Based on the provided sources, here is a detailed explanation of the chapters and topics you requested, organized by week:

Week 16: Deep Learning and Convolutional Neural Networks (Chapter 16)

This chapter serves as an application case study on deep learning, a branch of machine learning using artificial neural networks 1 . Machine learning involves training application logic based on experience from datasets, and deep learning uses neural networks for this purpose 1 . The goal is to understand and develop kernels for deep learning approaches to machine learning tasks 2 ... .

- **Convolutional Neural Networks (CNNs): An Introduction**
  - CNNs are a popular deep learning algorithm with a high compute-to-memory access ratio and high levels of parallelism, making them well-suited for GPU acceleration 4 .
  - The computation in a convolutional network is organized as a sequence of layers 5 . These layers process inputs and outputs, often called feature maps or features 5 .
  - LeNet-5 is a historical example of a CNN, composed of three types of layers: **convolutional layers, subsampling layers, and fully connected layers** 6 . These are still key components of modern neural networks 6 .
  - The input to a CNN is typically an image 5 . The last layer computes an output, often representing probabilities for different categories (e.g., digits in handwriting recognition) 5 .
  - **Convolutional layers** generate output feature maps from input feature maps 7 . Each pixel in an output feature map is produced by performing a **convolution** between a small local patch of pixels from the previous layer's feature map and a set of weights called a **filter bank** 7 . The convolution result is then fed into an activation function 7 .
  - An output feature map can be the **sum of convolution results** from corresponding patches in *all* input feature maps 8 ... . Different pairs of input and output feature maps use different filter banks 8 .
  - Input feature maps are often stored as a 3D array X[C, H, W], where C is the number of input feature maps, H is height, and W is width 10 . Output feature maps are stored as Y[M, H_out, W_out], where M is the number of output feature maps and H_out, W_out depend on H, W, and the filter size K 11 . Filter banks are stored as W[M, C, K, K] 11 . Filter bank W[m, c, _, _] is used when using input feature map X[c, _, _] to calculate output feature map Y[m, _, _] 11 .
  - **Subsampling layers (pooling layers)** reduce the size of image maps by combining pixels 12 ... . For instance, a 2x2 neighborhood in the input feature map might be averaged to form one pixel in the output feature map 13 . The number of output feature maps in a subsampling layer is the same as its input, but dimensions are reduced 14 . Subsampling layers often include an **activation function** 15 .
  - **Fully connected layers** are layers where every output is a function of every input 16 . The process of evaluating outputs from inputs in a fully connected layer is a **matrix-vector multiplication** 17 . These layers become expensive for large inputs/outputs due to the size of the weight matrix 17 .

    - CNNs reduce the cost of fully connected layers by reducing the number of inputs each classifier (like a perceptron) takes and **sharing the same weights** across classifiers in convolutional layers **18** . This allows for many classifiers (large M) without excessive weights **19** .
- **Kernel for Backpropagation and Forward Propagation**
    - **Forward Propagation:** The computation goes through layers sequentially **5** . Sequential C code for the forward pass of a convolutional layer involves nested loops iterating over output feature maps (m), output pixels (h, w), input feature maps (c), and filter weights (p, q) **9** **...** .
    - The innermost loops (c, p, q) perform the 3D convolution for one output pixel **12** . These loops can be kept serial to avoid atomic operations if not needed **22** .
    - The four outer loop levels (n for batch, m for output feature map, h for output pixel row, w for output pixel column) offer significant parallelism: N * M * H_out * W_out parallel iterations **23** . This high degree of parallelism makes it excellent for GPU acceleration **23** .
    - A CUDA kernel can be designed with threads mapped to compute one element of one output feature map pixel **24** . 2D thread blocks can compute a tile of pixels in one output feature map **24** . Blocks can be organized in a 3D grid to capture batch (N), output feature map (M), and tile location (h-w) parallelism **24** **...** .
    - One proposed thread organization uses grid dimensions for M (X), tile location (Y), and N (Z) **25** . The tile location in the Y dimension combines vertical and horizontal tile indices **26** **...** .
    - The host code for launching such a kernel sets grid dimensions based on M, the number of tiles (H_grid * W_grid), and N **28** .
    - **Forward Kernel Code Example:**
    - *This kernel code iterates over input channels (c) and filter dimensions (p, q) serially for each output pixel (n, m, h, w) assigned to a thread* **27** **...** . This kernel consumes significant global memory bandwidth **31** . Optimizations like constant memory caching and shared memory tiling can reduce global memory traffic **31** .
    - **Backpropagation:** Training CNNs uses methods like stochastic gradient descent and backpropagation **32** . Backpropagation calculates gradients of the loss function with respect to parameters (@E/@w) and inputs (@E/@x) **33** **...** .
    - Calculating **@E/@x** for a convolutional layer involves a sum of "backward convolution" with the transposed filter bank (@E/@y convolved with wT) **35** **...** . This is because each input element contributes to multiple output elements through weights, and its gradient receives contributions from the gradients of those output elements **35** **...** .
    - Calculating **@E/@w** for a convolutional layer involves accumulating gradients over all pixels in the corresponding output feature map **38** . This is essentially a convolution between the input feature map and @E/@y **38** .
    - **Backward @E/@x Kernel (C code):**

    - *This code snippet header indicates a function for calculating input gradients.* **39** **...** . It assumes @E/@y for the current layer is available and space for @E/@x is allocated **37** **...** .
    - **Backward @E/@w Kernel (C code):**
    - *This code snippet header indicates a function for calculating weight gradients.* **38** **...** . It initializes the gradient accumulation to zero and then iterates to sum contributions **40** .
- **Matrix Unroll and Using Matrix Multiplication Instead of Convolution**
    - A convolutional layer can be represented as an equivalent matrix multiplication operation (GEMM) to use highly efficient linear algebra libraries like cuBLAS **41** .
    - The central idea is **unfolding and duplicating input feature map pixels** such that all elements needed to compute one output feature map pixel are stored as one sequential column in a new matrix **42** **...** .
    - The filter banks are represented as a matrix where each row contains all weight values needed to produce one output feature map **44** **...** . This matrix has dimensions M (output feature maps) by C * K * K (total weights per output pixel) **44** **...** . There is no duplication of weights in this matrix **45** .
    - The output Y is computed by multiplying the filter bank matrix (W) by the expanded input matrix (X_unrolled) **45** .
    - **Data Layouts:**
        - Input X: N x C x H x W array **46** .
        - Output Y: M x (H_out * W_out) array (produced by multiplication) **47** .
        - Filter W: M x C x K^2 array **47** .
        - Expanded Input X_unrolled: (C * K * K) x (H_out * W_out) matrix **48** **...** . Each column contains input pixels needed for one output pixel **42** **...** . Each row corresponds to a specific (c, p, q) position relative to the output pixel location **50** .
    - Preparing the X_unrolled matrix can be complex and increase the input size significantly (up to K^2 times) **47** . A sequential C function to generate X_unrolled iterates through input channels (c), filter patch dimensions (p, q), and output dimensions (h, w) **48** **...** . This process is easily parallelizable **52** .
    - **CUDA Unroll Kernel:** Each thread can be responsible for gathering K*K input elements from one input feature map for one element of an output feature map **52** . The total number of threads would be (C * H_out * W_out) **53** . Threads can be organized using a 1D block and deriving multidimensional indices **53** .
    - **Unroll Kernel Code Example:**
    - *This kernel maps each thread to a specific (c, w_unroll) position in the X_unrolled matrix (equivalently, a (c, h_out, w_out) combination) and has it write the K*K elements from the input X needed for that column segment* **54** **...** . Adjacent threads write adjacent elements in a row of X_unrolled **50** .
    - Implementing convolutions with matrix multiplication can be very efficient because it leverages highly optimized GEMM libraries **56** . However, materializing the expanded X_unrolled matrix in global memory

can be costly in terms of space and bandwidth **57** . Optimized libraries like CUDNN avoid this by generating and loading X_unrolled data directly into on-chip memory instead of storing it globally first **58** **...** .

Week 15: Sparse Matrix Computation (Chapter 14, excluding 14.5, 14.6)

This chapter presents sparse matrix computation as an important parallel pattern **60** . Sparse matrices have a large number of zero elements and are common in applications modeling complex phenomena **60** .

- **Sparse Matrix Vector Multiplication (SpMV)**
  - SpMV is a key computation involving sparse matrices.
  - Due to the large number of zeros, **compaction techniques** are used to reduce storage, memory accesses, and computation on zero elements **60** .
  - Different **storage formats** are used, and they impact performance considerations like storage cost, computation, memory access efficiency (especially coalescing), and load balance **61** .
- **COO Format**
  - COO (Coordinate List) format stores only the nonzero elements of a sparse matrix **61** .
  - It typically uses three arrays: one for the nonzero values, one for their row indices, and one for their column indices **62** .
  - A simple SpMV kernel using COO assigns one thread to each nonzero element **62** .
  - **A simple SpMV kernel with COO:** Multi-threaded access to the output vector requires using **atomic operations** for accumulation, as multiple threads might update the same output element **62** .
  - Memory access efficiency is often poor with simple COO kernels because adjacent threads, processing nonzeros that might be far apart in the matrix, access non-adjacent memory locations when writing to the output vector and sometimes reading input vectors **62** .
- **CSR Format**
  - CSR (Compressed Sparse Row) format is another common sparse matrix storage format **63** **...** .
  - It groups the nonzero elements by row **64** .
  - The sources mention that a CSR representation is needed for vertex-centric push implementations in graph traversal because it provides accessibility to outgoing edges (nonzeros of a given row) **65** .
  - The sources also state that code can be implemented to convert from COO to CSR using primitives like histogram and prefix sum **66** .
  - The sources do *not* provide kernel code specifically for performing SpMV using the CSR format within the provided excerpts, but mention CSR as a relevant format **63** **...** .
- **ELL Format**
  - ELL (ELLPACK) format is a sparse matrix storage format mentioned as improving memory coalescing **63** .

    - The sources list it as one of the formats for representing a sparse matrix 64 .
    - The sources do *not* provide kernel code specifically for performing SpMV using the ELL format within the provided excerpts, but highlight its benefit for memory coalescing 63 .
- **Memory Coalescing and Other Considerations**
    - Memory access efficiency is a key design consideration for sparse matrix computations 61 .
    - In the simple COO kernel, memory accesses are **not coalesced** when adjacent threads read input elements or write to the output vector, leading to poor memory bandwidth utilization 62 . For example, adjacent threads processing A, A 67 , A 67 ... in a matrix might write to widely separated locations in the output vector 62 .
    - The ELL format is mentioned specifically as improving memory coalescing 63 .
    - Other considerations include storage cost, the amount of computation performed on nonzero elements, and load balance 61 . Different formats compare differently on these considerations 61 .

Week 14: Sorting and Merging (Chapters 12 excluding 12.7, 12.8; Chapter 13 excluding 13.5, 13.6, 13.8)

- **Radix Sort (Chapter 13, excluding 13.5, 13.6, 13.8)**
    - Radix sort sorts keys by distributing them across buckets based on individual digits 68 . This process is repeated for each digit, preserving order from previous digits 68 .
    - The iterations (for each digit) are performed sequentially 69 .
    - The opportunity for parallelization arises **within each iteration** 69 . The focus is on implementing a kernel that performs a single radix sort iteration 69 .
    - A key step in a radix sort iteration is finding each key's **destination index** in the output list 70 .
    - Threads can perform a boundary check, load a key, and extract the bit for the current iteration 70 .
    - Threads then **collaborate to perform an exclusive scan on the extracted bits** (0s and 1s) 71 ... . This exclusive scan (prefix sum) is crucial 72 .
    - The result of the exclusive scan array, at each position i, contains the number of elements with a value of 0 at the current bit position among elements 0 through i-1 72 . This allows calculating the destination index 72 .
    - The kernel code for a single radix sort iteration needs to handle potential barrier synchronization for the exclusive scan, possibly requiring sophisticated techniques or multiple kernel launches 71 ... .
    - **Radix Sort Iteration Kernel Code Example:**
    - *Note: The source excerpts for Chapter 13 only provide snippet 13.4 showing logic for finding the bit and calling scan 70 ... . The provided C code above is reconstructed based on the description in the source and is not directly from the source text, which assumes external functions like `ExclusiveScan` and `GetTotalZeros`.*
- **Merge Sort (Chapter 13, excluding 13.5, 13.6, 13.8)**

- Merge sort is mentioned as a parallel sort method that can be implemented using the parallel merge implementation **73** **...** .
- The sources do *not* provide a detailed algorithm or kernel code for parallel merge sort itself, only indicating that it relies on a parallel merge operation **74** .

- **Co-Rank Function and Parallel Merge (Chapter 12, excluding 12.7, 12.8)**
  - Merging involves combining elements from two sorted lists (A and B) into a single sorted list (C) **76** .
  - A sequential merge algorithm iteratively compares the current elements of A and B, taking the smaller one **77** .
  - For parallel merge, the work is divided among threads, with each thread responsible for producing a subarray of the output list C **78** .
  - The **co-rank function** is essential for this parallel approach **78** . For a given rank (index k) in the output list C, the co-rank function determines how many elements come from list A (i) and how many come from list B (j) such that A[i-1] and B[j-1] are the elements that would meet just before C[k] in a sequential merge **78** . In other words, C[k] is the minimum of A[i] and B[j] **76** **...** . The co-rank is essentially the pair (i, j) such that i+j=k **78** .
  - The co-rank function is implemented using binary search on one of the input arrays **80** . For a target rank k in C, it finds the index i in A such that A[i] is approximately the k-th element when A[i] is merged with B[k-i] **78** . The exact index i is found by checking if A[i] is less than B[k-i-1] and greater than or equal to B[k-i] (assuming appropriate boundary checks) **76** . The corresponding j is k-i **78** .
  - Each thread calls the co-rank function twice: once for its starting output index (k_curr) to find (i_curr, j_curr), and once for the next thread's starting output index (k_next) to find (i_next, j_next) **78** **...** . This determines the subarrays of A and B that the thread needs ((i_next - i_curr) from A and (j_next - j_curr) from B) to produce its output subarray C[k_curr] through C[k_next-1] **79** .
  - A basic parallel merge kernel (Fig 12.9 referenced) uses threads to calculate their start/end indices using the co-rank function and then calls a sequential merge function on the identified subarrays **78** **...** .
  - **Basic Merge Kernel Code Example (Conceptual, based on Fig 12.9 description):**
  - The basic kernel has poor memory access efficiency **82** . When threads call the sequential merge function, adjacent threads access non-adjacent memory locations for input and output subarrays **82** . Co-rank function accesses (binary search) are also irregular **80** .
  - A **tiled merge kernel** improves efficiency **83** **...** . Blocks collectively process larger subarrays **83** . Co-rank is called at the block level to find block-level subarray boundaries **83** .
  - Threads in a block **cooperatively load elements** of the block-level A and B subarrays into **shared memory** in a coalesced pattern **83** **...** .
  - Due to limited shared memory, processing is iterative **87** . In each iteration, threads load tiles from global to shared memory and then merge portions within shared memory **87** **...** .

    ◦ The tiled kernel achieves substantial reduction in global memory accesses for the co-rank function and makes global memory accesses for loading data **coalesced** 90 .

Week 13: Convolution on CUDA (Chapter 7)

Based on the provided sources, there is **no information available** regarding Convolution on CUDA from Chapter 7. The excerpts only include material from Chapter 10 onwards.

Week 12: Reduction and Prefix Sum (Chapters 10 excluding 10.8; Chapter 11 excluding 11.7)

- **Reduction on CUDA (Chapter 10, excluding 10.8)**
    ◦ Reduction is an important parallel pattern used in many data-processing applications 91 . It involves combining elements of an array into a single result (e.g., sum, minimum, maximum) 92 .

    ◦ Sequential reduction uses a loop to visit all elements 92 .

    ◦ Parallel reduction algorithms use the basic concept of **reduction trees** 92 >< 93 .

    ◦ A simple reduction kernel implements a reduction tree **within a single block** 93 . For N elements, it uses N/2 threads 93 .

    ◦ In each time step (iteration), threads add pairs of elements, producing partial sums 94 ... . The number of active threads halves in each step 94 .

    ◦ A `stride` variable is used for threads to locate the appropriate partial sums 96 . The stride doubles in each iteration (1, 2, 4, ...) 96 .

    ◦ Threads execute the same kernel code, but an `if-condition` based on `threadIdx.x` determines which threads are active and perform the addition 97 . Threads failing the condition are inactive 97 . Fewer threads are active as iterations progress 97 . Only thread 0 remains active at the last iteration 97 .

    ◦ `__syncthreads()` is used within the loop to ensure partial sums are written to memory before being read by active threads in the next iteration 97 ... .

    ◦ **Simple Sum Reduction Kernel Code Example (Based on Fig 10.6 description):**

    ◦ The simple kernel (Fig 10.6) results in a **high degree of control divergence** 98 . Active and inactive threads are mixed within warps 98 ... . For example, in the first iteration, half the threads are inactive 98 .

    ◦ An **improved reduction kernel** (Fig 10.9 referenced) aims to minimize control divergence 100 . It uses a different access pattern where threads process elements that are further apart initially 100 . This kernel still has an `if-statement` 100 . However, the pattern ensures that whole warps become inactive together, **eliminating control divergence within warps** for certain strides (e.g., warps 0 and 1 active, warps 2 and 3 inactive for a 128-thread block, warp size 32) 99 .

    ◦ Using **shared memory** further optimizes reduction by reducing global memory accesses 101 . The kernel (Fig 10.11 referenced) first loads data into shared memory, performs reduction in shared memory, and then thread 0 writes the final sum to global memory 101 . `__syncthreads()` is used before the loop to synchronize shared memory access 101 .

    - For large inputs beyond a single block's capacity, **hierarchical, segmented multiblock reduction** is used 102 . The input array is partitioned into segments, each processed by a different block 102 . Each block performs a reduction tree on its segment independently 102 . The results from each block are accumulated to the final output using an **atomic add operation** 102 **...** . Partitioning is done by assigning a segment based on the block index 104 .

- **Prefix Sum on CUDA (Chapter 11, excluding 11.7)**
    - Prefix sum (scan) is an important algorithm component for parallelizing many applications 105 .
    - Work efficiency is a key consideration in analyzing parallel algorithms 106 .
    - A sequential inclusive scan algorithm iterates through the input, adding the current element to the previous result to generate the next output element 107 . Its computational complexity is O(N) 108 .
    - Parallel segmented scan algorithms partition the input into segments, with each thread block performing a scan on a segment 108 **...** .
    - The **Kogge-Stone algorithm** is a parallel inclusive scan algorithm 109 . It involves multiple iterations where elements are updated by adding a value from a `stride` distance away 110 . The stride doubles in each iteration (1, 2, 4, ...) 110 **...** .
    - A Kogge-Stone kernel performs local scans on segments small enough for a single block 112 . Input elements are loaded into shared memory by threads collaboratively 111 **...** . Each thread is assigned to an element in the shared memory array (XY) 114 .
    - The core iterative calculation in the Kogge-Stone kernel updates elements in shared memory based on elements `stride` away 113 . `__syncthreads()` is used to synchronize between iterations 111 .
    - **Kogge-Stone Scan Kernel Code Example (Fig 11.3):**
    - *This kernel loads data to shared memory* 111 *, performs the scan in shared memory using a loop with doubling strides* 111 **...** *, synchronizes each step* 111 *, and writes results to global memory* 113 *.*
    - An **exclusive scan kernel** can be adapted from the inclusive scan kernel with minor adjustments to how input elements are loaded 115 **...** .
    - The **Brent-Kung algorithm** is another parallel scan algorithm, generally faster through the reduction tree phase but potentially slower due to SIMD inefficiencies 117 . It can also be adapted for exclusive scan 116 .
    - For higher work efficiency, a **coarsened scan** can be used, divided into three phases 118 : 1) each thread performs a sequential scan on its subsection, 2) threads collaborate to perform a scan on the last element of each section (using Kogge-Stone or Brent-Kung), and 3) results from the second phase are used to update the elements in each subsection 118 **...** .
    - For arbitrary-length inputs (millions/billions), a **hierarchical scan** approach is needed to consolidate results from different sections/blocks 121 **...** .
    - A hierarchical scan partitions the large dataset into sections processed by individual blocks 121 **...** . After initial block-wide scans (scan blocks), each element contains the accumulated value only within its

block **122** .

- These block scan results need to be combined by adding the sum of all preceding blocks' elements to each element of a given block **122** **...** .
- This consolidation can be implemented with **three kernels** **124** **...** :
  1. A kernel (like Kogge-Stone or Brent-Kung) performs the block-wide scan and writes the sum of each block to a separate array (S) **124** **...** .
  2. A second kernel performs a scan (e.g., Kogge-Stone or Brent-Kung with a single block) on the S array to get the cumulative sum of block sums **125** .
  3. A third kernel takes the cumulative block sums from the scanned S array and adds the appropriate sum to each element in the Y array (which still holds the block-local scan results) **125** **...** . Threads in a block add the sum of all previous scan blocks to the elements of their scan block **126** .
- Storing and reloading partially scanned results (scan blocks) to global memory between these kernels can affect performance **126** **...** .

I hope this detailed explanation drawing from the provided sources is helpful!