The most performant version is with 16^3 bricks for memoized bricks, providing a speedup of 13.5% over the cuDNN baseline. This implementation reduces the DRAM transaction time by 17.8% compared to cuDNN.

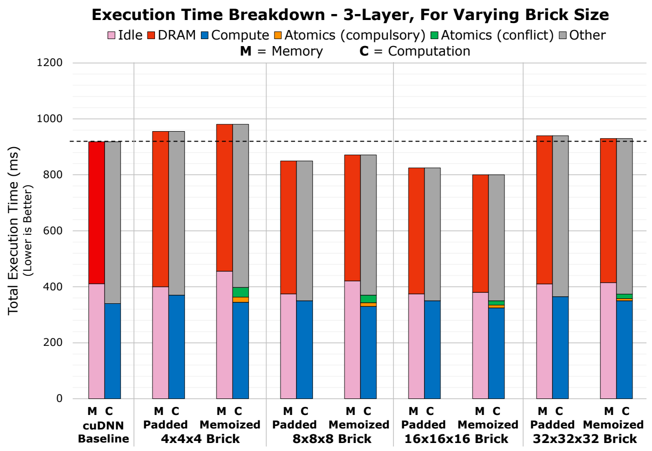

Figure 11: Breaking down the execution time of the different merged executions for the three-layer proxy benchmark as a function of brick size.

For the above microbenchmarks, though subgraphs of 3 layers and 16^3 bricks with memoized bricks perform the best, choosing the optimal subgraph and brick sizes, and merged execution approach depends on the problem specifications and hardware characteristics, which motivates the performance models in Section 3.3.

5 RELATED WORK

In this section, we discuss prior work in optimizing deep learning workloads, including libraries, compilers, and graph-level optimizing frameworks. We also discuss related optimizations for HPC applications. These are compared to BrickDL.

5.1 DNN Libraries and Compilers

Vendor libraries such as NVIDIA cuDNN, CUTLASS, AMD MIOpen, and Intel OneDNN provide highly tuned, optimized implementations for DNN primitives. Popular deep learning frameworks TensorFlow, PyTorch, and JAX leverage these off-the-shelf libraries along with hand-crafted kernels for commonly used DNN operations. NVIDIA's TensorRT SDK [1] and Meta's AITemplate Framework [44] provide manually-optimized routines for deep learning inference.

Given the recent explosion in diverse AI accelerators and modularization of DNN models, several domain-specific compilers have been developed, targeted at performance portability and productivity that generate high-performance kernels for individual operators in DNN graphs [6, 10, 31, 34, 36, 55, 57]. TorchScript [2], a component of the PyTorch JIT compiler, transforms DNN models into optimized and serialized format for inference through Python introspection or tracing. TensorFlow XLA [29] is a framework-agnostic JIT compiler that generates optimized code specific to the underlying hardware. The TVM compiler [6] uses the Ansor [54] auto-scheduler to generate optimized schedules representing different implementations, employing an autotuning search over a vast configuration space of kernels. Similar to BrickDL, Triton [36], Graphene [14], and FreeTensor [34] express and analyze operators at a finer granularity. In contrast to these systems, BrickDL employs the fine-grain blocked brick data layout that is better suited to DNN workloads, optimizing their data residency on GPUs.

5.2 Graph-Level Optimizations for DNN

Graph compilers perform inter-operator optimization of DNN graphs such as operator fusion, using graph rewriting rules that substitute subgraphs with pattern matching, and by using global passes that optimize the entire graph, for e.g., efficient memory scheduling. TASO [20], PET [37], and TENSAT [46] automatically generate graph substitutions using pre-defined rules and operator sets to perform graph-level optimizations. Rammer [24] and IOS [11] exploit parallelism between independent operations at the graph level and perform efficient inter-operator scheduling, in addition to optimizing individual operators. Astitch [56] performs fusion of memory-intensive operators to optimize data movement. Apollo [49] and DNNFusion [27] fuse memory-bound and compute-bound operations using rule-based algorithms but cannot fuse a chain of compute-intensive operations. Other compilers (TVM, XLA [29], TorchScript) and libraries (TensorRT, cuDNN) enable the kernel fusion of DNN primitives (e.g., convolution) with element-wise operations (e.g., ReLU). Bolt [43] can fuse back-to-back convolutions but requires subsequent convolutions in the chain to be unit-strided and point-wise (1×1 filter) without padding. BrickDL overcomes the limitations of operator fusion with an alternate approach of inter-layer merged execution of convolutions. Merged execution, when coupled with these existing graph-level optimizations, can further optimize the performance of deep learning frameworks.

Pertinent to merged execution in BrickDL, Xu et al. [45] partition CNN layers to fit large activations in a single GPU's memory for model training with checkpointing. Approaches similar to merged execution have also been applied to design accelerators for DNN inference [3, 53]. Related to our work, DistDL [17] and DistConv [12] implement efficient halo exchanges with spatial model parallelism for training CNN models. While BrickDL currently supports inference only, merged execution can be extended to enable fine-grained hybrid model parallelism for distributed DNN training.

5.3 Locality and Parallelism for Stencils

The convolution operators in deep learning exhibit a stencil pattern, where an output point is calculated as a weighted sum of neighboring input points. Stencil computations also appear in the solution of partial differential equations, for which there is significant prior work. Stencils are memory-intensive, yet often exhibit nearest neighbor data reuse. So, achieving data locality is imperative in minimizing data movement and maximizing performance.

Many optimizations for stencils achieve data locality by *restructuring execution order* so that data reuse occurs temporally while data resides in fast memory. In *space-time tiling* [8, 25, 26], the N+1-dimensional space (array dimensions plus time) hyperspace is tiled into hyper-parallelpipeds or trapezoids. By constraining execution to proceed from one parallelpiped to the next, cache working sets can be tailored to be less than cache capacity, thereby maximizing data locality. In *recursive space-time tiling* (also known as *cache oblivious algorithms*) [13, 21], the N+1-dimensional space is recursively cut in either space or time until a cutoff is reached at which point execution can proceed. *Wavefronts* [5, 9, 40] create a pipeline of execution. Effectively, one can tile the iteration space into 2.5D planes that proceeds through the remaining dimensions of space and time. One can choose to parallelize within the spatial dimension of the wavefront [39] or in the time dimension of the wavefront [38] or both. *Cache oblivious wavefronts* attempt to extract the both of best worlds [7, 35]. In essence, they perturb the cache oblivious recursion algorithm, incentivizing the execution of temporally concurrent independent tiles. These approaches to execution order transformations for stencils achieve locality by restructuring the computation to operate on logically neighboring data in chunk sizes that better utilize the memory hierarchy. Recursive approaches exhibit significant overhead from global synchronizations, and other approaches result in complex code. All require careful tuning of the sizes of data chunks to maximize reuse and parallelism in a deep and tapered cache hierarchy.

An alternative approach is to *reorganize data into a blocked physical representation*, such that logically neighboring data is stored contiguously in memory. Stencil optimization that uses such *fine-grain data blocking* include YASK [47], Briquettes [19], RTM on the Cell processor [4], and optimized code generation by Zhao et al. [50–52] that target large, higher-order, compute-intensive stencils. This concept has also been applied to optimize stencils and FFT in a 6D phase-space tokomak simulation fusion code [30]. As compared to execution reordering, with fine-grained data blocking, the logically neighboring data chunks are part of the same or adjacent address streams, reducing unnecessary data movement associated with long strides that span multiple vectors, cache lines, or pages. Fine-grained data blocks also reap benefits from hardware features that favor spatial reuse, e.g., prefetchers.

BrickDL incorporates both *execution reordering* and *data reorganization* for inter-layer *merged execution* with *bricks* in DNNs. This approach can also be extended to optimize reuse along spatial and temporal dimensions of stencil computations.

6 DISCUSSION AND CONCLUSIONS

This paper has presented BrickDL, which to the best of our knowledge is the first optimization system for DNNs that exploits reuse across arbitrary chains of convolutions, with merged execution of layers on fine-grained data blocks. Instead of generating fused kernel code, BrickDL invokes each operator's kernel at the fine-grained granularity of bricks and reschedules their order of execution. We explored two approaches to mitigating data dependences in parallel execution — padded bricks (halo regions or ghost zones in HPC parlance) and memoization with dynamic runtime (locks on blocks of the activation layers) — both of which demonstrated improved performance and reduced data movement on GPUs. As compared to prior work on optimizing stencil optimizations, BrickDL's merged execution uniquely combines data reorganization with execution order and parallelization optimizations tailored to fine-grained blocks and addresses the changing problem size associated with operators in DNN graphs.

While we do not propose BrickDL as a replacement for state-of-the-art DNN systems, BrickDL's optimizations are beneficial and make a firm case for integrating them with these deep learning systems for better performance. In general, the bricks can be integrated with compilers that support domain-specific frameworks and expose data layouts as abstractions, e.g., LLVM MLIR compiler infrastructure [23] at the DLTI dialect.

Analysis with the microbenchmarks points to more opportunities for data movement reduction: replacing cuDNN library calls with code generation for DNN primitives along with optimizations such as wavefront parallelization [41] and performing skewed cuts across layers [42]. Similarly, there should be hardware-software opportunities to accelerate memoization via non-blocking computations and locality aware ordering.

BrickDL's optimizations also apply to the sequences of computations on structured grids found in HPC codes, including layered computations such as multi-grid and adaptive mesh refinement. Indeed, we believe optimizations such as these centered around data layouts will play an increasingly important role in future HPC and AI systems, where managing data movement is paramount.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] 2023. *NVIDIA TensorRT SDK.* https://developer.nvidia.com/tensorrt
[2] 2023. *PyTorch TorchScript.* https://pytorch.org/docs/stable/jit.html
[3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.
[4] Mauricio Araya-Polo, Félix Rubio, Raúl De la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. 2009. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Scientific Programming* 17, 1-2 (2009), 185–198.
[5] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Mary Hall. 2014. Converting Stencils to Accumulations Forcommunication-Avoiding Optimizationin Geometric Multigrid. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*. 9–16.
[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[7] Rezaul Chowdhury, Pramod Ganapathi, Yuan Tang, and Jesmin Jahan Tithi. 2017. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 339–350.
[8] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review* 51, 1 (2009), 129–159.
[9] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–12.
[10] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 370–384.
[11] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. IOS: Inter-Operator Scheduler for CNN Acceleration. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 167–180.
[12] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. 2019. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 210–220.
[13] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*. 361–366.
[14] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 302–313.
[15] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. 2017. Learning spatio-temporal features with 3d residual networks for action recognition. In *Proceedings of the IEEE international conference on computer vision workshops*. 3154–3160.
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[17] Russell J. Hewett and Thomas Grady. 2020. *DistDL: Distributed Deep Learning*. https://doi.org/10.5281/zenodo.3990527
[18] Mark D Hill and Alan Jay Smith. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput.* 38, 12 (1989), 1612–1630.
[19] Jagan Jayaraj. 2013. *A strategy for high performance in computational fluid dynamics*. University of Minnesota.
[20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.
[21] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2006. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*. 51–60.

[22] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. 2018. Exascale Deep Learning for Climate Analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 649–660.

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain-specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[25] John McCalpin and David Wonnacott. 1998. *Time skewing: A value-based approach to optimizing for memory locality*. Technical Report. Rutgers University.

[26] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[27] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[28] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[29] Amit Sabne. 2020. XLA: Compiling machine learning for peak performance. (2020).

[30] Benjamin Sepanski, Tuowen Zhao, Hans Johansen, and Samuel Williams. 2022. Maximizing Performance Through Memory Hierarchy-Driven Data Layout Transformations. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 1–10.

[31] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems* 3 (2021), 208–222.

[32] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[33] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[34] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. 2022. FreeTensor: a free-form DSL with holistic optimizations for irregular tensor programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 872–887.

[35] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A Chowdhury. 2015. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 205–214.

[36] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[37] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 37–54.

[38] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. IEEE, 579–586.

[39] Samuel Williams, Dhiraj D Kalamkar, Amik Singh, Anand M Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. 2012. Optimization of geometric multigrid for emerging multi- and manycore processors. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[40] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. 2006. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers*. 9–20.

[41] Michael Wolfe. 1986. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15 (1986), 279–293.

[42] David Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. IEEE, 171–180.

[43] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems* 4 (2022), 204–216.

[44] Bing Xu, Ying Zhang, Hao Lu, Yang Chen, Terry Chen, Mike Iovine, Mu-Chu Lee, and Zhijing Li. 2022. *AITemplate*. https://github.com/facebookincubator/AITemplate

[45] Yufan Xu, Saurabh Raje, Atanas Rountev, Gerald Sabin, Aravind Sukumaran-Rajam, and P Sadayappan. 2022. Training of deep learning pipelines on memory-constrained GPUs via segmented fused-tiled execution. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 104–116.

[46] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268.

[47] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK—Yet Another Stencil Kernel: A framework for HPC stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE, 30–39.

[48] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. 2017. Dilated residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 472–480.

[49] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems* 4 (2022), 1–19.

[50] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. 2019. Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–44.

[51] Tuowen Zhao, Mary Hall, Hans Johansen, and Samuel Williams. 2021. Improving communication by optimizing on-node data movement with data layout. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 304–317.

[52] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. 2018. Delivering Performance-portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 59–70.

[53] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[55] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.

[56] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.

[57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.