LANGUAGES AND COMPILERS FOR WRITING EFFICIENT HIGH-PERFORMANCE COMPUTING APPLICATIONS

A Dissertation Presented

by

ABHINAV JANGDA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2022

Robert and Donna Manning College of
Information and Computer Sciences

© Copyright by Abhinav Jangda 2022

All Rights Reserved

LANGUAGES AND COMPILERS FOR WRITING EFFICIENT HIGH-PERFORMANCE COMPUTING APPLICATIONS

A Dissertation Presented

by

ABHINAV JANGDA

Approved as to style and content by:

Arjun Guha, Chair

Emery Berger, Member

Marco Serafini, Member

Saeed Maleki, Member

James Allan, Chair of the Faculty
Robert and Donna Manning College of
Information and Computer Sciences

DEDICATION

For my maternal grandfather and grandmother, whose memories I will cherish forever.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Arjun Guha, for his support and guidance over the past five years. Your guidance in conducting research and presenting ideas greatly impacted my growth as an independent researcher. You taught me how to filter good ideas and continue to pursue them even when getting negative results. I am lucky to have had the opportunity to work with you and receive your guidance.

I would like to thank my research mentors, Emery Berger, Uday Bondhughula, Saeed Maleki, Marco Serafini, Madan Musuvathi, Todd Mytkowicz, and Olli Saarikivi. You have helped me become a better researcher, writer, and speaker.

I am thankful that the CICS administration staff has shielded me from the administrative burden. Leanne Leclerc and Eileen Hamel ensured that I enrolled in all credits every semester and was up to date with all the Ph.D. requirements. Lourie Downey ensured that I received the reimbursement for conference travel as soon as possible.

I am grateful for the friendship of Mimansa Jaiswal, Yokila Arora, Virat Shejwalkar, Mohit Yadav, Gaurav Anand, Anil Saini, Diptyaroop Maji, and Manish Motwani. I will miss our pre-pandemic lab conversations and movie nights with PLASMA Lab students Sam Baxter, Breanna Devore-McDonald, Joseph Spitzer, Donald Pickney, Amit Rawat, and Bobby Powers.

Words cannot express my gratitude to my family for their consistent help in fulfilling my ambitions. My successes have been possible only because of the countless sacrifices of my parents, Hari Datt and Neena Jangda. My elder sister, Tanvi Jangda, has always been there for me, even when we were on the other side of the planet.

Finally, I am deeply indebted to my maternal grandparents, Ramesh Sharma and Ratna Devi, for making countless memories I cherish every day.

ABSTRACT

LANGUAGES AND COMPILERS FOR WRITING EFFICIENT HIGH-PERFORMANCE COMPUTING APPLICATIONS

SEPTEMBER 2022

ABHINAV JANGDA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Arjun Guha

Many everyday applications, such as web search, speech recognition, and weather prediction, are executed on high-performance systems containing thousands of Central Processing Units (CPUs) and Graphics Processing Units (GPUs). These applications can be written in either low-level programming languages, such as NVIDIA CUDA, or domain specific languages, like Halide for image processing and PyTorch for machine learning programs.

Despite the popularity of these languages, there are several challenges that programmers face when developing efficient high-performance computing applications. First, since every hardware support a different low-level programming model, to utilize new hardware programmers need to rewrite their applications in another programming language. Second, writing efficient code involves restructuring the computation to ensure (i) regular memory access patterns, (ii) non-divergent control flow, and (iii) complete utilization of different programmer managed caches. Furthermore, since these low-level optimizations are known only to hardware experts, it is difficult for a domain expert to write optimized code for new computations. Third, existing domain specific languages suffer from optimization barriers in the language constructs that prevent new optimizations and hence, these languages provide sub-optimal performance.

To address these challenges this thesis presents the following novel abstractions and compiler techniques for writing image processing and machine learning applications that can run efficiently

on a variety of high-performance systems. First, this thesis presents techniques to optimize image processing programs on GPUs using the features of modern GPUs. These techniques improve the concurrency and register usage of generated code to provide better performance than the state-of-the-art. Second, this thesis presents NextDoor, which is the first system to provide an abstraction for writing graph sampling applications and efficiently executing these applications on GPUs. Third, this thesis presents CoCoNet, which is a domain specific language to co-optimize communication and computation in distributed machine learning workloads. By breaking the optimization barriers in existing domain specific languages, these techniques help programmers write correct and efficient code for diverse high-performance computing workloads.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

In the last decade, there has been an exponential increase in computationally intensive applications, such as web search, speech recognition, and face recognition. These applications run on high performance systems that contain several Central Processing Units (CPUs) and Graphics Processing Units (GPUs). For instance, speech recognition utilizes large neural networks that are trained on thousands on GPUs. These large systems can be programmed using low-level languages and APIs, such as NVIDIA CUDA [2] and AMD HIP [1] to program each GPU, and Message Passing Interface(MPI) [40] to program distributed systems. Since these low-level languages and APIs provide access to all features of GPUs and distributed systems, it is possible to write an efficient GPU programs in these low-level languages.

However, efficiently programming these large scale systems in low-level languages for a variety of tasks is hard for even expert programmers because of three reasons. First, there are different programming models for programming different hardware. For example, NVIDIA CUDA can be used to program only NVIDIA GPUs, and AMD HIP for AMD GPUs. Futhermore, to utilize a distributed cluster, programmers have to reconstruct the application around the Message Passing Interface(MPI) [40]. Rewriting a program for different hardware is both unproductive and time consuming. Second, these languages exposes all low level features of the underlying hardware, hence, to maximize performance, the computation must be programmed using these low level features. For example, unlike on CPUs, the cache on GPUs is not hardware managed. Hence, to utilize the cache on GPUs, programmer needs to restructure the computation that enables explicit reads and writes in cache. Third, programmers need to deal with several parallel and distributed programming issues, like synchronization, data locality, and load balancing. Since the cache on GPUs must be programmed explicitly, the programmer is responsible for adding explicit synchronization between threads of a GPU to avoid data races. Furthermore, since these programming

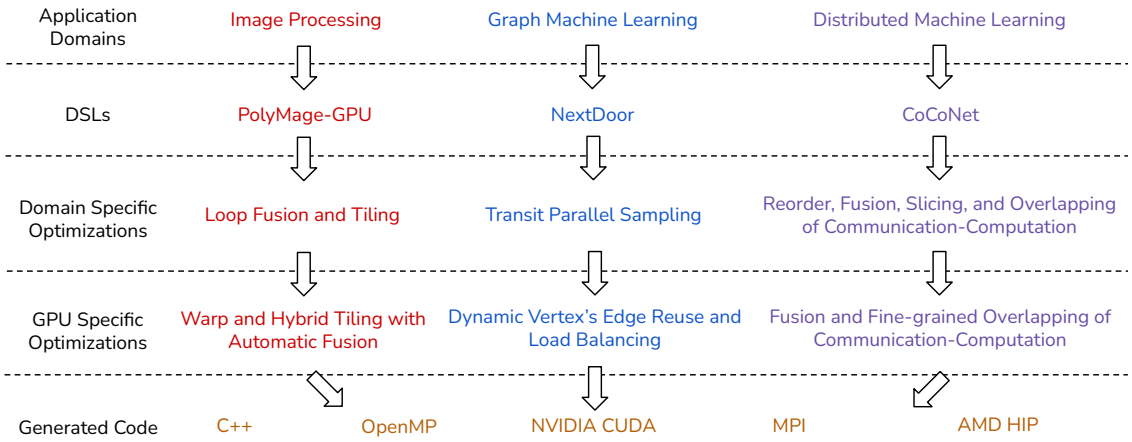


Figure 1.1: Overview of DSLs presented in this thesis. Each DSL performs both domain specific and hardware specific optimizations. Using DSLs alleviate the programmer burden and makes GPUs programming accessible to domain experts.

models and optimizations are known only to hardware experts, it is difficult for a domain expert to write new computations for which existing optimized implementations are not available.

This thesis presents novel compiler based approaches to write efficient GPU applications in the domains of image processing, graph machine learning, and distributed machine learning. This thesis presents the design of three new domain specific languages (DSLs), optimizations, and compilation techniques. These DSLs enables domain experts to write efficient and correct code for high performance systems from a single GPU to a cluster of GPUs. Figure 1.1 shows compilation pipelines for three DSLs for writing image processing, graph machine learning, and distributed machine learning applications. Each DSL enables domain experts to express applications in a natural manner and also performs domain specific static type checking to catch errors at the compile time. Hence, a DSL enables writing correct applications quickly. Then, each DSL compiler performs several domain specific optimizations. These optimizations restructure the application to generate equivalent and efficient algorithms. Furthermore, each DSL compiler performs several GPU specific optimizations that are applicable only for that domain. New optimizations can be added to the compiler to use new features in the latest hardware. Finally, each DSL compiler generates a binary executable for each application. Now, I will briefly explain all three DSLs and their compilation techniques.

PolyMage-GPU is a domain specific language for writing image processing applications and contains new tiling techniques for executing image processing applications efficiently on modern GPUs [54]. These techniques utilizes features of modern GPUs, such as warp synchronization and warp shuffles. These techniques obtained significant performance improvement over existing techniques [68, 90, 94] to execute image processing programs on GPUs.

NextDoor is a system for describing graph sampling applications and executing these applications efficiently on GPUs [55]. NextDoor provides a simple API to describe graph sampling applications in a few lines of code. NextDoor uses various load balancing and caching strategies to find and exploit reuse in these applications, which helps in achieving regular control flow and regular memory access pattern.

CoCoNet is a domain specific language for expressing and optimizing distributed computations on a cluster of GPUs [56]. CoCoNet separates a program into (i) an algorithm that specifies the computation and communication and (ii) a schedule that specifies the transformations to be done on the algorithm. CoCoNet provides several constructs for optimizing collective communication operations between GPUs of a cluster. CoCoNet enables optimizing the implementations of model parallel,data parallel, and pipeline parallel neural network training and inference in few lines of code and provides significant improvements over the state-of-the-art techniques.

The rest of this thesis is structured as follows. Chapter 2 provides background on NVIDIA GPUs, CUDA, and distributed machine learning. Chapter 3 describes warp overlapped tiling, hybrid tiling techniques, and a cost model of GPUs to choose loop fusion choices. Chapter 4 describes NextDoor, which is a system for expressing graph sampling applications and executing them efficiently on GPUs. Chapter 5 describes the CoCoNet language for describing and optimizing distributed programs that involves collective communication operations for Machine Learning workloads. Chapter 6 describes domain specialized architectures and discusses the process of adapting languages to domain specialized architectures. Chapter 7 describes the related work and shows differences between our work and existing work.

CHAPTER 2

BACKGROUND

This chapter first presents the essentials of NVIDIA Graphics Processing Units (GPU) hardware architecture. Then the chapter briefly describes distributed machine learning training and inference approaches.

2.1 An Introduction to NVIDIA GPUs and CUDA

The fundamental unit of computation in an NVIDIA GPU is a *thread*. Threads are statically grouped into *thread blocks* and assigned a unique ID within a block. An NVIDIA GPU contains multiple *streaming multiprocessors (SMs)*, each of which executes one or more thread blocks. Rather than scheduling each individual thread, an SM schedules a subset of threads from the thread block known as a *warp*. On NVIDIA GPUs, a warp is set of 32 threads with consecutive thread IDs.

NVIDIA GPUs employ a *Single Instruction Multiple Threads (SIMT)* execution model, i.e., all threads in a warp run the same instruction in lock-step. One consequence of this execution model is that two threads of a warp cannot execute both sides of a branch concurrently. Therefore, when the threads in a warp encounter a branch, the subset of threads that do not take the branch must wait for other threads to complete the branch. This phenomenon is known as *warp divergence* and can lead to poor performance.

NVIDIA GPUs have a deep memory hierarchy. Two major types of memory are: (i) *global memory*, which is accessible to all threads running on all SMs and (ii) *shared memory*, which is a memory private to a SM and is only available to thread blocks assigned to that SM. Furthermore, modern NVIDIA GPUs allow threads of a warp to read values stored in another thread's registers of the same warp using *warp shuffle* instructions. The amount of global memory on each GPU is in the order of Gigabytes but has higher latency and lower bandwidth than shared memory and registers. Each thread block can access up to 48KB of shared memory and each thread can have

```
1 int val = rand ();
2 for (int offset = 16; offset > 0; offset /= 2)
3   val += __shfl_sync(0xffffffff, val, threadIdx.x+offset, warpSize);
```

Figure 2.1: CUDA kernel summing variable `val` of all threads of a warp using `__shfl_sync`.

up to 256 registers. Programmers need to explicitly use shared memory and registers to cache intermediate data.

Since caching the intermediate data can lead to data races without synchronization, NVIDIA GPUs provide two kinds of synchronization: (i) *Thread block synchronization* synchronizes all threads in a block: until all warps in the block reach the same `__syncthreads` statement, no warp is allowed to proceed, and (ii) *warp synchronization* synchronizes all threads in a warp, and no thread can proceed until all threads in the warp reach the synchronization point (`__syncwarp`). A consequence of thread block synchronization is that it decrease SM utilization when all warps of other thread blocks on the SM are stalled on a global memory access while warps of current thread block are waiting for synchronization. However, in warp synchronization other warps can make progress because the warp synchronization only synchronizes all threads of a warp. Hence, thread block synchronization is costlier than warp synchronization.

Since all resources of an SM like shared memory and registers are of fixed size, there is a maximum limit on the number of warps it can concurrently execute. This limit is represented by a value known as *occupancy*, which is the ratio of number of concurrent warps executed by each SM for a given GPU program to the maximum number of warps an SM can execute concurrently. The value of occupancy of a GPU program depends on the number of thread blocks, the number of threads per thread block, the shared memory used by each thread block, and the registers used by each thread.

The *warp shuffle* instructions [2, Chapter B.16] available in recent NVIDIA GPUs allow threads to read register values from other threads in the same warp. The `__shfl_sync` instruction takes four arguments: a 32-bit mask of threads participating in the shuffle, the variable stored in the register to read, the index of the source thread containing the register, and the warp size. Similarly, `__shfl_down_sync` and `__shfl_up_sync` read registers from a thread with an index immediately before or after the calling thread. Figure 2.1 shows an example from [3]

of reduction using `__shfl_sync`. In this example, CUDA kernel invoked with 32 threads in x-dimension. At each iteration, each thread add next `offset` thread's `val` to its `val`. At the end of loop, `val` of the first thread contains the sum. For a shuffle to succeed both the calling thread and source thread must execute the instruction.

2.1.1 Characteristics of an Efficient GPU Program

A GPU program should have following characteristics to ensure complete use of all GPU resources.

Minimimum Warp Divergence Warp divergence happens when the threads in a warp encounter a branch and a subset of threads that do not take a branch must wait for other threads to complete their branch. Hence, warp divergence can decrease GPU utilization, thereby, decreasing the performance. Thus, minimum warp divergence is a characteristic of a high-performant GPU program.

Balanced Load An efficient GPU program should balance its load across all threads in a thread block to ensure that all execution resources of a GPU are utilized. Futhermore, the load must be balanced among all thread blocks because a thread block can have a maximum number of threads, and if some thread blocks have higher load than others, then other thread blocks will remain idle.

Coalesced Memory Accesses When threads in a warp accesses global memory, then these accesses are combined into one or more transactions of a fixed size (32 Bytes for NVIDIA GPUs). Minimizing the number of transactions is a key characteristic of an efficient GPU program. A GPU program can achieve minimum transactions only if threads performs coalesced memory accesses, i.e., consecutive threads accesses consecutive memory locations.

Maximum Cache Usage GPUs provide two fast memories that can be utilized as cache: shared memory and registers. Since both caches have significantly lower latency and higher bandwidth than global memory, maximizing the cache usage can significantly improve the performance.

Maximum Warps per SM Since each SM on a GPU have a fixed amount of shared memory and registers, the number of threads that can concurrently run on an SM (known as *occupancy*) is

dependent on the amount of shared memory allocated to each threadblock and registers assigned to each thread. However, we cannot always decrease the usage of shared memory and registers significantly to increase occupancy because it will decrease the amount of cached data and lead to poor performance. Hence, an efficient GPU program runs at a sweet spot between occupancy and usage of shared memory and registers that leads to high-performance.

Utilize all Resources Simultaneously A GPU's resources can be divided into three groups computation, memory, and network resources. Even though a GPU's hardware thread scheduler tries to utilize all resources completely by overlapping computation with memory and network accesses, if a GPU program does not utilize any of these resource then these resources can be under utilized. Hence, a GPU program must overlap different GPU programs in a fine-grained manner where each kernel might utilize only some of the resources.

2.2 Distributed Neural Network Training and Inference

Machine learning at scale requires distributed hardware. For example, training large models such as BERT [38] with 340 million parameters, GPT-2 [88] with 1.5 billion parameters, and GPT-3 [26] with 175 billion parameters, requires thousands of machines. Training and inference for these large models on multiple GPUs exploit three forms of parallelism:

Data Parallelism In data parallelism, the dataset is divided among GPUs. Each GPU perform the forward pass and backward pass based on the input dataset. After the backward pass, each GPU now hold different values of gradient for same parameter because each GPU has different dataset. Hence, all GPUs obtains the average value of gradient of each parameter. After computing the average, each parameter is updated using optimizers, such as Adam [62] and LAMB [115]. This process ensures that value of each parameter on all GPUs are same.

Model Parallelism In model parallelism [101], the model is divided among the GPUs. Each GPU perform a part of the layer computation and then all GPUs sums the part to obtain all values of layer on each GPU.

Pipeline Parallelism In pipeline parallelism [51] each layer (or a batch of contiguous layers) is assigned to a group of GPUs. When computation of one layer (or the batch of contiguous layers)

is completed, the output of the computation is send to the next group of GPUs that contains next layers (or next batch of contiguous layers).

2.2.1 Communication Collectives

Distributed training and inference utilizes several communication primitives defined by the the MPI Standard [40] and implemented in libraries, such as, NVIDIA Collective Communication Library (NCCL) [15] and OpenMP. A key class of communication primivites used in distributed training and inference are collective communication primitives, where all nodes collectively performs several sends and receives to obtain the output. This section gives an overview of important communication collectives.

This thesis follow the MPI terminology to represent the process ID of a distributed process as *rank* and the set of all processes as WORLD, such that, ranks ranges from 0 to $|\texttt{WORLD}| - 1$. All communication collectives takes an input buffer b_i of size N_i and writes to an output buffer b_o of size N_o. Below are some common communication collective primitives supported by NCCL and OpenMPI:

- *AllReduce* performs a reduction operation on b_i and leaves identical copies of b_o on all ranks.
- *AllGather* gathers all N_i values of b_i from all ranks to b_o, such that, $N_o = N_i \times |\texttt{WORLD}|$.
- *ReduceScatter* performs a reduction operation on b_i and scatter the result among all ranks in b_o, such that, $N_o = N_i \div |\texttt{WORLD}|$.
- *Reduce* takes a root rank r, performs reduction on b_i and only writes the result to b_o of r.
- *Broadcast* takes a root rank r. It copies N_i values using *Send*of b_i of rank r and leaves identical copies in b_o of all ranks.
- *Send* takes a destination rank r. It copies N_i values of b_i from current rank to rank r.

CHAPTER 3

EFFICIENT IMAGE PROCESSING ON MODERN GPUS

Image processing programs are essential in several domains, including computer vision, embedded vision, computational photography, and medical imaging. These programs run on a variety of platforms, from embedded systems to high-performance clusters that process large amounts of image data.

An image processing program is logically structured as a directed acyclic graph of connected stages, where each stage performs per-pixel data parallel operations on its input image and produces an output image for dependent stages. There are several domain-specific languages (DSLs) for writing image processing programs, including Halide [90], PolyMage [75], and Forma [92]. These DSLs allow the programmer to write independent stages in a natural way, but still get high-performance code by applying key optimizations, including *loop fusion* and *overlapped tiling*. After loop fusion, overlapped tiling [75, 90, 92] splits each stage into overlapping regions (known as tiles) that can be processed in parallel without synchronization with other tiles. Existing approaches to execute image processing programs on a GPU [22, 49, 86, 90, 92, 93, 94, 107, 117] maps each tile to a *thread block* and stores intermediate results (scratchpad arrays) in the shared memory associated with a thread block. However, these techniques give suboptimal performance on modern GPUs for three reasons. 1) Processing an overlapped tile per thread block has a high synchronization cost across stages. 2) Smaller tiles have more overlapped regions (and thus require more redundant computation), but larger tiles require more shared memory accesses (and thus lower occupancy). 3) State-of-the-art autoscheduling algorithms for loop fusion and tile-size selection do not employ a rich cost model for GPUs. For example, cost models in [68, 94] do not consider the number of global memory transactions, the ability to hide latency of global memory accesses, and occupancy.

This chapter introduces, PolyMage-GPU (based on PolyMage [75]), a compiler for image processing programs that leverages the architecture of modern GPUs to generate high performance code. PolyMage-GPU exploits the fact that all threads in a *warp* can synchronize using warp syn-

chronization, which has significantly lower overhead than thread block synchronization. In addition, modern GPUs have *warp shuffle* [2, Chapter B.16] instructions that allow threads in a warp to read each others' register values. PolyMage-GPU uses warp shuffles to lower shared memory usage and support larger overlapped tiles. PolyMage-GPU contains a cost model for GPUs that accounts for several factors, including the number of global memory transactions, occupancy, and resource utilization. This cost model is used to determine the optimal tile, thread block sizes and loops to fuse, using Dynamic Programming Fusion [53].

The rest of this chapter is organized as follows. Section 3.2 presents an overview of the approach. Section 3.3 presents the technique for running one overlapped tile per warp. Section 3.4 presents hybrid tiling. Section 3.5 presents the automatic fusion algorithm. Section 3.6 evaluates this work over state-of-the-arts. Finally, Section 3.7 concludes this chapter.

3.1 PolyMage DSL

PolyMage [75] is a DSL embedded in Python for writing image processing pipelines. The PolyMage compiler transforms programs in the DSL into high-performance code for CPUs. Figure 3.1 shows an image blurring program (*blur*) with two stages (`blurx` and `blury`). The parameters to the pipeline are the number of rows and columns in the image (line 1). The program first feeds the input image (`img` on line 9) to `blurx`, and then the output of `blurx` to `blury`. Each stage is a function mapping a multi-dimensional integer domain to values representing intensities of image pixels (lines 20 and 22). The domain of the function is defined at lines 12–14. `blurx` takes the image as input and blurs it in the x-direction (lines 20–21). `blury` blurs the output of `blurx` in the y-direction and produces final output (lines 22–23). The PolyMage compiler performs loop fusion on producer-consumer stages to improve locality and provide parallel execution. When fusing two stages, PolyMage performs overlapped tiling using polyhedral transformations. Two adjacent tiles perform redundant computations to ensure that all the data required to compute the output of a tile (known as *liveouts*) is available within that tile, providing parallel execution of all tiles. Within a tile, the output of a producer stage is transferred to its consumer using small buffers, known as *scratchpads*. A scratchpad is small enough to fit in a CPU cache, or in the work presented in this thesis, in GPU shared memory or registers.

```
R,C = Parameter(Int,"R"),Parameter(Int,"C")

# Vars
x = Variable(Int,"x")
y = Variable(Int,"y")
c = Variable(Int,"c")

# Input Image
img = Image(Float,"img",[3,R+2,C+2])

# Intervals
cr         = Interval(Int,0,2)
xrow,xcol  = Interval(Int,1,R),Interval(Int,0,C+1)
yrow,ycol  = Interval(Int,1,R),Interval(Int,1,C)

# Functions
cond       = Condition(x,'>=',1) & Condition(x,'<=',R) &
             Condition(y,'<=',C) & Condition(y,'>=',1)

blurx      = Function(([c,x,y],[cr,xrow,xcol]), Float,"blurx")
blurx.defn = [Case(cond,(img(c,x-1,y) + img(c,x,y) + img(c,x+1,y))/3)]
blury      = Function(([c,x,y],[cr,yrow,ycol]), Float,"blury")
blury.defn = [Case(cond,(blurx(c,x,y-1) + blurx(c,x,y) + blurx(c,x,y+1))/3)]
```

Figure 3.1: PolyMage DSL specification for *blur*.

3.1.1 Dependence Vectors

PolyMage uses dependence vectors to encode the dependencies between consumer and producer stages. A *dependence vector* [112] is the difference of the time stamps when a value is consumed and when it is produced. For example, in the *blur* program, the `blury` stage, at `(2,c,x,y)`, consumes values that the `blurx` stage produces at `(1,c,x,y-1)`, `(1,c,x,y)`, and `(1,c,x,y+1)`. This is captured by the dependence vectors (1,0,0,-1), (1,0,0,0), and (1,0,0,1).

3.1.2 Dynamic Programming Fusion

Dynamic Programming Fusion (DP-Fusion) [53] is an algorithm that performs automatic fusion of image processing pipelines in a few seconds. DP-Fusion finds schedules that are competitive with schedules generated by autotuner over a few days, and are better than a greedy CPU autoscheduler [76]. Instead of using a greedy algorithm and a simple cost function, DP-Fusion enumerates all valid fusion possibilities and uses dynamic programming combined with an analytic cost function to significantly decrease the runtime of a combinatorial algorithm. Among all fusion possibilities, DP-Fusion finds the best fusion choices on the basis of the cost of candidate

```
blur_otptb(img[3][R][C], blury[3][R-2][C-2])
  shared blurx[blockDim.y][tile*blockDim.x+2];
  c = threadIdx.z;
  y = blockIdx.y*blockDim.y + threadIdx.y;
  for (tx = 0; tx < tile+1; tx++)
    xx = tx * blockDim.x + threadIdx.x;
    x  = (blockIdx.x*blockDim.x)*tx + xx;
    if (xx < tile*blockDim.x+2)
      blurx[y][xx] = (img[c][y-1][x]+img[c][y][x]+img[c][y+1][x])/3;
  __syncthreads ();
  for (tx = 0; tx < tile; tx++)
    xx = tx * blockDim.x + threadIdx.x;
    x  = (blockIdx.x*blockDim.x)*tx + xx;
    blury[c][y][x] = (blurx[y][xx-1]+blurx[y][xx]+blurx[y][xx+1])/3;
```

Figure 3.2: Equivalent CUDA code generated by Halide for *blur*, where both *blurx* and *blury* are fused in an overlapped tile of size `tile` in x and 1 in y.

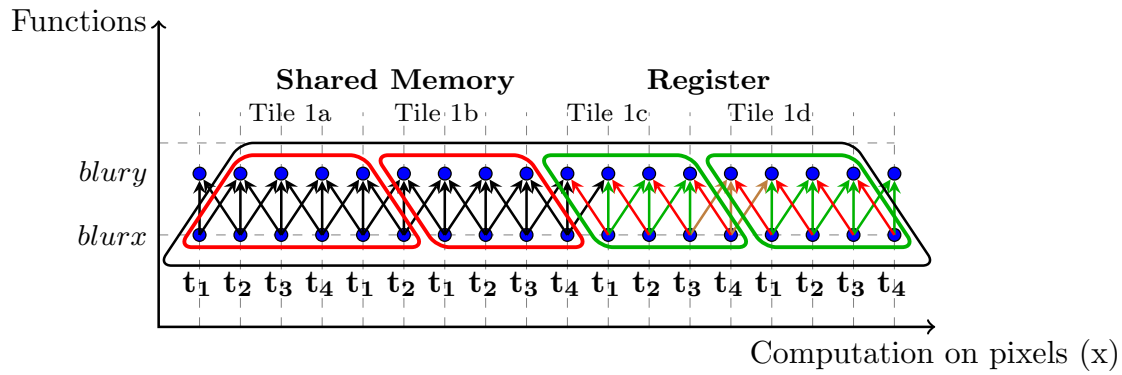


Figure 3.3: Hybrid Tiling for *blur* program with tile of 2 in x and warp size of 4. The overlapped tile is split into four tiles. Red tiles are stored in shared memory and green tiles in registers.

fused loops. The cost of fused loops is calculated using a cost function that also uses a model to determine tile sizes. PolyMage uses DP-Fusion to find the best schedules for image processing programs executing on multi-core CPUs [53].

3.2 PolyMage-GPU Overview

This section provides an overview of PolyMage-GPU's warp tiling and hybrid tiling by generating optimized code for the *blur* program in Figure 3.1.

Figure 3.2 shows CUDA code that is equivalent to the code that Halide produces for *blur* pipeline in Figure 3.1. The code fuses both *blurx* and *blury* together and uses overlapped tiles

```
1  blur_otpw_ht(img[3][R][C],
2               blury[3][R-2][C-2])
3    shared blurx[blockDim.y]
4                [blockDim.x/warpSz]
5                [tile/2*warpSz+2];
6    y = blockIdx.y * blockDim.y+threadIdx.y;
7    c = threadIdx.z;
8    warpSz = warpSize;
9    warp = threadIdx.x/warpSz;
10
11   for(tx = 0; tx < 2; tx++)
12    for(txx = tx*tile/4;
13        txx<(tx+1)*tile/4+1;txx++)
14     xx = tile*warpSz+threadIdx.x%warpSz;
15     x = (blockIdx.x+1)*blockDim.x*tx+
16          threadIdx.x;
17     if(xx < tile*warpSz+2)
18      blurx[y][warp][xx]=(img[c][y-1][x]+
19      img[c][y][x]+img[c][y+1][x])/3;
20   x = warp+8*warpSz+lane_x;
21   blurx_8 = (img[c][y-1][x]+img[c][y][x]+
22              img[c][y+1][x])/3;
23   x = warp+9*warpSz+lane_x;
24   blurx_9 = (img[c][y-1][x]+img[c][y][x]+
25              img[c][y+1][x])/3;
26   /*for all iterations till 15*/
27   syncwarp();
28   for(tx = 0; tx < 2; tx++)
29    for(txx = tx*tile/4;
30        txx <(tx+1)*tile/4; txx++)
31     xx = tile*warpSz+threadIdx.x%warpSz;
32     x = (blockIdx.x+1)*blockDim.x*tx+
33         threadIdx.x;
34     if(xx > 0 and xx < tile/2*warpSz+2)
35       blury[c][y][x] =
36         (blurx[y][warp][xx-1]+
37         blurx[y][warp][xx]+
38         blurx[y][warp][xx+1])/3;
39   blurx_l_2_8 = shfl_up(blurx_8, 2);
```

```
40   /*for all iterations till 15*/
41   syncwarp();
42   for(tx = 0; tx < 2; tx++)
43    for(txx = tx*tile/4;
44        txx <(tx+1)*tile/4; txx++)
45     xx = tile*warpSz+threadIdx.x%warpSz;
46     x = (blockIdx.x+1)*blockDim.x*tx+
47         threadIdx.x;
48     if(xx > 0 and xx < tile/2*warpSz+2)
49       blury[c][y][x] =
50         (blurx[y][warp][xx-1]+
51         blurx[y][warp][xx]+
52         blurx[y][warp][xx+1])/3;
53   blurx_l_2_8 = shfl_up(blurx_8, 2);
54   blurx_l_1_8 = shfl_up(blurx_8, 1);
55   int rt = 7*warpSz+warpSz;
56   if(lane_x == 0)
57    blurx_l_2_8=blurx[y][warp][rt-2];
58    blurx_l_1_8=blurx[y][warp][rt-1];
59   if(lane_x == 1)
60    blurx_l_2_8=blurx[y][warp][rt-1];
61   x = warp+8*warpSz+lane_x;
62   blury[c][y][x] = (blurx_l_2_8+
63                     blurx_l_1_8+
64                     blurx_8)/3;
65   blurx_l_2_9 = shfl_up(blurx_9, 2);
66   blurx_l_1_9 = shfl_up(blurx_9, 1);
67   _blurx_l_2_9=shfl(blurx_8,warpSz-2);
68   _blurx_l_1_9=shfl(blurx_8,warpSz-1);
69   if(lane_x == 0)
70     blurx_l_1_9 = _blurx_l_1_9;
71     blurx_l_2_9 = _blurx_l_2_9;
72   if(lane_x == 1)
73     blurx_l_2_9 = _blurx_l_1_9;
74   x = warp+9*warpSz+lane_x;
75   blury[c][y][x] = (blurx_l_2_9+
76                     blurx_l_1_9+
77                     blurx_9)/3;
78   /*for all iterations till 15*/
```

Figure 3.4: Hybrid Tiling CUDA code for *blur*, with *blurx* and *blury* fused in an overlapped tile of size `tile` in the x-dimension, which is computed by one warp.

of length `tile` in the x-dimension and unit length in y-dimension. During the execution, all threads in a thread block 1) compute *blurx* in parallel by looping over all points in the tile (lines 5–9), 2) store the result of *blurx* in a scratchpad (which is in shared memory), 3) use thread-block synchronization to ensure that all *blurx* values are ready (line 10), and 4) calculates *blury* in parallel, which depends on *blurx* (line 11– 14). On an NVIDIA GTX 1080Ti, this code exhibits its best performance (1.40ms) on a $4096\times4096\times3$ input with 8 tiles and block sizes of $64\times4\times1$. However, thread block synchronization requires to warp switching that can lead to decrease in performance, so there is room for improvement.

Overlap Tile per Warp (OTPW) The program can be modified to assign each overlapped tile to a warp, instead of a thread block, so that, the program uses warp synchronization (`__syncwarp`), which allows the SM to execute a warp even if another warp is waiting for a memory access. This code exhibits its best performance (1.35ms) with 8 tiles and block sizes of $64\times4\times1$. This is a $1.04\times$ speedup over the prior approach. This choice of tile size produces 0.8% redundant computations per warp. Although, using tile size 16 leads to fewer redundant computations (0.4%), it increases the running time (1.45ms) because of far more shared memory usage (over 16KB). This limits the number of warps that the GPU can run concurrently, i.e., occupancy is only 62.5%.

Hybrid Tiling To further improve performance, we propose *hybrid tiling*, which is a technique that decreases the size of the scratchpad buffer in shared memory, by storing some parts of the overlapped tile in registers. Since each tile is assigned to a warp, *warp shuffle* instructions can be used to enable threads in a warp to read register values from other threads in the same warp. This eliminates the need for per thread redundant computation that arise in register blocking. Figure 3.3 sketches the structure of the computation, assuming four tiles: the first two tiles are stored in shared memory, whereas the latter two tiles are stored in registers. When a *blury* value depends on a *blurx*-value in a register, it can read it directly, using warp shuffles to read across threads if needed.

On a GTX 1080Ti, the code so far only uses 24 registers. With a tile size of 16, half of the tile is stored in registers, which halves the shared memory usage, and leads to 100% occupancy. With hybrid tiling, the code runs in 1.2ms which is $1.13\times$ faster than the *OTPW* approach, and $1.16\times$ faster than the original program.

Figure 3.4 sketches the CUDA code for *blur* that uses *overlap tile per warp* and *hybrid tiling*. In this code `shfl*` refers to `__shfl*_sync`. In the figure, the data points of *blurx* for first two tiles are stored in shared memory while the later tiles are stored in the registers. Lines 11–19 processes *blurx* on the first two tiles stored in shared memory using a warp by assigning consecutive data points to consecutive threads in a warp and looping over all points in both tiles. Lines 20–40 unroll the loop and store each data point in registers for two register tiles. Lines 42–52 compute the values of *blury* for first two tiles that are stored in shared memory. Lines 53–54 retrieve the values of *blurx* from other threads using *warp shuffle*. Since the first two values for the first thread

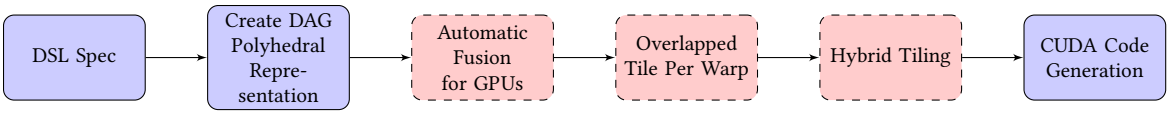


Figure 3.5: Compilation pipeline of image processing program written in PolyMage-GPU, which is based on PolyMage [75].

in a warp are the values produced and stored in shared memory by last two threads of that warp, lines 56–60 retrieve the last two values of shared memory for that warp. Line 64 computes each *blury* point for the eighth iteration of the larger overlapped tile. Similarly, for the ninth iteration, lines 65–73 retrieve the values of `blurx_9` from previous threads and for first two threads of warp values of `blurx_8` are retrieved from last two threads of the warp.

Loop Fusion The final problem involves choosing best performing tile and block sizes. I present an automatic fusion algorithm that considers key factors affecting the performance of GPU kernels which are not considered in previous works [18, 68, 76]: 1) number of global memory transactions, 2) achieved and theoretical occupancy, 3) GPU resource usage, and 4) fraction of overlapping computations.

All techniques described in the paper, i.e., OTPW, hybrid tiling, and the fusion algorithm are implemented in PolyMage-GPU. Figure 3.5 shows the structure of the compilation pipeline.

3.3 Overlap Tile per Warp

This section describes the how to generate a warp overlapped tile for any loop.

Let (b_x, b_y, b_z) be the coordinates of a thread block and (B_x, B_y, B_z) be the thread block size. Consider a group of fused stages with tile sizes (T_x, T_y, T_z) that consumes a three-dimensional input of size (N_x, N_y, N_z), where each dimension is labelled $i \in \{x, y, z\}$. The linear thread ID for a three-dimensional coordinates of a thread (t_x, t_y, t_z) is: $t_x + B_x \times t_y + B_x \times B_y \times t_z$. The Warp ID of a thread is the thread ID divided by `WarpSize` and the index of a thread in a warp (known as its *lane ID*) is the remainder. Warp sizes, W_x, W_y, W_z, are defined as:

$$
\begin{aligned}
W_x &= \texttt{minimum}(B_x, \texttt{WarpSize}) \\
W_y &= \texttt{minimum}(B_y, \texttt{WarpSize} \div W_x) \\
W_z &= \texttt{minimum}(B_z, \texttt{WarpSize} \div (W_x \times W_y))
\end{aligned}
$$

These warp sizes are the number of threads with distinct IDs of that dimension in a warp. The number of warps in dimension i in a thread block is equal to the ratio of block size to the warp size of that dimension ($\lceil B_i/W_i \rceil$). The warp ID of a thread in a dimension is the floor of division of the thread's ID in that dimension to the warp size of that dimension, i.e.($\lfloor (b_i \times B_i + t_i)/W_i \rfloor$). Moreover, the lane ID is the remainder ($(b_i \times B_i + t_i) \bmod W_i$). Note that the product of all the warp sizes obtained using these equations is equal to `WarpSize`. For given overlapped tile sizes, we create a *warp overlapped tile* by extending the tile sizes of each dimension to cover exactly one warp. The total number of points in a warp overlapped tile excluding the redundant computations is the product of the number of points in the given overlapped tile sizes and `WarpSize`. For the given overlapped tile size, the size of the warp overlapped tile is $(T_x \times W_x, T_y \times W_y, T_z \times W_z)$.

The size of each scratchpad for a stage is exactly the number of data points computed by the thread block for that stage. For the n^{th} stage, each warp computes two types of data points in the i^{th} dimension: 1) $T_i \times W_i$ computations for the tile, and 2) O_i^n overlapping computations. The number of data points computed (and the size of the scratchpad) for n^{th} stage is represented as $\prod_{i \in \{x,y,z\}} \lceil B_i/W_i \rceil \times (T_i \times W_i + O_i^n)$.

3.4 Hybrid Tiling

This section presents *hybrid tiling*, which divides a tile between shared memory and registers. Hybrid tiling relies on the fact that each overlapped tile fits in a single wrap. Hybrid tiling use *warp shuffle* instructions to allow each thread to access data from other threads in a warp, which eliminates the need for certain redundant computations per thread. Hybrid tiling solves the issues of shared memory only tiling by 1) storing a part of a tile in registers to decrease allocated shared memory, 2) providing extra storage for larger tile sizes, which results in fewer redundant computations, and in turn, fewer global memory loads and total computations; and 3) storing tiles partially in registers, which leads to faster access to data points.

The warp overlapped tile is split over a *split dimension*, into several parallelogram tiles with left tiles stored in shared memory and right tiles stored in registers (Figure 3.3). These smaller parallelogram tiles are of warp size in the split dimension, and the same size as the warp overlapped tile in other dimensions. The slope of the parallelogram tiles are parallel to the right hyperplane of the warp overlapped tile in the split dimension, which ensures there is no cyclic dependence between two adjacent tiles.

Since all producer loads by *OTPW* are in the shared memory, these loads have to be converted to access data stored in registers if necessary. Figure 3.3 shows that there are four types of producer loads: (i) the producer loads in red are from the registers of current thread, if the load index is same as the iteration in the split dimension (Type ①); (ii) the producer loads in black are from shared memory (Type ②), if the load index is less than the lower bound of the register tile in the split dimension; (iii) the producer loads in green are from the registers of another thread in same parallelogram tile (Type ③), if the load index in the split dimension is less than the iteration in the split dimension; and (iv) the producer loads in brown are from registers of another thread in previous parallelogram tile (Type ④), if the difference between the lane ID of the current thread in the split dimension and the difference between the iteration and load index in the split dimension is less than zero.

Algorithm 1 is the hybrid tiling algorithm that uses dependence vectors between producer and consumer stages. The arguments to the 2-D-HybridTiling function are the group of stages (G), tile sizes ($T_x \times T_y$), warp sizes ($W_x \times W_y$), and register tile size ($fracReg$) as a fraction of the tile size in the split dimension. The result of the function is the CUDA code that does hybrid tiling. First, the algorithm finds a split dimension with tile size greater than 1 (line 16). If no such dimension is found, then tiles must be stored entirely in the shared memory. The rest of the algorithm assumes that the x-dimension is the split dimension. Let ϕ_{rx} and ϕ_{ry} be the right hyperplanes of warp overlapped tiles of G in the x and y dimensions respectively. The algorithm first generates the shared memory tile using the PolyMage compiler, and then generate register tiles using the GenRegTile function that takes a stage of the group (H), the hyperplanes (ϕ_{rx}, ϕ_{ry}), the register tile size ($R_x \times R_y$), and the warp sizes ($W_x \times W_y$) as arguments (lines 24–25).

For all the iterations in the register tile, including the overlapping computations, the algorithm stores each computed value of stage H in a distinct variable, instead of shared memory (line 5). The

```
1 val = Reg_P[x][c*y + d]
```

(a) Code for a register access from same thread is generated when the source lane ID is the current lane ID, i.e. $\phi_x = \phi_{rx}$ (Type ①).

```
1 currTileSrcLane = (laneId.x + phi.x) + (laneId.y + diffPhi.y)*warpSize.x;
2 /*Type 3:*/ val = __shfl_sync(getMask(), Reg_P[x][c*y+d], currTileSrcLane);
3 /*Type 2:*/ if (laneId.x + diffPhi.x < 0) val = ShMem_P[a*x+b][c*y+d];
```

(b) Code generated when current iteration is the first iteration of register tile and $\phi_x - \phi_{rx} \neq 0$. When the sum of lane index and $\phi_x - \phi_{rx}$ is less than zero, then value is accessed from shared memory tile (Type ②), otherwise value is accessed from register of thread in the same parallelogram tile (Type ③). `diffPhi.x` is the value of $\phi_x - \phi_{rx}$. `diffPhi.y` is the value of $\phi_y - \phi_{ry}$

```
1 prevTileSrcLane = (warpSize.x - 1 + diffPhi.x) +
2 (warpSize.y - 1 + diffPhi.y)*warpSize.x;
3 currTileSrcLane = (laneId.x + diffPhi.x) + (laneId.y + diffPhi.y)*warpSize.x;
4 /*Type 3:*/ val = __shfl_sync(getMask(), Reg_P[x][c*y+d], currTileSrcLane);
5 /*Type 4:*/ if (laneId.x + diffPhi.x < 0)
6   val = __shfl_sync(getMask(), Reg_P[x-1][c*y+d], prevTileSrcLane);
```

(c) Code generated when current iteration is not the first iteration of register tile and $\phi_x - \phi_{rx} \neq 0$. When the sum of lane index and $\phi_x - \phi_{rx}$ is less than zero, then value is accessed from register of last $|\phi_x - \phi_{rx}|$ threads of previous parallelogram tile (Type ④) otherwise value is accessed from register of thread in the same parallelogram tile (Type ③). `diffPhi.x` is the value of $\phi_x - \phi_{rx}$. `diffPhi.y` is the value of $\phi_y - \phi_{ry}$

Figure 3.6: Code generation cases for a producer `p[a*x+b][c*y+d]` at iteration {x, y} of register tile that generates all four load types of Figure 3.3.

algorithm replaces each producer load in the loop with either a shared memory read or a warp shuffle (lines 6–14). Then the algorithm computes the dependence vector between the producer and consumer (line 7) as ϕ_x and ϕ_y. Figure 3.6 shows the code generated for three cases that arise when generating code for a load `P[a*x+b][c*y+d]`. The figure shows two types of source lane IDs that contain the register, which stores the value of the producer load: 1) `currTileSrcLane` is the lane ID for a source thread in the current parallelogram tile and 2) `prevTileSrcLane` is the lane ID for a source thread in the previous parallelogram tile. Value of both IDs in x-dimension depends on $\phi_x - \phi_{rx}$ and in y-dimension depends on $\phi_y - \phi_{ry}$. I now explain each of the three cases in detail. 1) If $\phi_x = \phi_{rx}$, then the value needed for this load is stored by the current thread's register and the algorithm generates the code for Type ① (line 9). 2) When $\phi_x - \phi_{rx} \neq 0$ and the iteration in split dimension, i.e., x-dimension is first iteration of the register tile, then first $|\phi_x$

Algorithm 1 Hybrid Tiling

```
1: function GenRegTile(H, φ_rx, φ_ry, R_x× R_y, W_x×W_y)
2:     for all {x, y} ∈ [1, ...R_x] × [1, ...R_y] do
3:         Let iteration {x, y} be
4:         H[x][y] = f(P[a*x+b][c*y+d], ...)
5:         Store H[x][y] in a register array Reg_H[x][y]
6:         for all loads P[a*x+b][c*y+d] ∈ f do
7:             φ_x, φ_y = dependence vectors between P[a*x+b][c*y+d] and H[x][y]
8:             if φ_x == φ_rx then
9:                 Generate Type① code in Figure 3.6a
10:            else if x == 1 then
11:                Generate Type ② and ③ code from Figure 3.6b
12:            else
13:                Generate Type ③ and ④ code from Figure 3.6c
14:            Replace P[a*x+b][c*y+d] with val in generated code
15: function 2-D-HybridTiling(G, fracReg, T_x× T_y, W_x×W_y)
16:     splitDim = a dimension with tile size greater than 1
17:     If no split dimension exists then return
18:     Let φ_rx and φ_ry be right hyperplanes of G in x and y
19:     Let splitDim is the x-dimension.
20:     Create parallelogram tiles in x-dim of size W_x parallel to φ_rx
21:     R_x ← T_x× fracReg, S_x ← T_x× (1 - fracReg)
22:     R_y ← S_y ← T_y
23:     for all H ∈ G do
24:         Gen. Shared Mem Tile with tile size S_x×S_y
25:         GenRegTile(H, φ_rx, φ_ry, R_x× R_y, W_x×W_y)
```

- ϕ_{rx}| threads of warp loads from shared memory (Type ②) and remaining threads loads from registers of threads in same parallelogram tile (Type ③). Figure 3.6b shows the code generated for this case. The conditional determines whether to load from shared memory or from another thread's register. The `__shfl_sync` function loads the value from the source thread's register. The function `getMask` retrieves the mask of threads that can participate in the warp shuffle. 3) Otherwise, if a thread needs to load from another thread's register that stores value of either the current parallelogram tile (Type ③) or the previous parallelogram tile (Type ④), then the algorithm generates the code in Figure 3.6c. Two warp shuffles are generated that are executed by all threads and a conditional expression selects which loaded value to use.

Model	GTX 1080Ti	Tesla V100
Simultaneous Multiprocessors	28	80
CUDA Cores per SM	128	64
Global Memory Bandwidth	484 GBps	898 GBps
Maximum Shared Memory Per Thread Block	48 KB	96 KB
Shared Memory per SM	96 KB	
Maximum Warps per SM	64	
Maximum Thread Blocks per SM	16	32
Registers per SM	65536	
Maximum Registers Per Thread	256	
Warp Size	32	
Global Memory Transaction Size	32 B for L2 Cache 128 B for L1 Cache	

Table 3.1: Specifications of the GPUs used in experiments.

3.5 Automatic Fusion for GPUs

This section presents an automatic fusion algorithm that selects 1) sets of stages to fuse, 2) their tile sizes, and 3) their thread block sizes. This approach leverages DP-Fusion [53], which is an algorithm that efficiently enumerates all fusion possibilities, given a cost function. I introduce a cost function that calculates the minimum cost of a sequence of fused loops, along with optimal tile sizes and thread block sizes.

The algorithm takes two types of information about each stage as input. First, the register usage of the stage, which is determined using `nvcc`. Second, the running time per iteration of the stage, which is obtained by generating code for each individual stage, where global memory loads are replaced by shared memory loads.

The function Cost takes four arguments: 1) a group of stages to fuse, G, 2) tile sizes, 3) thread block sizes, 4) fraction of tile stored in registers, and returns the cost. The function refers to the the hardware configuration of a GPU (Table 3.1). The expression below calls the Cost function for all tile sizes, thread block sizes, and fraction of tile stored in registers including 0.0 (hybrid tiling disabled) and 1.0 (except the overlap in split dimension the complete tile is stored in registers), and global memory transaction size for both L1 and L2 global memory cache, and returns the minimum cost with the appropriate global memory cache enabled, tile sizes, thread block sizes, and the fraction of tile stored in registers:

$$\underset{\substack{\text{tileSize}\in\text{Tile Sizes},\\ \text{tbSize}\in\text{Thread Block Sizes},\\ \text{fracRegTile}\in\{0.0,0.1,\ldots,1.0\},\\ \texttt{GLMemTxSize}\in\{32,128\}}}{\text{argmin}} \quad \textsc{Cost}\,(G, \text{tileSize}, \text{tbSize}, \text{fracRegTile})$$

The Cost function determines the cost based on 1) the number of global memory transactions per warp, 2) theoretical maximum occupancy, 3) achieved occupancy, 4) shared memory usage, 5) register usage, 6) the fraction of redundant computations, and 7) the load imbalance. Cost is the weighted sum of these factors. The function also ensures the dependence vectors between all stages of a group are constants after alignment and scaling of dependencies. The function determines the dimension sizes of the group, total threads created, threads per thread block, number of warps per thread block, and warp overlapped tile sizes. The function distribute all thread blocks equally across all SMs. Then the function retrieve the volume of each tile, the number of intermediate buffers, and multiply them with the number of warps per thread block to determine the shared memory usage per thread block.

If hybrid tiling is used, the function splits the shared memory tile into two parts and updates the register tile. The function checks if the shared memory used per thread block is more than the maximum shared memory.

Below I explain how value of each component of cost are calculated:

- **Number of Global Memory Transactions**: The function estimates the number of global memory transactions that either load input images or inputs to the group by retrieving the global memory addresses based on the input accesses and coalescing these accesses to minimum number of transactions.

- **Theoretical Occupancy**: The function estimates theoretical occupancy based on the shared memory and register utilization by calculating minimum of two occupancies. First occupancy is obtained from shared memory usage per thread block and second is occupancy obtained from register usage per thread.

- **Achieved Occupancy**: The function estimates the number of warps ready to execute at runtime as the ratio of time spent in global memory loads to the time spent in computations.

- **Shared Memory and Register Usage**: The function calculates per thread block shared memory usage and register usage when all thread blocks are executing concurrently based on the occupancy.
- **Fraction of Redundant Computations**: The function determines the fraction of overlap in each tile.
- **Load imbalance**: The function minimizes the load imbalance due to when the number of thread blocks per SMs are not always a multiple of number of thread blocks executing concurrently per SM based on the occupancy.

3.6 Evaluation

I implemented above techniques in PolyMage-GPU, which is available at `https://bitbucket.com/abhjangda/polymage-gpu`. In this section, I investigate the following questions: 1) How fast is PolyMage-GPU's automatic loop fusion algorithm? 2) How does the *OTPW* execution model compare to the state-of-the-art? Full details of performance analysis can be found at [54].

Experimental Setup All experiments are performed on a system containing a 3.4 GHz, quad-core Intel i5-4670 CPU with 16GB RAM and two GPUs (each experiment uses a single GPU): an NVIDIA GTX 1080Ti and an NVIDIA Tesla V100 (Table 3.1 lists their key specifications). I use six canonical image processing applications that have appeared in prior work [18, 53, 75, 76, 90]. Table 3.2 reports the number of stages and the size of the input image for each benchmark. I compare our work to the manually-written schedules present in Halide repository [4]. The execution time that reported for each benchmark is the sum of execution time of all generated CUDA kernels (obtained using `nvprof`), and does not include host and device memory transfer time. Each benchmark is executed for three samples with each sample containing 100 runs. I report the minimum of the average running time for each sample.

Cost Function Weights The cost function in automatic fusion requires several weights that are GPU-dependent. I determine the best weights empirically using leave-one-out cross validation, since, there are small number of benchmarks. Table 3.3 shows the weights.

Benchmark	Stages	Image size (W×H×c)	Fusion
Unsharp Mask	4	4256×2832×3	0.05s
Harris Corner	11	4256×2832	0.15s
Bilateral Grid	7	2560×1536	0.02s
Multiscale Interpolate	49	2560×1536×3	10s
Camera Pipeline	32	2592×1968	17s
Pyramid Blend	44	3840×2160×3	28s

Table 3.2: For each benchmark, the number of stages, size of input, and time taken for loop fusion.

	w_1	w_2	w_3	w_4	w_5	w_6	w_7
GTX 1080Ti	50	0.5	45	20	2	100	1
Tesla V100	50	0.5	60	10	2	100	1

Table 3.3: Value of weights obtained for both GPUs.

3.6.1 Automatic Fusion Time

I first measure the time it takes for automatic fusion to process each benchmark program to find an optimal schedule. PolyMage-GPU uses Bounded DP Fusion [53] to search for (i) thread block sizes (as a multiple of `WarpSize`), and (ii) tile sizes from 1 to 32 in each dimension. The *Fusion* column in Table 3.2 shows the time taken, which ranges from less than a second to up to 30 seconds for benchmarks with a few dozen stages.

3.6.2 Performance Evaluation

I now evaluate the performance of code generated by PolyMage-GPU after applying both OTPW with hybrid tiling and the loop fusion algorithm[1]. I compare code generated by PolyMage-GPU is against the manually-written schedules present in the Halide repository [4]. However, I wrote the schedule for *Pyramid Blend*, since it was not available.

Table 3.4 shows the absolute execution times of PolyMage-GPU and Halide and the speedup of PolyMage-GPU over Halide on both GPUs. On every benchmark, PolyMage-GPU is at least as fast as Halide, and in many cases, significantly faster. PolyMage-GPU is faster than manually written schedules in Halide with a geomean speedup of 1.65× and 1.33× on the GTX 1080Ti and Tesla V100 respectively. In general, PolyMage-GPU outperforms Halide because its fusion algorithm

[1]The generated CUDA 10.0 is compiled using `nvcc -O3 -arch=compute_61 -code=sm_61` on the GTX 1080Ti and `nvcc -O3 -arch=compute_70 -code=sm_70` on the Tesla V100.

Benchmark	Halide		PolyMage-GPU		Speedup	
	1080Ti	V100	1080Ti	V100	1080Ti	V100
Unsharp Mask	1.50×	0.45×	1.00×	0.39×	1.50×	1.15×
Harris Corner	1.80×	0.45×	0.80×	0.29×	2.25×	1.55×
Bilateral Grid	0.40×	0.20×	0.32×	0.20×	1.25×	1.00×
Multiple Interpolate	1.65×	0.60×	1.26×	0.54×	1.31×	1.11×
Camera Pipeline	1.90×	0.36×	1.04×	0.30×	1.83×	1.23×
Pyramid Blend	5.80×	2.90×	2.90×	1.30×	2.00×	2.23×
Geomean					1.65×	1.33×

Table 3.4: Execution times (in ms) of benchmarks and speedup of PolyMage-GPU over Halide's manually written schedules.

chooses better thread block and tile sizes, and the runtime technique has lower synchronization cost, decreased shared memory usage, and improved occupancy. The only exception is the *Bilateral Grid* benchmark on V100, where Halide's manual schedules are competitive with PolyMage-GPU because Halide can fuse the histogram stage, which performs a reduction, with subsequent blurring stages [105], whereas PolyMage-GPU cannot.

3.7 Conclusion

This chapter presented new techniques for executing image processing programs efficiently on GPUs and a compiler from PolyMage DSL to generate efficient GPU code using these techniques. I show that PolyMage-GPU generated code outperforms state-of-the-art. PolyMage-GPU is is available at `http://bitbucket.com/abhijangda/polymage-gpu`.

CHAPTER 4

ACCELERATING GRAPH SAMPLING ON GPUS

Graph representation learning learn features of data instead of hand-engineering them. Graph representation learning is a fundamental step in domains such as social network analysis, recommendations, epidemiology, and more. Several algorithms for graph representation learning first *sample* the input graph to obtain mini-batches and then *train* a deep neural network (DNN) based on the samples. For example, DeepWalk [84] and node2vec [44] use variants of random walks. GraphSAGE [47], which Pinterest uses for recommendation [114], samples the k-hop neighborhood of a vertex and uses their attributes to learn an embedding for each vertex.

Although several systems effectively leverage GPUs for the DNN training step, the same is not true for the sampling step. Graph sampling takes a significant portion of total training time in real-world applications. Table 4.1 shows that graph sampling can take up to 62% of an epoch's time.[1] Hence, accelerating graph sampling is important to improve the end-to-end training time.

Input Graphs	**PPI**	**Reddit**
GraphSAGE [47]	51%	45%
FastGCN [29]	26%	52%
LADIES [122]	40%	62%
ClusterGCN [33]	4.1%	24%
GraphSAINT [116]	25%	30%
MVS [34]	24%	25%

Table 4.1: Fraction of time spent in graph sampling in training.

Since samples are drawn independently, graph sampling is an "embarrassingly parallel" problem that seems ideal for exploiting the parallelism of GPUs. However, for a GPU to provide peak performance, the algorithm must be carefully designed to ensure regular computation and memory accesses, which is challenging on irregular graphs. Several systems have been designed for

[1]Experiments performed on two 16-core Intel Xeon Silver CPUs and an NVIDIA Tesla V100 GPU.

random walks [113], graph mining [32, 52, 74, 106], and graph analytics [69, 80, 97, 111]. These systems consider samples (or subgraphs) as the fundamental unit of parallelism: they grow all samples in parallel by looking up the neighbors of the vertices of each sample. However, such an approach has two issues: (i) irregular memory accesses and divergent control flow because consecutive threads can access the neighbors of different vertices, and (ii) lower parallelism because computation on all vertices in a sample is performed serially by the thread responsible for growing the sample.

This chapter presents NextDoor, the first system to perform efficient graph sampling on GPUs. Figure 4.1 shows the architecture of NextDoor. NextDoor provides a high-level API that abstracts away the low-level details of implementing sampling on GPUs and enables ML experts to write efficient graph sampling algorithms with few lines of code. NextDoor introduces a new approach for parallel graph sampling I call *transit-parallelism*. In transit-parallelism, the fundamental unit of parallelism is a *transit vertex*, which is a vertex whose neighbors may be added to one or more samples of the graph. In transit-parallelism, each transit vertex is assigned to a group of threads such that each thread adds one neighbor of the transit vertex to one sample. This technique provides better GPU execution efficiency due to low warp divergence, coalesced global memory accesses, and caching of the transit vertex edges in low-latency shared memory. Thus the irregular computation on the graph is transformed to a regular computation. NextDoor effectively balances load across transit vertices. NextDoor achieves significant speedups over state-of-the-art systems for graph sampling and improves training time of existing GNN systems by upto 4.75×. NextDoor is available at `https://github.com/plasma-umass/NextDoor`.

The rest of this chapter is organized as follows. Section 4.1 presents the first abstraction to represent graph sampling applications. Section 4.2 presents the NextDoor API based on the abstraction of Section 4.1. Section 4.3 discusses the shortcomings of existing approaches for graph computations and presents a novel transit parallel approach to graph sampling. Section 4.4 presents the details of NextDoor. Section 4.5 evaluates NextDoor against existing systems using ten graph sampling applications and four graphs. Section 4.6 concludes this chapter.

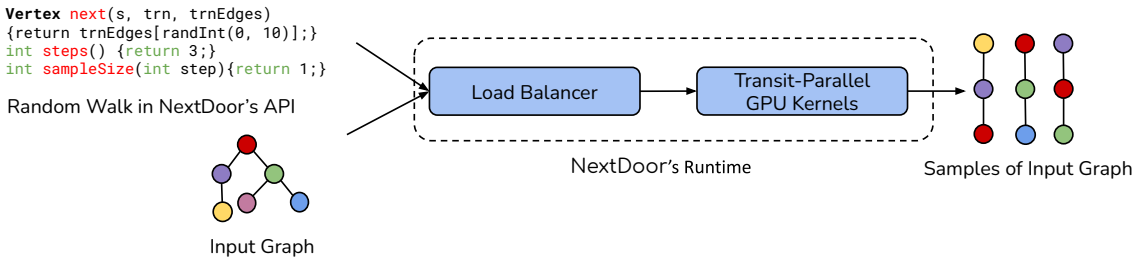


Figure 4.1: Overview of NextDoor. A user can define a sampling application, such as a random walk, in NextDoor's API. NextDoor will execute the application on an input graph using transit-parallelism based GPU kernels and improve the execution efficiency using load balancing techniques. NextDoor outputs the samples of input graph.

4.1 An Abstraction for Graph Sampling

This section introduces a general-purpose abstraction for graph sampling and uses it to express common sampling algorithms. The input to a graph sampling algorithm is a graph and an initial set of *samples*, where each sample is a subset of vertices (and optionally edges) of the graph. The algorithm iteratively grows each sample to include additional vertices in a series of *steps*, and its output is the final set of expanded samples. At each step, a sampling algorithm performs the following operations for each sample:

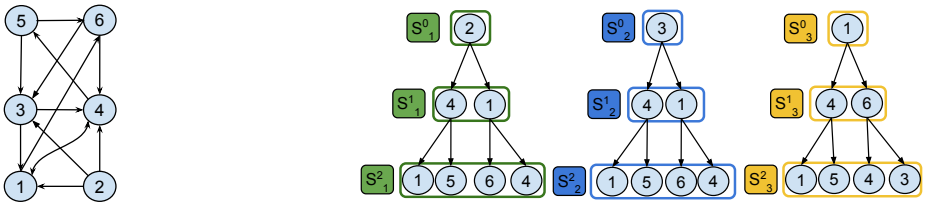
1. Iteratively sample one vertex at a time and add it to the sample. This operation can access the neighborhood of some vertices, which are referred as *transit* vertices in this thesis.

2. Determine the set of transit vertices for the next step.

A graph sampling application can be expressed by providing user-defined functions that describe how to perform these operations. Namely, the *next* function describes how to sample one new vertex. The *samplingType* function describes the granularity to sample new vertices based on two sampling types:

1. *Individual Transit Sampling:* the *next* function is executed for each transit a fixed number of times. It has access to the neighborhood of that transit.

2. *Collective Transit Sampling:* the *next* function is executed for each sample a fixed number of times. It has access to the combined neighborhood of all transit vertices



(a) Graph

(b) 2-hop Neighborhood sampling

Figure 4.2: Execution of a 2-hop Neighborhood sampling on graph in Figure 4.2a for samples S_1 to S_3. Initially each sample contain a single vertex. Output of sampling for each sample contains all vertices sampled at all steps.

Finally, the *stepTransits* function selects the vertices of the sample that will act as transit vertices in the next step. Other user-defined parameters are the number of steps k, which could be set to ∞ if it varies from sample to sample, and the maximum number m_i of new vertices sampled per transit vertex (for individual transit sampling) or per sample (for collective transit sampling) at step i. I below show that using these user-defined functions and parameters we can express a wide variety of sampling algorithms.

Random walks [44, 82, 84] A random walk application produces a set of random walks through the graph that start from some initial set of root vertices. In a static random walk, the probability of picking an edge is known beforehand, whereas in a dynamic random walk, the probability of picking an edge depends on properties of vertices that were previously visited on that walk. DeepWalk [84] performs fixed-size biased static random walks, where the probability of following an edge is proportional to the edge weight. Personalized Page Rank [48] performs a variable-size biased static random walk, where the probability of ending the random walk is defined by the user. In contrast, node2vec [44] is a dynamic random walk, which can be biased to stay closer to the starting vertex or to sample vertices that are further away.

This abstraction supports random walks as follows. Random walks are individual transit sampling applications because they sample a single neighbor of each transit vertex of the sample at each step and thus, every element m_i is 1. Since the transit vertex is the previously sampled vertex, $stepTransits$ function returns the previously sampled vertex in the sample. The root vertices are the initial samples, such that each sample is assigned one root vertex. The number of steps k describes the length of fixed-size random walks in algorithms like DeepWalk and node2vec. In these

algorithms, $next$ always returns a vertex. However, for applications that perform a variable-size walk, such as Personalized Page Rank, k is set to ∞. For termination, $next$ can decide to not add a new vertex to the sample. The walk for a sample ends when the sample has no more new transit vertices.

***k*-hop Neighborhood Sampling** [47] A k-hop neighborhood sampling algorithm employed in GraphSAGE [47] adds one or more neighbors of a transit vertex at each step. Figure 4.2b shows the execution of a 2-hop Neighborhood sampler that samples two neighbors of each transit at every step. This sampling is individual transit sampling and can be represented in above abstraction by setting $k = 2$, $m_1 = m_2 = 2$, having $stepTransits$ return all the vertices added in the previous step as transit, and having *next* uniformly choose neighbors of each transit vertex and add them to the sample. S_j^i denotes the vertices obtained after step i for sample S_j. Initially, S_1, S_2, and S_3 contains a single vertex. In the first step, the neighbors of transit vertices, i.e., ②, ③, and ① are added to the sample by both applications. In the second step, vertices sampled in first step becomes the transit vertices. Output of k-hop neighborhood sampling for each sample contains all vertices sampled at all steps.

4.2 Graph Sampling using NextDoor

This section presents NextDoor's API, which is based on the graph sampling abstraction presented in the previous section. The API allows users to write a variety of graph sampling algorithms in just a few lines of code.

4.2.1 Programming API

The inputs to NextDoor are a graph, an initial set of samples, and several user-defined functions (Figure 4.3), which we detail below. The output is an expanded set of samples. If desired, NextDoor can pick the initial set of samples automatically (e.g., select one random vertex per sample).

The user selects either collective transit or individual transit sampling using `samplingType`. The `stepTransits` function returns the transit vertices for a sample at a given step. In the individual transit sampling, number of transit vertices for each sample at step j are $\prod_{i=0}^{j} m_i$. In the collective transit sampling, number of transit vertices for each sample are m_{i-1}. This function

```
1 Vertex       next(Sample s, Vertex* transits, Edge* edges, int step);
2 int          steps();
3 int          sampleSize(int step);
4 bool         unique(int step);
5 Vertex       stepTransits(int step, Sample s, int transitIdx);
6 SamplingType samplingType();
```

Figure 4.3: User defined functions required to implement a graph sampling application in NextDoor

takes three arguments: 1) the step (`step`), 2) the sample (`s`), and 3) the index of transit out of all transits to return (`transitIdx`). The user must also define a sampling function to use at each step of the computation (`next`). This function receives four arguments: 1) the sample (`s`), 2) the source edge set to sample neighbors from (`edges`), 3) transit vertices (`transits`) forming the source edge set, and 4) the current step (`step`). If the sampling is individual transit sampling then `transits` contains only a single transit vertex and `edges` contains the edges of this transit vertex. Otherwise, `transits` contains all transit vertices of the sample and `edges` contains edges of all transit vertices. The result of `next` must be a vertex to add to s (or a constant NULL that indicates not to add a neighbor). The function `s.prevVertex(i, pos)` returns the vertex added at position `pos` of the last i^{th} step, and the function `s.prevEdges(i, pos)` returns the edges of that vertex. This information is necessary for applications, such as node2vec. The `steps` function defines the number of computational steps in the application (k). For applications that do not run for a fixed number of steps, such as Personalized Page Rank and Layer Sampling, they can return a special constant `INF` and the sampling process for a sample is stopped when no new transit vertices are added to the sample. The value returned by the `sampleSize` function determines how many times the `next` function is invoked on each individual or collective neighborhood for each sample at each step. The `unique` function specifies if at a step the sample should contain only unique vertices. The `Vertex` class has utility methods for computing the vertex degree, the maximum weight of all edges (`maxEdgeWeight`), and the prefix sum of all edges' weights. Users can extend the class to include application-specific vertex attributes to be added to the samples.

```
Vertex next(s, trns, edges, step) {
  Vertex t = s.prevVertex(2,0);
  float p = 2.0, q = 0.5;
  float maxW = trns[0].maxEdgeWeight();
  return rejection-smpl (trns[0],
        edges, maxW, t, t.edges, p, q);}
int steps()
{return 100;}
int sampleSize(step)
{return 1;}
bool unique(step)
{return false;}
Vertex stepTransits(step, s, transitIdx)
{return s.prevVertex(1, transitIdx);}
SamplingType samplingType()
{return Individual;}
```

(a) node2vec random walk of length 100

```
Vertex next(s, trns, edges, step) {
  int idx = randInt(0, edges.size());
  return srcEdges[idx];}


int steps()
{return 2;}
int sampleSize(step)
{return (step == 0) ? 25 : 10;}
bool unique(step)
{return false;}
Vertex stepTransits(step, s, transitIdx)
{return s.prevVertex(1, transitIdx);}
SamplingType samplingType()
{return SamplingType::Individual;}
```

(b) GrapSAGE's 2-hop neighbors

Figure 4.4: Use Cases of NextDoor

Output format NextDoor supports two output formats based on the application. 1) NextDoor can return an array of samples, such that each sample contains all transit vertices sampled at all steps. This format is required by GNNs that use random walks and layer sampling. 2) NextDoor can return vertices sampled at each step in an individual array. This format is required by GNNs that uses k-hop neighborhood sampling. The arrays are stored in the GPU in both cases.

4.2.2 Use Cases

I now present the implementation of node2vec and k-Neighborhood graph sampling algorithms using NextDoor.

node2vec The node2vec algorithm is a second-order random walk. Let v is transit vertex and t be the transit vertex of last step. The probability of picking edge (v, u) depends on hyperparameters p and q, and is determined using three cases: (i) if $u = t$ then the probability is p, (ii) if $u \neq t$ and u is a neighbor of t then the probability is $1/q$, or (iii) if $u \neq t$ and u is not a neighbor of t then the probability is 1. The next vertex is sampled using rejection sampling, which takes these parameters as input [113].

Figure 4.4a presents node2vec in NextDoor. The argument `transits` of `next` contains one transit vertex, since random walk is individual transit sampling. Parameters p and q can be

returned by a user-defined function or added as constants. `next` performs rejection sampling (`rejection-smpl`), the details of which are discussed in [113]. `stepTransits` returns the vertex added at previous step. `sampleSize` returns 1 because we add only one neighbor of transit at each step. `steps` returns the length of walk, i.e., 100.

k-hop neighbors Figure 4.4b implements GraphSAGE's 2-hop neighborhood sampler in NextDoor. `stepTransits` returns the vertices added at previous step as transits. `next` retrieves the transit vertex for this sample in `transit` variable, and choose a neighbor of the transit vertex. Since this is a 2-hop sampling, `steps` returns 2. GraphSAGE [47] sets the number of neighbors as $m_1 = 10$ and $m_2 = 25$, as reflected in `sampleSize`. MVS [34] is implemented in a similar way as it obtains 1-hop neighbors of all initial vertices in the sample.

4.3 Paradigms for Graph Sampling on GPUs

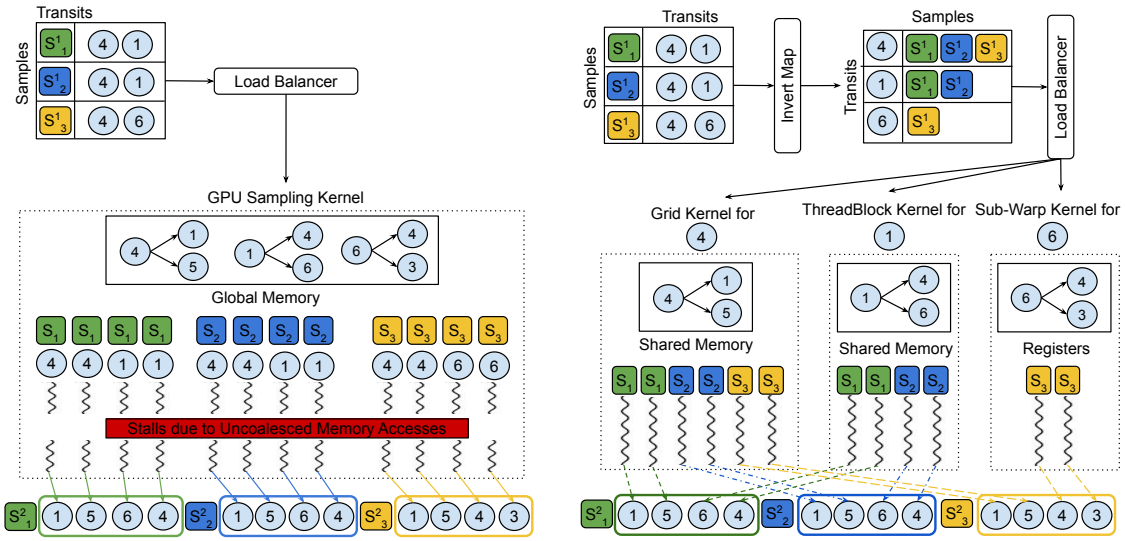
This section presents two paradigms for parallel graph sampling: sample-parallelism, which is used by existing systems for graph sampling [29, 33, 34, 47, 113, 116, 122], and transit-parallelism, which solves shortcomings of sample-parallelism.

4.3.1 Sample-Parallelism

Graph sampling is an "embarrassingly parallel" problem and the natural approach to parallelization is to process each sample in parallel, which we call the *sample parallel paradigm*. I now discuss the details of this approach for performing both individual and collective transit sampling.

Individual Transit Sampling The NextDoor API enables a new, fine-grained approach to sample parallelism. At each step i, we assign consecutive m_i threads to a pair of sample and transit. Each thread then calls the user-defined function (`next`) on its transit. The algorithm visits samples and their transits in parallel. Figure 4.5a shows an example of sample parallel execution for the second step of Figure 4.2b. In this example, each sample is assigned to a thread block containing four threads. Each thread samples one vertex for the assigned transit and writes this vertex to the output.

Collective Transit Sampling For collective transit sampling, before calling the `next` function on the combined neighborhood of all transits, this neighborhood must be completed. In a sam-



(a) Sample Parallel execution of second step. Each pair of sample and transit is assigned to contiguous threads. Each thread writes one vertex to the sample. Since contiguous threads can access different transits, we have divergent control flow and no locality.

(b) Transit Parallel execution of second step. Each transit vertex is assigned to a group of threads: Grid, Thread Block, or Warp. Each thread writes a neighbor of the transit vertex to the sample achieving non-divergent control flow and locality.

Figure 4.5: Execution of second step for 2-hop neighbor sampling of Figure 4.2b using Sample Parallelism and Transit Parallelism.

ple parallel approach, the combined neighborhood is computed in the same way as the individual transit sampling is performed, where threads copy the neighbors of one transit to the combined neighborhood, which is stored in global memory. After computing the combined neighborhood, each pair of sample and transit is assigned to m_i consecutive threads and call `next` on this neighborhood.

Limitations Sample parallelism makes poor use of the GPU for the following reasons. 1) In an individual transit sampling, for each sample, the algorithm calls `next` on the neighbors of several transit vertices in parallel. However, if two threads in a warp are assigned to process two distinct transit vertices with different numbers of neighbors, the thread processing the smaller set of neighbors may stall until the other thread completes. Thus the algorithm suffers from warp divergence. Similarly, there is warp divergence when computing the combined neighborhood. 2) The algorithm also suffers from poor load balancing. The amount of work done by `next` is likely to depend on the number of neighbors of the transit vertex. For example, while computing combined

neighborhood in collective transit sampling, different number of neighbors of each transit vertex leads to load imbalance within a thread block. 3) The graph must be stored in global memory, so accessing neighbors of transit vertex incurs high latency. Moreover, threads in a block may access the neighbors of different transit vertices, which leads to no locality. Hence, the GPU cannot coalesce the reads and cannot cache neighbors in its shared memory.

For example, the execution in Figure 4.5a suffers from two of the above issues. Since all four threads do not process same transit, there is divergent control flow and the adjacency list must be stored in global memory, leading to lack of locality among all threads of a thread block.

4.3.2 Transit-Parallelism

To overcome the limitations of sample parallelism, I present the *transit parallel* paradigm. Transit parallelism groups all samples with same transit vertex and process all samples for one transit vertex by assigning these samples to consecutive threads. This approach exposes regularity in sampling.

At each step the transit parallel paradigm works as follows. Before running sampling on the GPU, the paradigm create a map of transit vertices to their samples by grouping all samples associated with same transit vertex. The paradigm assigns each transit vertex to a group of threads, which may be organized as a grid, thread block, or warp. In individual transit sampling, each sample is assigned to consecutive threads in the group, and each thread calls `next` to add one neighbor of the transit to its sample. Similarly, in collective transit sampling, the paradigm create the combined neighborhood of the transits by assigning each sample to consecutive threads in the transit group (grid, thread block, or warp), and consecutive threads in the group add neighbors of the transit to the combined neighborhood of the sample. Building the combined neighborhood in a sample-parallel manner takes a significant portion of execution time. NextDoor speeds up this step by using the transit parallel approach. In this case, instead of sampling new vertices from the neighborhood of each transit of the sample using `next`, the system adds the entire neighborhood to the combined neighborhood. Collective sampling applications then select new vertices from the combined neighborhood per sample.

Figure 4.5b shows the execution of the second step of 2-hop neighbor sampling of Figure 4.2a in NextDoor using the transit parallel paradigm. According to its load balancing approach (Sec-

tion 4.4), NextDoor assigns transit ④ to a grid, such that all thread blocks in that grid are assigned samples of ④ and each thread adds one neighbor of ④ to one sample of ④, i.e., either S_1, S_2, or S_3. Similarly, it assigns vertex ① to a thread block and each thread adds one neighbor of ① to one of the samples associated with ①.

Advantages The transit parallel paradigm has two advantages for both individual and collective transit sampling: 1) contiguous threads perform a similar amount of work, because each thread calls the next function with same neighbors, and 2) contiguous threads accesses the neighbors of same transit. Both advantages ensure non-divergent control flow, and locality of memory accesses. This eliminates warp divergence and addresses load balancing [2]. Moreover, since all threads work with same neighbors, caching the neighbors in the shared memory will speed up later accesses.

For example, in the execution shown in Figure 4.5b each transit is assigned to one group of threads and this group caches the neighbors of each transit vertex in either shared memory or registers. Furthermore, each thread calls the *next* function on the same set of neighbors, which ensures non-divergent control flow among contiguous threads.

4.4 Efficient Transit Parallelism on GPUs

NextDoor implements transit parallelism on a GPU, with CPU-based coordination. This section first describe the techniques that allow NextDoor to execute individual transit sampling applications efficiently and then describe how NextDoor uses same techniques for executing collective transit sampling applications.

4.4.1 Sampling in Individual Transit Sampling

This section describes how NextDoor executes individual transit applications using transit parallelism.

4.4.1.1 Leveraging Warp-Level Parallelism

A GPU can *coalesce* several global memory accesses together into one memory transaction only if threads in a warp access consecutive addresses. The transit parallel paradigm lends itself

[2] A badly-written user-defined function may have these issues, but NextDoor avoids them in the core algorithm.

to a GPU implementation that supports coalescing *reads* to global memory, by having consecutive threads read the same adjacency list (i.e., of the shared transit vertex).

However, coalescing *writes* of new vertices to samples requires extra care. A two-level transit parallel approach maps different transit vertices to thread blocks and different samples to threads. This does not result in coalesced writes, since threads in the same warp add vertices to different samples. Instead, NextDoor uses three levels of parallelism: transits to thread blocks, samples to warps, and a single execution of the `next` function to a thread. Thus each thread writes one vertex to its sample and all threads in the warp issue one coalesced write to the same sample. Figure 4.5b shows this mapping as follows. First, transits ④, ①, and ⑥ are mapped to a group of threads. Then, samples (S_1, S_2, S_3) are mapped to subwarps and each thread executes `next`.

Sub-warps In an ideal scenario, there would be a one-to-one relationship between warps and samples, which would ensure that each thread in a warp writes to the same sample, using a single coalesced transaction to the global memory. However, there is a fixed number of threads per warp (usually 32) and this number can sometimes be larger than the required number of executions of the `next` function. Instead of letting threads be idle, NextDoor shares same warp among several samples. This yields some advantages. Suppose a warp of 32 threads is shared among 4 samples with each sampling having 8 contiguous threads. Then writes to the samples only generate 4 memory transactions rather than the 32 that we would obtain by assigning each thread to different sample. This also does not lead to warp divergence because all threads in a warp sample neighbors of the same transit vertex.

The term *sub-warp* here refers to a set of contiguous threads of same warp assigned to same sample. NextDoor uses sub warps as a fundamental unit of resource scheduling. All sub warps have the same size, which is determined using `sampleSize` function for the current step. Threads of the same sub-warp share the information of their registers using *warp shuffles*, and coordinate using *syncwarp*.

4.4.1.2 Load Balancing

In the transit parallel paradigm, each transit vertex is associated with a set of samples, which varies among transit vertices and steps. With three levels of parallelism, a transit vertex requires as many threads in a step as the total number of neighbors that will be added to its samples. Since

each transit can require different number of threads, there is load imbalance if same number of threads are assigned to all transits. To address this problem, NextDoor uses three types of GPU kernel:

1. The *sub-warp* kernel processes several transit vertices in a single warp. It is only applicable to transit vertices that require fewer threads than the maximum warp size (32).

2. The *thread block* kernel dedicates a thread block to a single transit vertex. It is only applicable to transit vertices that require more threads than in a warp, but less than the maximum thread block size (1,024).

3. The *grid* kernel processes a single transit vertex in several thread blocks. It is only applicable to transit vertices that requires more than 1,024 threads.

Scheduling To assign transits to kernels, NextDoor creates a *scheduling index* for each transit vertex. Creating a scheduling index involves three stages. First, NextDoor creates a transit-to-sample map based on the transits obtained from $stepTransits$ function (Figure 4.5b). Then, NextDoor partitions all transit vertices into three sets based on the number of samples associated with each transit vertex using parallel scan operations. Finally, the scheduling index of a transit vertex is set to the index of the transit vertex in its set. After picking a kernel type for a transit vertex, NextDoor assigns each sample of the transit vertex to a sub-warp in the kernel based on the thread index.

Caching NextDoor uses different caching strategies for different kernels to minimize memory access costs. When sampling neighbors of transit vertices in the *grid* and *thread block* kernels, the thread blocks for these kernels load the neighbors of transit vertices into the shared memory. However, when the neighbors do not fit in shared memory, NextDoor transparently loads neighbors from global memory. For transit vertices assigned to a *sub-warp*, NextDoor utilizes both shared memory and thread local registers to store neighbors. In this case, NextDoor transparently manages accesses to the neighbor list using *warp shuffle* instructions that allows consecutive threads to read neighbors from each others' registers. In summary, NextDoor uses the fastest caching mechanisms available for each kernel.

4.4.2 Transit-Parallel Collective Transit Sampling

Collective transit sampling applications require computing the combined neighborhood of all the transits of each sample. This is a potential performance bottleneck, so NextDoor uses transit parallelism to speed up the process. It constructs the combined neighborhood as if it were an individual transit sampling application that runs for only one step. Instead of sampling new vertices from the neighborhood of one transit, NextDoor adds all the vertices in the neighborhood to the combined neighborhood of the sample. After building a single combined transit neighborhood per sample, one could in principle detect which samples have the same combined neighborhoods and expand all these samples in a transit-parallel manner. The likelihood of two combined neighborhood being equal, however, is generally low, and detecting which samples have the same combined neighborhood is expensive. Therefore, NextDoor adds new vertices to the sample using a sample-parallel approach.

4.5 Evaluation

Benchmarks The graph sampling applications mentioned in Section 4.2 are used as benchmarks for the experiments. Applications' parameters are set as follows. For *PPR* the termination probability is set to 1/100, i.e., mean length is 100. For all other random walks, the walk length is set to 100. For *node2vec* p and q are set to 2.0 and 0.5 respectively. For these random walks, initially there is one vertex per sample. For *MultiDimensional Random Walk* (MultiRW), each sample contains 100 root vertices. GraphSAGE [47]'s hyperparameters are used for *k-hop Neighborhood Sampling*, i.e., $k = 2$, $m_1 = 25$, and $m_2 = 10$. For *Layer Sampling* final sample size is set to 2000 and step size for all steps is set to 1000. For *FastGCN*, *LADIES*, and *MVS* Sampling batch size and step size are set to 64. For *ClusterGCN Sampling* vertices are randomly assigned in clusters and each sample contains 20 clusters.

Datasets Table 4.2 lists the details of real world graphs used in the evaluation obtained from Stanford Network Analysis Project [65]. Weighted version of these graphs is generated by assigning weights to each edge randomly from [1, 5).

Experimental setup The experimental system contains two 16-core Intel Xeon(R) Silver 4216 CPU, 128 GB RAM, and an NVIDIA Tesla V100 GPU with 16GB memory running Ubuntu 18.04. I

Name	Abrv	# of Nodes	# of Edges	Avg Degree
Protein-Protein Interactions	PPI	50K	1.4M	28.0
com-Orkut	Orkut	3M	117M	39.0
cit-Patents	Patents	3.77M	16.5M	4.37
soc-LiveJournal1	LiveJ	4.8M	68.9M	14.3

Table 4.2: Graph used in our evaluation.

report the average time of 10 executions. The execution time contains time spent on GPU, which includes the time spent in sampling and creating the scheduling index.

4.5.1 Graph Sampling Performance

This section presents the results of comparison of NextDoor with the following baselines.

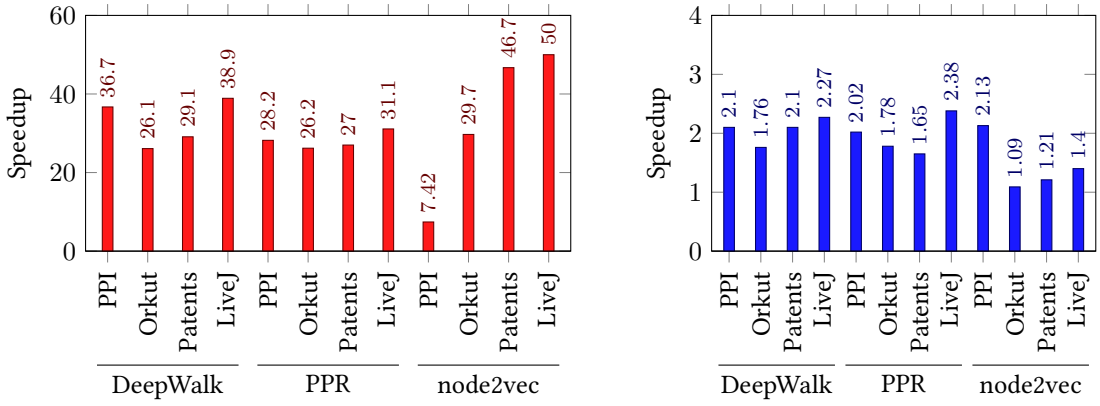
SP is the optimized sample-parallel graph sampling system based on the NextDoor API.

KnightKing [113] is a state of the art system for doing random walks using CPUs. It uses rejection sampling as a technique to select new vertices of a random walk and supports sampling using distributed systems. Its API restricts expressing only random walks, hence, KnightKing is a baseline only for random walks.

Existing GNN Samplers I compare NextDoor against the samplers of existing GNNs. These samplers are written for TensorFlow or numpy and are designed to run only on multi-core CPUs, not GPUs. This is because sampling is an irregular computation that is more easily implemented on CPUs. For k-hop neighborhood, I compare against GraphSAGE's sampler [47]. For MultiRW, I compare against GraphSAINT's sampler [116]. For sampling algorithms in FastGCN [29], ClusterGCN [33], MVS [34], and LADIES [122], I compare against samplers in their reference implementations.

Performance Results NextDoor provides an order of magnitude speedup over KnightKing (Figure 4.6a) for all random walk applications, with speedups ranging from 26.1× to 50×. NextDoor provides an order of magnitude speedup over the implementations of existing GNNs (Figure 4.7a). These large speedups are possible due to the massive parallelism and memory access latency hiding capabilities of the GPU. Furthermore, SP is significantly faster than all baselines.

NextDoor provides significant speedups over SP on all graph sampling applications, with speedups ranging from 1.09× to 5.1×. The speedup depends significantly on the application. For



(a) Speedup over KnightKing.

(b) Speedup over SP.

Figure 4.6: Speedup of NextDoor on random walk applications and real world graphs over KnightKing.

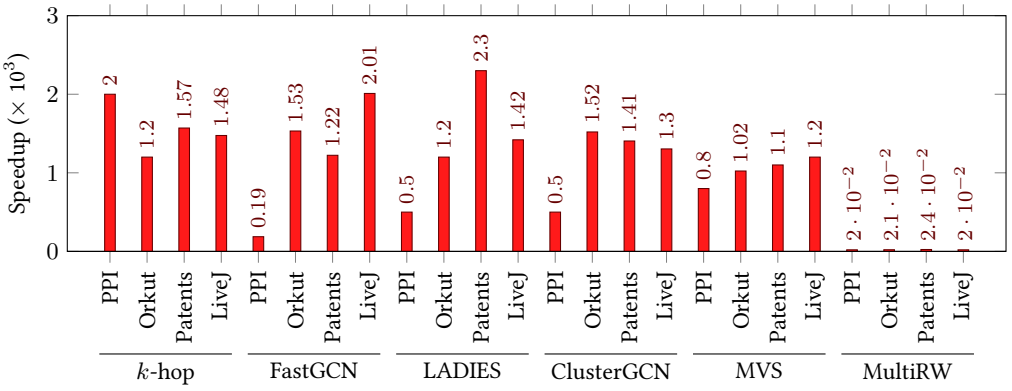
GNNs	PPI	Reddit	Orkut	Patents	LiveJ
GraphSAGE	1.30×	1.21×	OOM	1.20×	1.22×
FastGCN	1.25×	1.52×	4.75×	2.3×	4.31×
LADIES	1.07×	1.37×	2.27×	2.1×	2.34×
ClusterGCN	1.03×	1.20×	OOM	1.4×	1.51×

Table 4.3: End-to-end speedups after integrating NextDoor in GNNs over vanilla GNNs.

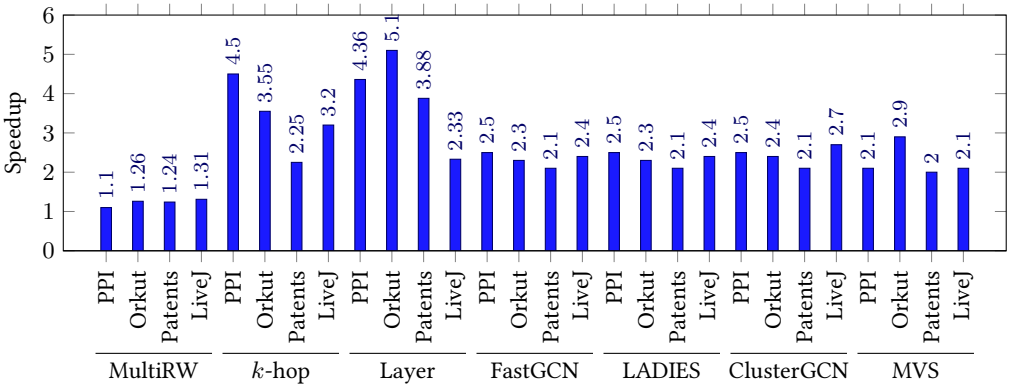
example, NextDoor obtains more speedup in DeepWalk and PPR than in node2vec because in node2vec at each step, for an edge from current transit vertex v to a vertex u, the algorithm might do a search over the edges of the previous transit vertex t to check if u is a neighbor of t, leading to memory accesses and warp divergence. Nevertheless, NextDoor still obtains speedup due to its transit-parallel paradigm. NextDoor achieves speedup over SP in all applications because NextDoor uses three levels of parallelism while SP can use only two levels of parallelism. Moreover, with FastGCN and LADIES, NextDoor is faster because it speeds up the computation of the combined neighborhood.

4.5.2 End-to-End Integration in GNN Systems

I performed an end-to-end evaluation of existing GNNs by replacing their sampler with the sampling implementation in NextDoor. Table 4.3 shows the performance improvement of our integration. The speedup for GraphSAGE is less than the maximum possible improvement in



(a) Speedup over GNN implementations ($\times 10^3$).



(b) Speedup over SP.

Figure 4.7: Speedup of NextDoor on graph sampling applications and real world graphs over GNN systems.

Table 4.1 due to a limitation of Tensorflow, which does not allow creating a tensor on the GPU memory. Therefore, samples are copied to the CPU and then again to the GPU for training. For FastGCN and LADIES, the speedup increases with larger graphs because the sampling time depends on the number of vertices in the graph, while the training time per batch remains constant.

4.6 Conclusion

This chapter presented NextDoor, the first system to express graph sampling applications and execute them efficiently on multiple GPUs. I show that NextDoor can significantly improves training times of several Graph Neural Networks. NextDoor is available at `http://github.com/plasma-umass/nextdoor`.

CHAPTER 5

CO-OPTIMIZING COMPUTATION AND COMMUNICATION FOR DISTRIBUTED MACHINE LEARNING

As the trend towards larger machine-learning models continue, from BERT [38] with 340 million parameters, GPT-2 [88] with 1.5 billion parameters, to GPT-3 [26] with 175 billion parameters, model training and inferencing have to be distributed. Moreover, as the computations become resource hungry, optimizing for even the last percentage can have huge benefits in terms of time, energy, and money savings [10, 103].

In machine learning systems today, such as PyTorch [83], computation and communication are treated as independent abstractions implemented in different libraries. For instance, computation libraries, such as cuBLAS [7] and cuDNN [8], provide optimized tensor algebra operations, while communication libraries, like NVIDIA Collective Communications Library [15], provide high-performance collective communication operations, such as AllReduce. Thus, in machine learning applications built atop of such frameworks, the computation and communication operations are invoked separately.

While this separation allows independent optimization of computation and communication kernels, breaking this abstraction boundary can unlock new optimizations that are otherwise not feasible. However, manually writing these optimizations for each scenario is unproductive. Thus, this thesis presents CoCoNet[1], which is a DSL for writing optimized distributed machine learning programs. Figure 5.1 presents the overview of CoCoNet. CoCoNet includes a domain specific language (DSL) to express programs containing both computation and communication operations. Then, the autotuner applies transformations to optimize the program while keeping the algorithm unchanged, such as fusing AllReduce and Dropout into FusedAllReduce and overlapping this with MatMul. Inspired by Halide [89], CoCoNet includes a *scheduling* language to specify an execu-

[1] CoCoNet stands for "**Co**mmunication and **Co**mputation optimization for neural **Net**works.

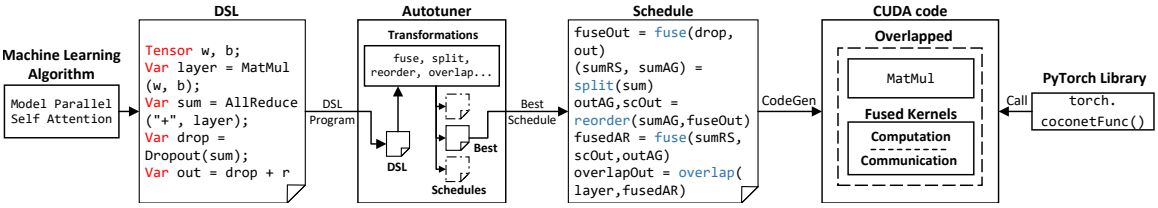


Figure 5.1: Overview of CoCoNet's workflow.

tion schedule of the program using a set of transformations. CoCoNet's *autotuner* automatically applies these transformations to optimize a program by breaking the communication and computation boundary. Hence, CoCoNet enables users to quickly generate optimized implementations for specific hardware, topology, and data sizes. CoCoNet's *code generator* automatically generates high-performance computation and communication kernels from a program and its schedule. CoCoNet generated optimized code is available through PyTorch. I used CoCoNet to optimize data-parallel training, model-parallel inference, and pipeline-parallel inference. CoCoNet generated kernels for the Adam [62] and LAMB [115] optimizers speeds up the training time of BERT models by upto 1.68×. CoCoNet's kernels for model parallelism speeds up the inference in BERT 3.9 Billion and GPT-2 8.2 Billion parameter models by upto 1.51×. CoCoNet's optimized pipeline parallelism kernels speeds up inference times in GPT-3 175 Billion parameter models by 1.33×. CoCoNet is available at `https://github.com/parasailteam/coconet`.

The rest of this chapter is organized as follows. Section 5.1 presents the syntax and semantics of CoCoNet. Section 5.2 presents several transformations provided by CoCoNet to optimize programs. Section 5.3 shows how to optimize data parallelism and pipeline parallelism using CoCoNet. Section 5.4 provides key details about the code generation process in CoCoNet. Section 5.5 evaluates CoCoNet against the state-of-the-arts over data-, model-, and pipeline-parallelism. Finally, Section 5.6 concludes this chapter.

5.1 The CoCoNet DSL

The CoCoNet DSL extends the data representation in existing machine learning frameworks and provides constructs to express both computation and communication. The CoCoNet DSL is embedded in C++. This chapter follows the MPI [40] terminology: RANK is the process ID of a

```
Tensor w (FP16, [H, H]   , Sliced(0) , WORLD, RANK);
Tensor b (FP16, [H]      , Replicated, WORLD);
Tensor in(FP16, [B, S, H], Sliced(2) , WORLD, RANK);
Tensor r (FP16, [B, S, H], Replicated, WORLD);

// layer(FP16, [B,S,H], Local, WORLD, RANK)
Var layer   = MatMul(in, w);
// sum(FP16, [B,S,H], Replicated, WORLD)
Var sum     = AllReduce("+", layer);
// dropout(FP16, [B,S,H], Replicated, WORLD)
Var dropout = Dropout(sum + b, 0.1);
// out(FP16, [B,S,H], Replicated, WORLD)
Var out     = dropout + r;

Execute self_attention({w, in, b, r}, {out});
```

Figure 5.2: An example program in CoCoNet. (B: batch size, S: sequence length, H: hidden dimension size)

distributed process, GROUP is a set of concurrent distributed processes, and WORLD is the GROUP that includes all processes.

5.1.1 Tensor Layout

CoCoNet extends the concept of a tensor in machine learning frameworks from a single device data into distributed forms. Besides item datatype, like `FP32` and `FP16`, and shape, a CoCoNet tensor also includes a *layout* that describes the distributed allocation of tensor's data across a set of ranks. There are three layouts for a tensor: *sliced*, *replicated*, and *local*. A *sliced* tensor is equally distributed among all nodes in a group along a specified dimension with RANK identifying the slice for that process. For example, in Figure 5.2, which describes the Megatron-LM [101] model parallel logic of Self-Attention layer in CoCoNet, `w` is sliced among all ranks in WORLD in the first dimension and `in` is sliced in the third dimension. A tensor can also be *replicated* across all ranks in a group where it has the same value on each rank and it does not have a rank identifier. For example, the bias `b` and the residual connection `r` are replicated as shown in Figure 5.2. A *local* tensor has same shape on all ranks but different values on all ranks. A local tensor requires RANK to identify the values. For example, in Figure 5.2, `layer` is a local tensor that represents the result of MatMul operation. A `Scalar` is a zero-dimensional tensor that represents a variable available on all ranks. The next section presents the layout of intermediate tensors.

5.1.2 CoCoNet's Operations

A CoCoNet program is represented as a data-flow graph (DFG) with operations as vertices and data dependencies as edges. Operations in CoCoNet can be classified as (i) local computations, such as pointwise computations, matrix multiplication, and convolution, and (ii) cross rank communication operations, such as AllReduce, AllGather, and P2P Send-Recv. CoCoNet supports all common communication and computation operations.

A `Var` represents the intermediate tensor obtained after performing an operation. A `Var`'s shape and distribution layout are inferred based on the operation and inputs to the operation. For example, in Figure 5.2 line 7 performs a MatMul operation on the input (`in`) and weights (`w`). Since MatMul between two sliced tensors produces a local tensor, `layer` represents the partial result with *local* layout. At line 9, AllReduce computes the sum of `layer` of all ranks and returns a *replicated* tensor with the same values on each rank. The computations at lines 11–13 add the bias, use dropout as an activation, and add the residual. At line 11, the addition of `sum` and `b` follows PyTorch's broadcast semantics[2] by replicating `b` in all dimensions of `sum`. Thus, the shape and layout of output of these operations are same as `sum`. Finally, `Execute` defines the name, inputs, and outputs of the program.

5.1.3 Fused Collective Communication Operations

CoCoNet enables efficient computations on the output of communication by providing fused collective communication operations, such as FusedAllReduce. Consider the AllReduce in Figure 5.2 followed by a Dropout (lines 9–11). The abstraction in existing machine learning frameworks requires the output of AllReduce to be stored in memory and then re-loaded by Dropout. FusedAllReduce avoids such stores and loads by directly passing the output of communication to following computations through registers. In addition to the argument of AllReduce, a FusedAllReduce takes computations as extra arguments. Section 5.4.2 discusses the implementation of Fused Collective Communication Operations.

[2]https://pytorch.org/docs/stable/notes/broadcasting.html

5.1.4 Overlapping Operations

CoCoNet supports overlapping multiple dependent computation and communication operations using the `Overlap` construct. For example, consecutive MatMul and AllReduce in Figure 5.2 (lines 7–9) can be overlapped to fully utilize both network and computation resources. Section 5.4.3 discusses the implementation of this construct.

5.2 CoCoNet Transformations

CoCoNet provides four semantics preserving *transformations* to optimize a program written in the DSL. All transformations are valid based on rules described in the sections below. CoCoNet automatically checks the validity of each transformation based on these rules and throws an error for an invalid transformation.

An order of transformations is refer as a *schedule*. A user can manually specify the schedule to optimize the program. Additionally, a user can invoke the autotuner to automatically find the best performing schedule for the given problem sizes and the underlying architecture. I present each transformation by applying them on the program from Figure 5.2.

5.2.1 Splitting Communication

The `split` transformation breaks a collective communication operation into two communication operations. One of the two split policies supported by CoCoNet is

AllReduce Split RS-AG splits an AllReduce into a ReduceScatter to produce a sliced tensor and an AllGather on the sliced tensor to return a replicated tensor.

Running Example The AllReduce in Figure 5.2 is split into `rsSum` that does a ReduceScatter on `layer` and `agSum` that does an AllGather on `rsSum`.

```
(rsSum, agSum) = split(layer, ARSplitRSAG);
```

Validity Since an AllReduce can always be split to a ReduceScatter and an AllGather, this transformation is always valid.

5.2.2 Reordering Operations

The `reorder` transformation swaps operations with an AllGather or a Broadcast in the DFG of a program. I explain this transformation for AllGather below:

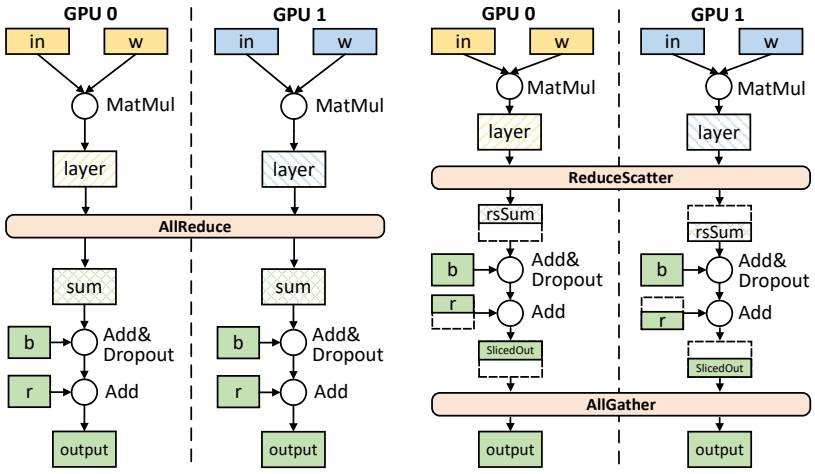


Figure 5.3: Equivalent programs (from Figure 5.2) using AllReduce (on left) or using ReduceScatter + AllGather (on right).

AllGather Reorder reorders an AllGather with communication and computation operations. This transformation changes the layout of the operations, the input and output of operations, and the input and output of the AllGather. I explain this transformation below using the running example.

Running Example In the program obtained from previous example, we can reorder the AllGather (`agSum`) with computations `d` and `out`. The reorder transformation replaces these operations in the DFG with three new operations: `scD` and `scOut`, both of which performs sliced computations, and `agOut`, which gathers the final result of computations.

```
(scD, scOut, agOut) = reorder(d, out, agSum, AGReorder);
```

The new sliced computations perform the same operations as original computations with two differences: (i) the output of AllGather used in the computation is replaced by the input of AllGather, and (ii) since the input of AllGather is sliced, all tensors input to the computations are also sliced along the same dimension as the input of AllGather. After reorder, `scD` performs the same computation as `d` but `scD` takes `rsSum` and `Slice(r)` as input. Therefore, the layout of `scOut` is also sliced while the computation is same as `out`. Furthermore, the new AllGather is

performed on the outputs of the computations, for example, after reorder, the AllGather (`agOut`) is performed on `scOut`. Figure 5.3 shows the workflow of this schedule.

Validity The `reorder` transformation is valid only if operations being reordered with an AllGather can be sliced along the dimension the AllGather is performed. The rules of slicing an operation depend on the type of operation and the dimensions of inputs to the operations. For example, `d` and `out` can be sliced because the computations have the same dimensions as `agOut`. Section 5.3 shows how P2P Send can be reordered with an AllGather.

5.2.3 Fusing Operations

Fusing multiple computations is a common technique used by existing compilers [31, 37, 46, 89**?**]. CoCoNet extends this concept to fuse multiple computations and communications in a single operation and provides this capability using the `fuse` transformation. Below I explain two fuse policies supported by CoCoNet:

Computation Fuse fuses a series of computations in a single operation that performs all these operations.

AllReduce Fuse fuses a series of ReduceScatter, sliced computations, and AllGather operations in a single FusedAllReduce that performs all these operations.

Running Example The fuse transformations enables fusing ReduceScatter (`rsSum`), computations (`scD` and `scOut`), and AllGather (`agOut`) into a FusedAllReduce.

```
fuseAR = fuse(rsSum, scOut, agOut, ARFuse);
```

The `comp` method of `fusedAR` specifies the computation to be fused with FusedAllReduce and returned `out` is the output.

Validity Fusing multiple operations into one operation is valid only if the dependencies in the DFG after fusion are preserved.

5.2.4 Overlapping Operations

CoCoNet provides the `overlap` transformation to overlap a series of producer-consumer operations to utilize multiple resources of hardware simultaneously.

Running Example Finally, the overlap transformation allows overlapping the matrix multiplication (`layer`) with FusedAllReduce (`fuseAR`).

```
Var avg = AllReduce("+", g);
Var m_  = Update(m, (m*beta1+(1-beta1)*avg));
Var v_  = Update(v, (v*beta2+(1-beta1)*avg*avg));
Var m1  = m_/(1-Pow(beta1, t));
Var v1  = v_/(1-Pow(beta2, t));
Var p_  = Update(p, (p - lr * m1/(Sqrt(v1))));

Execute adam({g,p,v,m,lr}, {p_});
```

(a) Traditional implementation where tensors g is local to each rank and p,m, and v are replicated on all ranks.

```
comps                    = fuse(m_, v_, m1, v1, p_, ComputationFuse);
(rsG, agG)               = split(avg, ARSplitRSAG);
(scComp, agP, agM, agV) = reorder(agG, comps, AGReorder);
asSlice(m); asSlice(v);    dead(agM); dead(agV);
fuseAR                   = fuse(rsG, scComp, agP, AllReduceFuse);
```

(b) An Optimized Schedule. Tensors g is local, p is replicated on all ranks, while m and v are sliced among all ranks.

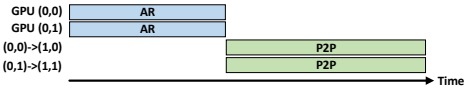
Figure 5.4: Optimizer parameter update using Adam in CoCoNet. The implementation takes four input tensors: parameters (p), gradients (g), momentum (m), and velocity (v).

```
layerWithAR = overlap(layer, fusedAR);
```

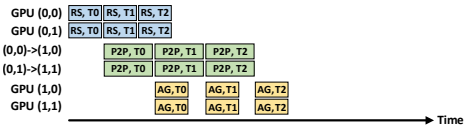
Validity Overlapping multiple operations is valid only when all operations have a producer-consumer relationship between them.

5.2.5 Automatic Exploration of Schedules

CoCoNet provides an *autotuner* to automatically explore the space of all schedules of a program and return the schedule that provides the best performance for the underlying architecture and input sizes. First, the autotuner fuses all pointwise computations up to a pre-defined threshold to decrease the search space and then exhaustively explores the schedule space in a breadth first search manner. Finally, the autotuner generates code for all schedules in its search space, executes all programs, and returns the schedule with minimum execution time. Table 5.2 shows that the autotuner takes only a few seconds to explore the schedule space for all workloads.



(a) In Megatron-LM each GPU sends redundant data.



(b) Communication operations can be overlapped at the granularity of each *communication buffer tile* of data in single kernel call.

Figure 5.5: Two different schedules of pipeline parallelism.

5.3 Distributed Workloads in CoCoNet

This section presents how to optimize two distributed machine learning workloads using CoCoNet: (i) parameter update using Adam [62], and (ii) point-to-point communication in pipeline parallelism.

Adam in Data Parallel Training: Figure 5.4a shows the traditional implementation of parameter update using Adam. First, all ranks average the gradients using AllReduce and then perform computations to update the optimizer state and model parameters. `Update` updates the values of a tensor and reflects the new values in that position in the DFG (lines 2–3). Figure 5.4b presents a schedule that optimizes this by distributing the computation on all ranks in a single kernel. Line 1 fuses all computations in `comps`. Line 2 splits the AllReduce into a ReduceScatter and an AllGather, such that computations take output of AllGather (`agG`) as input. Line 3 reorders AllGather with computations, such that, each rank performs computations on a slice of tensors. Line 4 slices optimizer states on all ranks to decrease memory usage and removes corresponding AllGather. Finally, line 5 fuses all operations in a single kernel.

Point-to-Point Communication in Pipeline Parallelism: Figure 5.5a shows a scenario of pipeline parallelism in Megatron-LM with two transformer layers assigned to two groups each with two ranks. Rank i in group j is shown by (j, i). Each group uses model parallelism within its transformer layer. Pipeline parallelism in Megatron-LM works as follows. First, all ranks in the first group reduce their input using AllReduce to get replicated output. Then each rank performs pointwise computations over the replicated output. Finally, the first group sends the result of computations to the corresponding rank in the second group using point-to-point (P2P) sends. (Line 2 in Figure 5.6a shows these computations but are omitted in Figure 5.5 for simplicity). Since the

```
Var sum    = AllReduce("+", in);
Var send   = Dropout(recv+b,0.1) + r;
Var output = Send(send, GroupRank(GROUP+1, RANK));

Execute transformer({in}, {output});
```

(a) Traditional implementation. Each rank of a group sends same data to next group.

```
1 fuseSend          = fuse(send, output, SendFuse);
2 (rsSum, agSum)    = split(sum, ARSplitRSAG);
3 (scSend, agOut)   = reorder(fuseSend, agSum, AGReorder);
4 overlapOut        = overlap(rsSum, scSend, agOut);
```

(b) An Optimized Schedule. Each rank sends only a slice of data to ranks in next group and all operations are overlapped.

Figure 5.6: Optimizing pipeline parallelism of Megatron-LM. Input tensors: layer output `in`, bias `b`, and residual `r`.

output of AllReduce in Figure 5.5a is replicated, redundant data is sent using P2P. We can avoid this redundant communication by splitting the AllReduce to ReduceScatter and AllGather and reordering the P2Ps with the AllGather. Hence, the inter-group communication is reduced by the group size. This workload can be further optimized by overlapping all communication operations. Figure 5.5b shows that if the buffers are split into multiple tiles (`T0`–`T2` in the figure), intra-group and inter-group communications can be overlapped.

Figure 5.6a is the original program, while Figure 5.6b optimizes it by applying transformations. Line 1 fuses the P2P send with computations. Line 2 splits the AllReduce and reorders the returned AllGather with the fused P2P send at line 3. Hence, P2P send and computations are performed on only a slice of data on the next group where the AllGather is also performed. Finally, all three new operations are overlapped in line 4.

5.4 The CoCoNet Code Generator

CoCoNet generates CUDA kernels for computation and communication operations for running on a distributed system with NVIDIA GPUs. For each operation, CoCoNet either generates (i) a call to a collective communication operation, (ii) a CUDA kernel for fused computations, (iii) a CUDA kernel for fused-collective communications (Section 5.4.2), or (iv) CUDA kernels for

overlapping of communication and computation operations (Section 5.4.3). Moreover, CoCoNet generates code for performing operations on multiple non-contiguous tensors (Section 5.4.4). After generating CUDA kernels, CoCoNet traverses the program's DFG to generate kernel calls. CoCoNet wraps generated programs as custom operators and integrates them into PyTorch, so that, applications like Megatron-LM can invoke them directly. I now discuss how CoCoNet adapts NVIDIA Collective Communication Library (NCCL), a widely-used hand-optimized high performance communication library, into a runtime to execute above CUDA kernels.

5.4.1 NCCL Architecture

NCCL communicates data stored in the global memory of one GPU to a memory location on another GPU using CUDA kernels. NCCL's CUDA kernels perform communication by directly copying data from memory of one GPU to another GPU using GPUDirect Remote Data Memory Access [11]. NCCL's architecture defines four key properties: (i) topology, (ii) protocols, (iii) channels, and (iv) threads in a thread block of the CUDA kernel. NCCL automatically sets key configuration values for these properties based on the size of the input buffer, network architecture, and the size of WORLD. To ensure good performance, CoCoNet's code generation must carefully reconfigure these properties when extending NCCL to custom communication and computation. We now provide a high level overview of these properties.

Topology NCCL creates logical topologies, such as ring and tree, over the underlying interconnect network.

Channels NCCL maps copies of a logical topology on the underlying interconnect network. Each copy is called a channel and is assigned to one CUDA thread block.

Protocols NCCL sends data using one of the three protocols: `LL`, `LL128`, and `Simple`. These protocols make different tradeoffs between latency and bandwidth based on the type of inter-node synchronization used: `LL` has the lowest latency and `Simple` provides the highest bandwidth.

Number of Threads NCCL sets a fixed number of threads for each channel (and thread block). NCCL's kernels have high register usage, which limits the number of thread blocks per SM to one.

NCCL Workflow After determining the topology, protocol, number of channels, and number of threads, NCCL calls its CUDA kernel for communication. Each collective communication has three

levels of tiling to fully utilize the massive parallelism of GPUs. Data is first divided into *buffer tiles* equal to the size of the communication buffer. Each buffer tile is further divided among all ranks and channels to obtain *chunks*. Each channel communicates a chunk of data at a time. The *threads* in channels copy elements in and out of the buffers and apply reduction operations (`sum`, `min`, `max`) if needed.

5.4.2 Fused Collective Communications

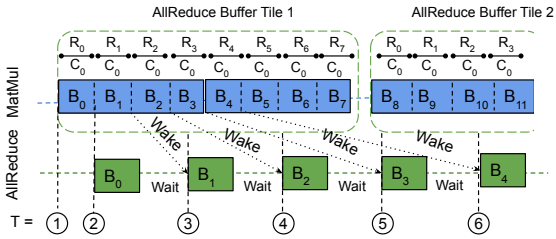
Fused Collective Communication extends NCCL's existing kernels to enable arbitrary pointwise computations and reductions. I inspected more than 10K lines of code in NCCL to identify where computations can be added to pass intermediate values from communication to fused computations directly through registers. CoCoNet supports fusion of both pointwise operations and reductions into NCCL collectives.

Each NCCL protocol utilizes a different mechanism for communication and CoCoNet generates code for all of them. The important features of a protocol are the pack type (64-bit for `LL`, 128-bit for `LL128` and `Simple`) and the load/store access pattern (shared memory for LL128, global memory for `LL` and `Simple`). CoCoNet generates template code for all element types in NCCL, and dispatches accordingly at runtime. There are some subtleties in the code generation worth discussing:

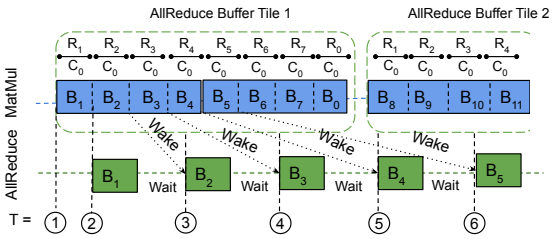
Mixed Precision When the element types of computations and the input tensors are different, CoCoNet finds the largest element type and based on the pack type of the protocol calculates how many elements can be loaded at once. CUDA code is generated to operate on these many elements.

Sliced Tensor When a sliced tensor is used by a fused collective communication, all memory accesses performed need to be mapped to elements of the sliced tensor. CoCoNet generates code that produces this mapping. To perform an AllGather on sliced tensors, the inverse of this mapping is produced.

Tensor Reduction To reduce a sliced tensor, each rank reduces locally and do an AllReduce. This AllReduce reuses already established connections among ranks in the surrounding communication kernel to avoid extra startup latency.



(a) Workflow of overlap on **rank 0**. Rank 0 starts with chunk 0.



(b) Workflow of overlap on **rank 1**. Rank 1 starts with chunk 1.

Figure 5.7: Workflow of CoCoNet's overlapping of MatMul with AllReduce for a Float 16 matrix [8192, 3072] on 8 ranks (R_0 to R_7) with 1 channel (C_0) and 16 MB buffer size. Size of each 2-D chunk (B_0 to B_{15}) is [1024, 1024].

5.4.3 Overlapping of Communication and Computation

Overlapping of computation and communication has been studied in the context of executing stencil computations in a distributed system [21, 23, 25, 36, 37, 63, 71, 73, 96, 104, 108]. These works use non-blocking MPI operations to communicate data and simultaneously perform computations on CPUs. A similar approach for overlapping of computation and communication operations for a GPU workload would involve dividing all operations into sub-operations and ensuring dependency between sub-operations using CUDA streams. However, this approach would provide sub-optimal performance because each sub-operation is performed on only a part of data, which leads to inefficient computation and under-utilization of communication bandwidth.

Figure 5.7 shows how the fine-grained overlapping of CoCoNet addresses this issue using the example of a MatMul followed by a ring AllReduce. First, it schedules the MatMul kernel (based on CUTLASS [9]) to produce chunks in the same order as the AllReduce consumes them. Here, the n^{th} rank sends chunks in the order starting from the n^{th} chunk. Hence, the MatMul kernel on n^{th} rank produces chunks in the same order. Second, CoCoNet invokes both kernels only once on different streams and synchronizes the AllReduce with the MatMul using an efficient fine-grained spin-lock on a memory buffer to ensure that the AllReduce wakes up as soon as the MatMul produces a chunk. Third, to provide opportunities to tune the 2-D tile sizes of the MatMul kernel, CoCoNet generates a 2-D AllReduce kernel that communicates 2-D chunks, while NCCL AllReduce only supports 1-D continuous chunk.

The example in Figure 5.7 works as follows. At T = ①, all ranks invoke MatMul and AllReduce kernels. On rank 0, after computing chunk 0, the MatMul kernel wakes the AllReduce kernel at T = ②, which starts communicating chunk 0. While on rank 1, at T = ② the MatMul kernel wakes the AllReduce kernel to communicate chunk 1. Concurrently, both MatMul kernels compute their corresponding next chunk. At T = ③, MatMul kernels finished computing chunk 1 on rank 0 and chunk 2 on rank 1 and wakes up corresponding AllReduce kernels to communicate these chunks. This process continues until all chunks are processed.

This process allows the MatMul kernel and AllReduce to be overlapped in a fine-grained manner, which reduces the startup latency of AllReduce. Since AllReduce communicates on the same chunk sizes, it achieves maximum communication bandwidth. Furthermore, the MatMul kernel achieves maximum efficiency because the kernel is invoked on the full matrix size.

5.4.4 Operations on Scattered Tensors

In data parallelism, communication and computation occur on different layers of widely different sizes. Since machine learning frameworks allocate parameters and gradients of layers in non-contiguous buffers, gradients are copied to a large buffer to avoid launching multiple AllReduce operations.

CoCoNet supports generating a single kernel for both computation and communication operations acting on non-contiguous tensors. In this section, we show how CoCoNet modifies NCCL to generate a single communication kernel for scattered tensors. This code generation is non-trivial because NCCL has several design decisions based on the assumption that it is communicating a single contiguous buffer. For example, each thread of a NCCL channel copies only a few elements in each iteration, and hence indexing the correct tensor at a particular offset requires a linear search through all non-contiguous tensors, which can lead to significant overhead. CoCoNet solves this problem by first dividing each tensor into buckets of size at most 2^{10} elements and then assigning buckets to warps in a round-robin manner. This mechanism allows each thread to quickly find the offset in a tensor, since a warp can directly index in its assigned bucket. CoCoNet pre-calculates the number of buckets that belong to the same contiguous buffer and calculates the offset for all of them only once.

Optimizer	Scattered Tensor	Single Tensor
Adam	33.89 ms	33.21 ms
LAMB	37.04 ms	36.71 ms

Table 5.1: Time to perform parameter update of all 360 tensors of BERT.

The process of breaking each tensor to buckets has computation overhead and extra memory requirements. Since this bucketing is done only once on the CPU and training tasks run for thousands of iterations on the same tensors, the computation overhead is negligible. Each bucket is represented by a pair of 64-bit tensor address and a 32-bit offset into the associated tensor, leading to $12 \times \lceil \frac{N}{2^{10}} \rceil$ bytes of extra memory for a tensor with N elements. However, this memory overhead is negligible for large models. For example, for BERT model with 334M elements, the memory requirement is 0.6%. Table 5.1 shows that the time to perform parameter update of all 360 tensors of BERT on 256 Tesla V100 with scattered tensors implementation and a single contiguous tensor of size equal to the sum of size of all tensors. These results shows that the overhead of scattered tensors is insignificant over contiguous tensors.

5.5 Evaluation

This section evaluates the effectiveness of CoCoNet through standalone experiments and end-to-end distributed machine learning scenarios of data, model, and pipeline parallelism.

All experiments are performed on a cluster of 16 NVIDIA DGX-2 nodes where each node contains dual 24-core Intel Xeon CPUs and 16 NVIDIA Tesla V100 (32GB) GPUs. Each GPU within a node is connected to six NVSwitches with six NVLinks (25 GBps per NVLink). Nodes are connected with 8 non-blocking EDR InfiniBand (100 Gbps) network. All nodes run Ubuntu 20.04, CUDA 11.3, cuDNN 8.2 and PyTorch 1.10.

5.5.1 Data Parallel Training

In data parallelism, communication involves an AllReduce of gradients among all ranks. The output is used by the optimizer to update the model parameters. We evaluate CoCoNet generated code for two widely-used optimizers, Adam and LAMB. All experiments in this section were performed on all 16 DGX-2 nodes in our cluster.

5.5.1.1 Standalone Experiments

We first perform standalone experiments to explore different CoCoNet schedules over a range of input tensors from 2^{10} to 2^{30} elements. The autotuner generates and executes implementations with different configurations, including all NCCL protocols and all channels from 2 to 64. For each tensor, the autotuner reports the best average result of 1000 iterations.

Baselines The baselines perform parameter update by first doing AllReduce over gradients and then call FusedAdam or FusedLAMB from NVIDIA Apex [6]. Both FusedAdam and FusedLAMB fuses all the parameter update computations.

CoCoNet Schedules The autotuner generates following three schedules of Adam and LAMB by applying different CoCoNet transformations for each input size and reports the best schedule to the user for each input size:

1. **AR-Opt** (Opt = Adam/LAMB) refer to the traditional parameter update technique, i.e., an AllReduce over gradients and then each GPU individually performs the optimizer computation. These schedules fuse all computations into a single kernel, thereby simulating the baseline implementations of FusedAdam and FusedLAMB.

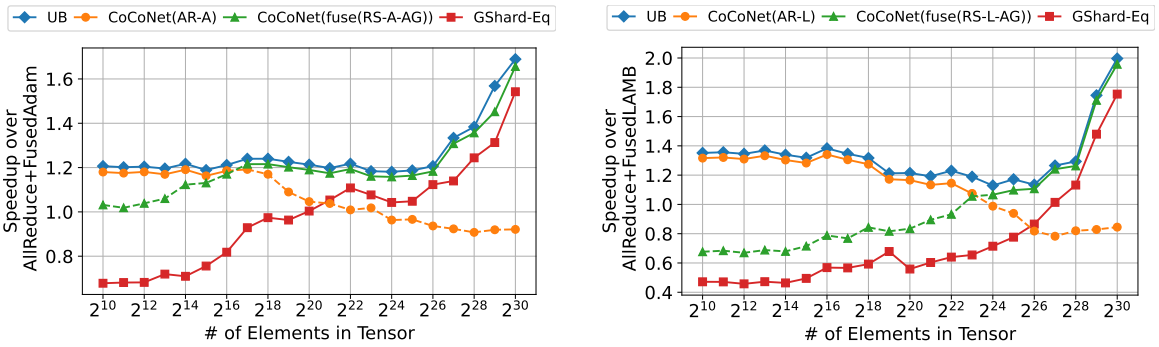
2. **GShard-Eq** or **RS-Opt-AG** (Opt = Adam/LAMB) are generated from *AR-Opt* by first splitting the AllReduce into ReduceScatter and AllGather, and then reordering AllGather with the fused optimizer computations. Hence, these schedules distribute parameter update across all ranks, similar to GShard [64] and ZeRO [91].

3. **fuse(RS-Opt-AG)** (Opt = Adam/LAMB) are generated by fusing all operations of *RS-Opt-AG* into FusedAllReduce.

5.5.1.1.1 Results

Figure 5.8 shows the speedup of CoCoNet schedules over the baseline for several tensor sizes. The results are shown for mixed-precision [13] using Float 16, and the results for Float 32 are qualitatively similar. In these figures, UB represents the cost of AllReduce alone without doing any computation, and thus is the upper bound of possible speedups.

Even though the *AR-Opt* schedules emulate the baseline implementations, they are faster on smaller tensors. This is because the baseline implementations perform additional preprocessing to optimize the amount of thread-parallelism and instruction-level parallelism per invocation. While



(a) Mixed-precision Adam.

(b) Mixed-precision LAMB.

Figure 5.8: CoCoNet speedup on 256 GPUs. For each size, CoCoNet runs the best schedules. UB (upper bound) takes AllReduce-only as max achievable speedup.

this preprocessing cost hurts smaller tensors, its benefit shows up for larger tensors where *AR-Opt* performs worse.

Since *GShard-Eq* and *fuse(RS-Opt-AG)* schedules distribute the optimizer computation, they perform better than the baseline for large tensors. The performance of *fuse(RS-Opt-AG)* shows the advantage of fusing computation and communication kernels as these schedules achieve near optimal speedups for large tensors. These schedules are respectively 13% and 14% faster than GShard-Eq for Adam and LAMB.

For smaller tensor sizes, multiple kernel calls are required for GShard-Eq schedules significantly hurt performance. Interestingly, *fuse(RS-Opt-AG)* schedules are slower than *AR-Opt* schedules for smaller tensor sizes though they require one less kernel call because the fused kernels have a higher register usage, thereby restricting the thread-level parallelism. This demonstrates that the fusion of communication and computation is not always a good idea.

Table 5.2 shows that the lines of generated code for each schedule are significantly more than the implementation in CoCoNet and the autotuner explored all schedules in 10 seconds. In summary, CoCoNet provides performance improvements over baselines with fewer lines of code. The *AR-Opt* and the *fuse(RS-Opt-AG)* reach close to optimal performance for smaller and larger tensors respectively. This amounts to a speedup of 1.2× to 1.7× for Adam and 1.35× to 2.0× for LAMB. There is no schedule that performs best for all sizes, which demonstrates the need for the autotuner.

Schedule	Generated CUDA	Program in CoCoNet	Autotuner Time
AR-Adam	16 lines	12 lines	
RS-Adam-AG	24 lines	16 lines	9 secs
fuse(RS-Adam-AG)	150 lines	17 lines	
AR-LAMB	80 lines	15 lines	
RS-LAMB-AG	140 lines	17 lines	10 secs
fuse(RS-LAMB-AG)	220 lines	18 lines	

Table 5.2: Lines of code of implementation of schedules of data parallel parameter update in CUDA and CoCoNet, and time taken by the autotuner to find the best schedule.

Optimizer	# of Parameters	Maximum Micro Batch Size		Speedup
		PyTorchDDP	CoCoNet	
Adam	336 M	32	32	1.22×
	1.2 B	8	32	1.52×
	3.9 B	OOM	8	–
LAMB	336M	64	128	1.20×
	1.2B	8	64	1.68×
	3.9B	OOM	8	–

Table 5.3: Maximum Micro Batch Size of implementations and speedup of CoCoNet over the PyTorchDDP when training BERT models using Adam and LAMB. OOM represents Out of Memory.

5.5.1.2 Integeration with BERT

I use CoCoNet generated optimizers to train three large BERT models from NVIDIA [16]. We integrated the scattered tensors implementation of *fuse(RS-Opt-AG)* schedule for both Adam and LAMB in PyTorch. PyTorch Distributed Data Parallel (PyTorchDDP) is the baseline for this experiment. PyTorch DDP [67] stores all gradients in buckets of 25MB and overlaps the AllReduce on each gradient bucket with computations during training. After reducing all gradients it calls FusedAdam or FusedLAMB. I use mixed precision training with both Adam with 8192 global batch size and LAMB with 65536 global batch size.

Results Table 5.3 shows the speedup provided by CoCoNet in training three BERT models over PyTorchDDP. For Adam optimizer, CoCoNet provides speedup over PyTorchDDP in training BERT 336M because CoCoNet's fused schedules perform better than FusedAdam. CoCoNet provides even higher speedup on BERT 1.2B models because the fused schedules decrease memory usage by distributing Adam's state over all GPUs, which improves the efficiency of matrix multiplication GPU kernels by enabling higher batch size per iteration. On 3.9B parameter model,

PyTorchDDP go Out of Memory, while CoCoNet's fused schedules can train the model. Results for LAMB are similar. CoCoNet provides up to 1.68× speedup over PyTorchDDP for LAMB.

5.5.2 Model Parallelism

Megatron-LM [101] uses a model parallel approach for inference and training of transformer models, such as BERT [38] and GPT-2 [88]. A transformer layer contains a self-attention block and a multi-layer perceptron (MLP) block. Last few operations of a self-attention block are the same computations as shown in Figure 5.2. An MLP block's last operations are similar to Figure 5.2 with the input tensor and weight sizes as $[B, S, 4 \times H]$ and $[4 \times H, H]$ (B, S, and H are batch size, sequence length, and hidden size, respectively). Since model parallelism is applied within one node, all experiments in this section are performed on a single NVIDIA DGX-2 node.

5.5.2.1 Standalone Experiments

I first perform standalone experiments to evaluate different schedules generated by the autotuner. I compare following schedules for model parallel self-attention code of Figure 5.2 and similar operations of multi-layer perceptron:

1. **MM-AR-C** is the baseline schedule of Figure 5.2. This schedule improves the Megatron-LM implementation by fusing all pointwise computations into one kernel.

2. **MM-RS-C-AG** This schedule is generated from *MM-AR-C* by splitting the AllReduce into a ReduceScatter and an AllGather, and reorders AllGather with computations.

3. **ol(MM,fuse(RS-C-AG)** is generated from the previous schedule by fusing the ReduceScatter, computation, and AllGather into a FusedAllReduce and then overlapping it with the MatMul. The autotuner returned this as the best schedule and hence represents CoCoNet in our results.

Results I evaluate these schedules with sizes of GPT-2 8.3 Billion parameter model (i.e., $S = 1024$, $H = 3072$) for 8 and 16 batch sizes. Figure 5.9 shows the times of all schedules normalized to the time of *MM-AR-C* schedule. CoCoNet's best schedule (*ol(MM,fuse(RS-C-AG))*) provides 1.38× to 1.62× speedup over *MM-AR-C* and 1.21× to 1.34× over *MM-RS-C-AG* because it overlaps FusedAllReduce with the matrix multiplication. Table 5.4 shows that the lines of generated

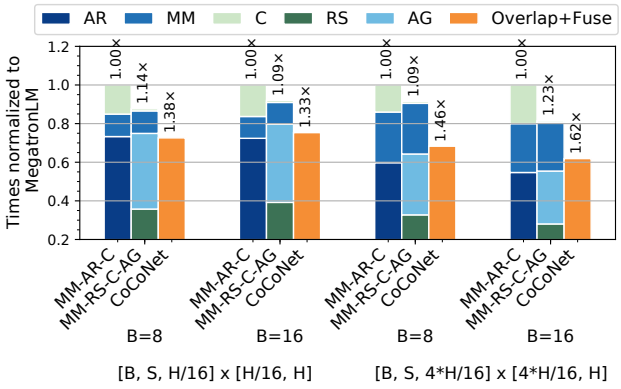


Figure 5.9: Times of CoCoNet's schedules of model parallel self-attention and multi-layer perceptron of GPT-2 normalized to baseline *MM-AR-C* schedule.

Schedule	Generated CUDA	Program in CoCoNet	Autotuner Time
MM-AR-C	20 lines	10 lines	
MM-RS-C-AG	140 lines	13 lines	12 secs
ol(MM,fuse(RS-C-AG))	≈ 2k lines	14 lines	

Table 5.4: Lines of code of implementation of schedules of model parallel Self Attention in CUDA and CoCoNet, and time taken by the autotuner to find the best schedule.

CUDA code for each schedule are significantly more than the implementation in CoCoNet and the autotuner explored all schedules in 12 seconds.

5.5.2.2 Integration with Megatron-LM

After integrating CoCoNet's overlap schedule in Megatron-LM, we found that CoCoNet improved inference times of BERT 3.9B parameter model by 1.51× and GPT-2 8.3B parameter model by 1.48×. Hence, overlapping matrix multiplication with fused collective communication significantly improves inference times.

5.5.3 Pipeline Parallelism

CoCoNet can decrease inference times in pipeline parallelism by fusing computation and communication and overlapping multiple communication operations. I evaluate CoCoNet on computations of model and pipeline parallelism in Megatron-LM for GPT-3 175B parameter models. A transformer layer contains several operations but the operations of interest for this experiment are

Schedule	Generated CUDA	Program in CoCoNet	Autotuner Time
AR-C-P2P	20 lines	10 lines	
AR-C-P2P-AG	20 lines	13 lines	11 secs
ol(RS,fuse(C-P2P),AG)	$\approx$ 2k lines	14 lines	

Table 5.5: Lines of code of implementation of schedules of pipeline parallel transformer layer in CUDA and CoCoNet, and time taken by the autotuner to find the best schedule.

presented in Figure 5.6a. All experiments in this section are performed on all 16 NVIDIA DGX-2 nodes.

5.5.3.1 Standalone Experiments

I first perform standalone experiments to evaluate different schedules generated by the autotuner. I compare the following schedules for pipeline parallelism code of Figure 5.6a:

1. **AR-C-P2P** is the implementation of Figure 5.6a, which optimizes over Megatron-LM bt fusing all pointwise computations.

2. **AR-C-P2P-AG** is generated by slicing the output of AllReduce to perform sliced P2P sends and computations, and finally an AllGather to collect the output of computations.

3. **ol(RS,fuse(C-P2P),AG)** is generated from the previous schedule by splitting the AllReduce into a ReduceScatter and an AllGather, reordering the AllGather with P2P send and computations, fusing computations with P2P sends, and finally, overlapping all three communication operations (Figure 5.5b). This schedule is returned by the autotuner as the best schedule and hence, represents CoCoNet in our results.

Results Figure 5.10 shows the breakdown of each operation with one transformer layer assigned to each node. The sequence length ($S = 2048$) and the hidden size ($H = 12288$) are of GPT-3 175B model. CoCoNet's best schedule *ol(RS,fuse(C-P2P),AG)* is 11.75×–12.21× faster than the baseline schedule *AR-C-P2P*. The speedups are because: (i) sliced P2P reduces cross node communication volume, (ii) fusing communication and computation operations improves memory bandwidth utilization, and (iii) overlapping communication using different connections (NVLink within node and InfiniBand across nodes) improves network bandwidth utilization, while other schedules utilize only one stack at a time. Table 5.5 shows that the lines of generated CUDA code for each

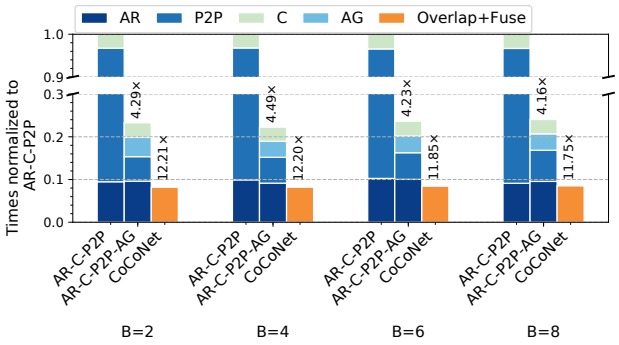


Figure 5.10: Times of three schedules for GPT-3 175B in CoCoNet for pipeline and model parallelism normalized to *AR-C-P2P*.

schedule are significantly more than the implementation in CoCoNet and the autotuner explored all schedules in 11 seconds.

5.5.3.2 Integration with Megatron-LM

I evaluated inference throughput of GPT-3 175B parameter models by integrating CoCoNet's *ol(RS,fuse(C-P2P),AG)* schedule in Megatron-LM. CoCoNet improves inference throughput of GPT-3 by 1.33× due to its fusion and fine-grained overlapping of multiple communication operations.

5.6 Conclusion

This chapter presented CoCoNet, the first language to describe distributed machine learning workloads and optimize them across computation and communication boundary. I show that CoCoNet generated code significantly improves several training and inference times of large language models. CoCoNet is available at `http://github.com/parasailteam/coconet`.

CHAPTER 6

FUTURE OF HARDWARE AND DOMAIN SPECIFIC LANGUAGES

Increasing computation cost of daily machine learning and computer vision applications has lead to the development of cost effective, fast, and energy efficient domain specialized architectures. In 2015 Google released the first generation *Tensor Processing Unit* (TPU) [60], which is a domain specialized architecture designed to improve the training and inference time of machine learning models. Since machine learning models contains mostly matrix multiplication, a TPU contains matrix multiplication units (MXUs) that can perform matrix multiplication on diverse data formats. Today there are several specialized matrix multiplication hardware, such as Cerebras WS-2 [27], Tensor Cores [81] in NVIDIA GPUs, and Matrix Cores [20] in AMD GPUs. All these accelerators provide orders of magnitude Floating Point Operations Per Second (FLOPS) than a general purpose hardware including GPUs. For example, Tensor Cores in NVIDA Tesla V100 GPUs provide 128 Tera FLOPs for 16-bit floating point type, while using CUDA cores in the same GPU provide 28 TFLOPs on 16-bit floats.

However, since these domain specialized accelerators (DSAs) are not general purpose hardware, to use these DSAs for other applications there is a need to develop new algorithms that utilizes matrix multiplications. This chapter discusses state-of-the-art in domain specialized accelerators and how DSLs presented in this thesis can be extended to support these domain specialized accelerators.

6.1 Machine Learning Accelerators

In this section I present a background on machine learning accelerators.

Google Tensor Processing Unit A Tensor Processing Units (TPU) [60] contains Matrix Multiplication Units (MXU) that generates the result matrix of size 128×128. Variable matrix sizes are supported by tiling the matrix multiplication into 128×128 blocks. A TPU support several sizes

including 16-bit and 32-bit IEEE floating points and 8-bit integers. Additionally, a TPU support 16-bit brain floats, which is a datatype for fast machine learning training without loss in accuracy. Brain float provides the same range as 32-bit floats because brain float contains 8-bit of exponent, which is same as the exponent in 32-bit. Since ML models also contains various vector and scalar operations, a TPU also contains vector and scalar processing units with several megabytes of on-chip memory that can be utilized as a programmer defined cache. Third generation TPU provides 123 TFLOPs for 16-bit floating point computations and 4 TFLOPs for 32-bit floating pointer computations.

Cerebras WS-2 Matrix multiplications in a machine learning model are mostly sparse, i.e., elements of weights and intermediate matrices contains zeros. Hence, multiplying only non-zeros elements rather than all elements can give significant performance improvements. Therefore, Cerebras designed Wafer Scale Engine (WSE) [27], which is a specialized architecture to perform sparse matrix multiplications. A WSE-2 wafer contains 850,000 cores and 40 GB of on-chip memory that can serve as a software handled cache.

NVIDIA Tensor Cores Recent NVIDIA GPUs, such as Tesla V100 and Tesla A100, also contains dedicated matrix multiplication hardware known as *Tensor Cores* [81]. Tensor Cores provides both dense and sparse matrix multiplications for a variety of data types including 8-byte integers, 16-bit IEEE float, and 16-bit brain float. Tensor cores provide orders of magnitude higher teraflops than the general purpose CUDA cores. For example, Tensor Cores in Tesla V100 provides 125 TFLOPS on 16-bit floating point, which are significantly higher than 28 TFLOPS provided by CUDA cores. Similarly, AMD GPUs contains dedicated matrix multiplication referred to as *Matrix Cores* [20].

6.1.1 Programming ML Accelerators

ML accelerators are connected to the host CPU through a PCI-e bus. Therefore, a CPU program needs to transfer the input data through the PCI-e bus to the accelerator's DRAM, and obtain the output data stored on accelerator's DRAM after the accelerator has performed computations. Machine learning frameworks follow the same pattern to utilize accelerators.

Figure 6.1 shows the code-generation pipeline of machine learning frameworks when utilizing the accelerators. PyTorch and Tensorflow supports these accelerators while providing the same

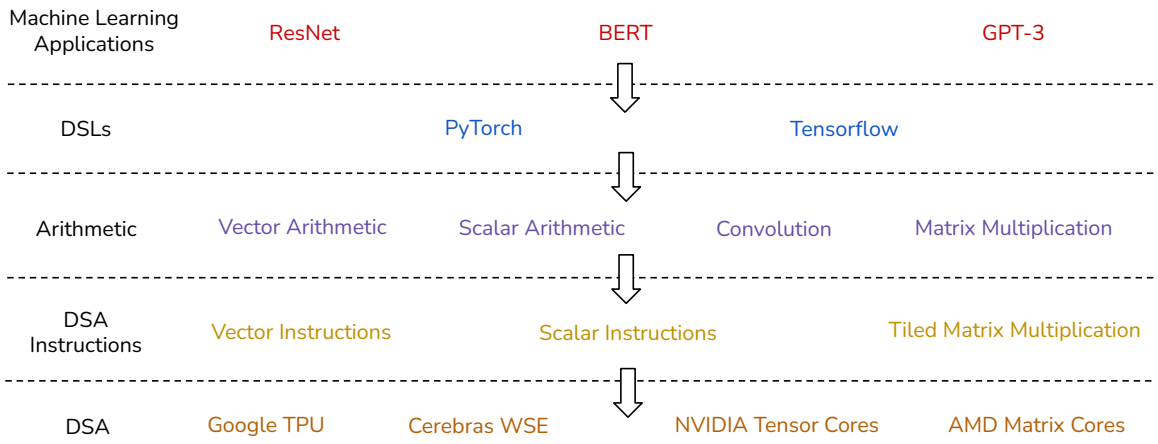


Figure 6.1: Pipeline of ML frameworks to Accelerators

abstraction. After a machine learning model is compiled to matrix multiplications, convolutions, and vector arithmetic operations, both frameworks convert these operators in the instructions supported by the accelerator. For example, vector and matrix multiplication maps directly to vector and matrix multiplication instructions. If an operator is not directly supported by the accelerator, like convolution, the operator is converted to a matrix multiplication with extra vector operations. Finally, the generated instruction are executed by the relevant accelerator.

«««< HEAD

6.2 Domain Specialized Accelerators for Other Tasks

Since a domain specialized accelerators are designed for computing specific task, *is it possible to use a domain specialized accelerator for tasks out of its domain?* In this section, I describe how DSLs presented in this thesis can be adapted for utilizing machine learning accelerators.

The DSLs presented in this thesis (Figure 1.1) has four components: abstraction, domain specific optimizations, architecture specific optimizations, and code generation. Since one of the goals of a DSL is to present a common abstraction across different architectures, the abstraction of a DSL should be same for all architectures. Similarly, the domain specific optimizations that optimizes a program written in high-level abstraction will be applicable irrespective of the target architecture. For example, the transit parallelism in NextDoor is valid for all domain specific accelerators that contains a fast on-chip memory because transit parallelism allows loading edges of common

transit in the fast on-chip memory. Similarly, loop fusion in PolyMage can be applied to image processing programs when executing on domain specialized architectures by storing intermediates in the on-chip memory.

However, the architecture specific optimizations and code generation is specific to the underlying architecture. For instance, NextDoor's load balancing is valid for GPUs and CPUs but not for the domain specialized accelerators because they might have different computation hierarchy. Hence, there is a need to develop new algorithms for executing other tasks using matrix multiplication on domain specialized architectures. For example, since graph processing algorithms like Breadth First Search can be implemented using matrix operations, it is possible to implement graph sampling algorithms using matrix operations. Similarly, image processing programs will need to be implemented as tiled matrix multiplication. After designing the algorithm, the architecture specific code generation will generate architecture specific instruction.

6.3 Conclusion

Domain specialized accelerators for machine learning applications are widely used and are here to stay. These accelerators provide fast implementation of matrix multiplication, which is a key operator in machine learning applications. «««< HEAD With a large availability of these accelerators, there is a need to develop new algorithms to use these accelerators for a variety of tasks. ======= With a large availability of these accelerators, there is a need to develop new algorithms to use these accelerators for a variety of tasks. As an example, this chapter presents a sketch of graph sampling algorithm that uses matrix multiplication. »»»> 5f76c5b (u)

CHAPTER 7

RELATED WORK

This chapter discusses the related work on executing image processing, graph computations, and distributed machine learning programs on multiple GPUs.

7.1 Execution of Stencil Computations on GPUs

State-of-the-art DSLs for image processing programs all employ loop fusion and overlapped tiling to increase locality between stages [75, 90, 92]. Halide and Forma use GPUs and execute one overlapped tile per thread block. Halide's original CPU autoscheduler [76] uses a greedy algorithm, whereas Dynamic Programming Fusion [53] efficiently enumerates all possible fusion choices for a CPU. Halide has a newer autoscheduler [18] that uses beam search with a learned cost model for CPUs. Halide's Gradient GPU autoscheduler [68] is a GPU autoscheduler for Halide that performs greedy function inlining and loop fusion with hard-coded thread block sizes for each tile. In contrast to these PolyMage-GPU's automatic fusion algorithm (Section 3.5) explores all possible fusion choices using dynamic programming and search over all thread block sizes using a cost function that considers several important factors, such as, global memory transactions and occupancy.

Several techniques support parallel execution of stencil computations on GPUs, using the *Overlap tile per thread block (OTPTB)* model [49, 93, 94, 95, 117]. Rawat et al. [94] use a sliding window on one spatial dimension and overlap tiling on the others to eliminate some redundant computations in Overtile [49]. *Hybrid hexagonal classic tiling* [43] also executes one tile per thread block. Flextended Tiles [117] uses rectangle trapezoid tiling to obtain tighter overlapped tile bounds. Artemis [93] is a DSL that allows an expert to guide challenging code optimizations using bottleneck analysis and tunable code parameters. Artemis and Flextended tiles are complementary to our work. These approach supports expression inlining, which pass the value of producer to

consumer through a register within the same thread. However, none of these employ PolyMage-GPU's *overlapped tile per warp (OTPW)* model and *hybrid tiling*, which stores portions of tiles in registers that is shared among threads of a warp.

In 2009, Hong and Kim [50] presented a general analytical model to predict the performance of GPU kernels. However, recent advances in GPU architectures, including changes to their memory hierarchy, have made their model out of date. Prajapati et al. [86] present an analytical model for predicting the runtime of stencil computations on GPUs (tiled using [43]). That model considers shared memory usage, theoretical occupancy, and warp switching. However, it omits several key factors, including register usage, the number of global memory transactions, achieved occupancy, and thread block sizes, which PolyMage-GPU's model considers.

Halide exposes warp shuffle instructions, which makes it possible to store portions of a tile in registers [5]. However, Halide restricts the size of the innermost dimension to be less than warp size, and cannot store tiles in both registers and shared memory. Other systems employ in-register storage and warp shuffles to improve the performance of GPU kernels [19, 24, 35, 66, 85, 109]. PolyMage-GPU allows multiple warps per thread block, allows the innermost dimension to have an arbitrary size, and is a hybrid technique that stores tiles in both registers and shared memory.

7.2 Graph Processing on GPUs and CPUs

There are two types of graph processing systems based on their abstraction: message passing and frontier centric. The first type of graph processing systems provide message-passing abstraction. These systems can run on CPUs [42, 70, 72, 79, 102, 120, 121] and GPUs [41, 61, 80, 97, 119]. Medusa [119] was the first GPU-based graph processing framework to provide a message passing abstraction. CuSha [61] and MapGraph [41] provide a Gather And Scatter (GAS) abstraction. CuSha uses a parallel sliding-window graph representation ("G-Shards") to avoid irregular memory accesses. Subway [97] splits the large graphs that do not fit in GPU memory into sub-graphs and optimizes memory transfers between CPU and GPU. Shi et al [100] present an extensive review of systems for graph processing on GPUs. PowerLyra [30] uses different computations on vertices based on their degree.

The second type of graph processing systems provide frontier-centric abstraction. Gunrock [111] was the first system to provide a frontier-centric abstraction. Gunrock exploits the property that

after any step of a graph computation, a set of *frontier* vertices are produced for the next step of the computation. The Advance operator in this abstraction defines the computation and generates a new frontier by assigning one thread to each neighbor of each vertex in the input frontier. SIMD-X [69] extends the frontier abstraction of Gunrock [111], but these extensions are not relevant for graph sampling. NextDoor performs better than both message passing and frontier centric systems because these systems can only assign transit to different threads but each thread needs to process all samples sequentially.

There are graph processing systems specifically designed for graph sampling and graph sampling. Graph mining systems follow a subgraph-parallel paradigm that is analogous to sample-parallelism [28, 32, 39, 52, 74, 87, 106, 110]. However, even the sample-parallel sampling algorithm of Section 4.3 introduces optimizations that are specific to the graph sampling abstraction of Section 4.1 and do not generalize to graph mining problems. 1) In graph sampling the number of samples is fixed, whereas graph mining problem may involve exploring an exponential number of subgraphs. 2) sampling adds a constant number of new vertices to each sample at each step. This makes it possible to associate new vertices to threads at scheduling time, *before* visiting the graph. 3) Sampling has a notion of *transit* vertices. NextDoor leverages all these features.

There are several CPU based graph sampling systems [33, 34, 47, 113, 116]. KnightKing [113] is a CPU based system to express random walks and efficiently execute them on a distributed system. KnightKing uses rejection sampling to execute random walks. Existing Graph Neural Networks, such as, GraphSAGE [47], ClusterGCN [33], MVS [34], and GraphSAINT [116] performs sampling on input graph using CPUs. In contrast to these systems, NextDoor provides a general abstraction for graph sampling and executes graph sampling efficiently on GPU using its transit parallel paradigm.

7.3 Optimizing Communication and Computation in Distributed Systems

Distributed Machine Learning Abstractions Existing machine learning frameworks [14, 17, 57, 83, 98] and DSLs [31, 37] provide abstractions for writing distributed machine learning workloads. Similar to CoCoNet, in these abstractions, a distributed machine learning program takes input tensors, performs operations on tensors, and returns tensors as the output. However, unlike these

abstractions, CoCoNet preserves the layout information for each tensor. The layout information enables CoCoNet to perform static type checking of each operation, and automatically perform transformations on the program, which is not possible with existing abstractions.

Distributed Neural Network Training Several works have improved data-, model-, and pipeline-parallel techniques for both training and inference. Mesh-Tensorflow [99] and GShard [64] create *shards* of weights and model state that can be split among ranks. Horovod [98] introduced the *Tensor Fusion* optimization that copies all gradients to a single buffer of 64MB, calls AllReduce on the buffer, and then copies the updated value to original gradients. ZeRO [91] splits weights and model state among ranks and uses ReduceScatter and AllGather to distribute computation. FlexFlow [58] performs operator splitting as a way to represent both data-parallelism and model-parallelism, but does not optimize computation with communication. CoCoNet provides several optimizations over these works that are possible only by breaking the abstraction: (i) scattered tensors that remove extra storage and memory copy operations, (ii) fusion communication collectives, and (iii) novel communication and computation overlapping techniques. PyTorch's DDP [67] overlaps AllReduce of gradients with the forward and backward pass. However, unlike CoCoNet, PyTorch's DDP requires extra memory for overlapping, which can increase training time for very large models [12] and do not support slicing of optimizer parameter update that significantly decrease memory usage. GPipe [51], Pipedream [77], and Narayanan et al. [78] proposed pipeline training to improve model parallelism, by dividing the forward and backward pass into several mini-batches, which are then pipelined across devices. vPipe [118] improves these works by providing higher GPU utilization. CoCoNet improves on these works by overlapping inter and intra-node communication operations. BytePS [59] utilizes CPU in heterogenous clusters to improve training, which is complementary to CoCoNet.

Overlapping Computation and Communication State-of-the-art works on overlapping [21, 63, 71, 73, 104] use either pipelined execution to overlap communication and computation or non-blocking MPI operations. Pencil [108] improves upon these works by performing pipelining within a process and supports computations in multiple connected iteration spaces. Several techniques distribute tiles and automatically generate communication [25, 37, 96]. Basu et. al. [23] uses overlapped tiling in each process to remove communication between processes. Denis and Trahay [36]

studied the efficiency of overlap. dCUDA [45] provides hardware supported overlap. These works for MPI+OpenMP are valid for CPU based stencil computations that require sends and receives to share the halo regions. However, unlike CoCoNet, these works do not support overlapping between collectives communication and complex computations like convolutions and matrix multiplications. CoCoNet supports overlapping multiple computation and communication operations on GPUs without an accelerator.

BIBLIOGRAPHY

[1] Warp Shuffle Functions in AMD HIP. URL https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_kernel_language.md#warp-shuffle-functions.

[2] CUDA C Programming Guide, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[3] Using CUDA Warp-Level Primitives, . https://devblogs.nvidia.com/using-cuda-warp-level-primitives/.

[4] Halide, . https://github.com/halide/Halide/
commit 52da814a2c3c4af78125757385a8a86efdde3234.

[5] Halide, . https://github.com/halide/Halide/
commit 59bca3c8e535f7f99c90efd1d932db934f9c01b6.

[6] NVIDIA Apex. https://github.com/NVIDIA/apex, Accessed: 2022-01-12.

[7] cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html, Accessed: 2022-01-12.

[8] cuDNN. https://docs.nvidia.com/cuda/cudnn/index.html, Accessed: 2022-01-12.

[9] CUTLASS. https://github.com/NVIDIA/cutlass, Accessed: 2022-01-12.

[10] OpenAI's GPT-3 Language Model: A Technical Overview. https://lambdalabs.com/blog/demystifying-gpt-3/, Accessed: 2022-01-12.

[11] GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, Accessed: 2022-01-12.

[12] NVIDIA Megatron-LM. https://github.com/NVIDIA/Megatron-LM/, Accessed: 2022-01-12.

[13] Training with Mixed Precision. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html, Accessed: 2022-01-12.

[14] Apache mxnet. https://mxnet.apache.org/, Accessed: 2022-01-12.

[15] NVIDIA Collective Communication Library. https://github.com/NVIDIA/nccl, Accessed: 2022-01-12.

[16] NVIDIA BERT. https://github.com/NVIDIA/DeepLearningExamples, Accessed: 2022-01-12.

[17] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[18] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michael Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Fredo Durand, and Jonathan Ragan-Kelley. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.*, 2019.

[19] Karan Aggarwal and Uday Bondhugula. Optimizing the Linear Fascicle Evaluation Algorithm for Many-core Systems. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, 2019.

[20] AMD. AMD Matrix Cores. `https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf`. Accessed June 24 2022.

[21] Youcef Barigou and Edgar Gabriel. Maximizing Communication–Computation Overlap Through Automatic Parallelization and Run-time Tuning of Non-blocking Collective Operations. *International Journal of Parallel Programming*, 45, 2017. doi: 10.1007/s10766-016-0477-7.

[22] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, 2008.

[23] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *20th Annual International Conference on High Performance Computing*, 2013. doi: 10.1109/HiPC.2013.6799131.

[24] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, 2016.

[25] Uday Bondhugula. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013. doi: 10.1145/2503210.2503289.

[26] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, 2020.

[27] Cerebras. Cerebras WS-2. https://www.cerebras.net/blog/an-ai-chip-with-unprecedented-performance-to-do-the-unimaginable/. Accessed June 24 2022.

[28] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. Large Scale Graph Mining with G-Miner. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[29] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*, ICLR'18, 2018.

[30] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.*, 2019.

[31] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018.

[32] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.*, 2020.

[33] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, 2019.

[34] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. Minimal Variance Sampling with Provable Guarantees for Fast Training of Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, 2020.

[35] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, 2019.

[36] A. Denis and F. Trahay. MPI Overlap: Benchmark and Analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*, 2016. doi: 10.1109/ICPP.2016.37.

[37] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed Halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016. doi: 10.1145/2851141.2851157.

[38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423.

[39] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[40] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[41] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, 2014.

[42] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[43] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, 2014.

[44] Aditya Grover and Jure Leskovec. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, 2016.

[45] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. dCUDA: Hardware Supported Overlap of Computation and Communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016. doi: 10.1109/SC.2016.51.

[46] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020. doi: 10.1145/3410463.3414632.

[47] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 2017.

[48] Taher H. Haveliwala. Topic-Sensitive PageRank. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, 2002.

[49] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, 2012.

[50] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[51] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32*. 2019.

[52] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[53] Abhinav Jangda and Uday Bondhugula. An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, 2018.

[54] Abhinav Jangda and Arjun Guha. Model-Based Warp Overlapped Tiling for Image Processing Programs on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, 2020.

[55] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating Graph Sampling for Graph Machine Learning using Graphics Processing Units. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 311–326, 2021.

[56] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, 2022. doi: 10.1145/3503222.3507778.

[57] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[58] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems*, 2019.

[59] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[60] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):67–78, jun 2020. ISSN 0001-0782. doi: 10.1145/3360307. URL `https://doi.org/10.1145/3360307`.

[61] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, 2014.

[62] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[63] N. Koziris, A. Sotiropoulos, and G. Goumas. A pipelined schedule to minimize completion time for loop tiling with computation and communication overlapping. *Journal of Parallel and Distributed Computing*, 63(11), 2003.

[64] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations*, 2021.

[65] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, 2014.

[66] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. Warp-Consolidation: A Novel Execution Model for GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, 2018.

[67] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 2020.

[68] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.*, 2018.

[69] Hang Liu and H. Howie Huang. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, 2019.

[70] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, 2010.

[71] H. Lu, S. Seo, and P. Balaji. MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, 2015. doi: 10.1109/HPCC-CSS-ICESS.2015.82.

[72] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[73] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010. doi: 10.1145/1810085.1810091.

[74] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[75] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[76] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.*, 2016.

[77] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019. doi: 10.1145/3341301.3359646.

[78] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[79] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[80] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.

[81] NVIDIA. NVIDIA Tensor Cores. `https://www.nvidia.com/en-us/data-center/tensor-cores/` Accessed June 24 2022, 2022.

[82] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[83] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 2019.

[84] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, 2014.

[85] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

[86] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, 2017.

[87] Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 2016.

[88] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[89] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[90] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[91] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[92] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, 2015.

[93] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L. Pouchet, and P. Sadayappan. On Optimizing Complex Stencils on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[94] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, 2016.

[95] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-No"el Pouchet, and P Sadayappan. On Optimizing Complex Stencils on GPUs. 2019.

[96] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[97] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing Data Transfer during out-of-GPU-Memory Graph Processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[98] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow, 2018.

[99] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems*, 2018.

[100] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)*, 2018.

[101] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020.

[102] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.*, 2013.

[103] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP, 2019.

[104] Hari Subramoni, Sourav Chakraborty, and Dhabaleswar K. Panda. Designing Dynamic and Adaptive MPI Point-to-Point Communication Protocols for Efficient Overlap of Computation and Communication. In *High Performance Computing*, 2017.

[105] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, 2017.

[106] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[107] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[108] Hengjie Wang and Aparna Chandramowlishwaran. Pencil: A Pipelined Algorithm for Distributed Stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[109] J. Wang, X. Xie, and J. Cong. Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[110] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[111] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. *SIGPLAN Not.*, 2016.

[112] Michael Wolfe. The Definition of Dependence Distance. *ACM Trans. Program. Lang. Syst.*, 1994.

[113] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. KnightKing: A Fast Distributed Graph Random Walk Engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ;19, 2019.

[114] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, 2018.

[115] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *International Conference on Learning Representations*, 2020.

[116] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*, ICLR '20, 2020.

[117] Jie Zhao and Albert Cohen. Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation. *ACM Trans. Archit. Code Optim.*, 2019.

[118] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. vPipe: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[119] J. Zhong and B. He. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[120] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[121] Youwei Zhuo, Jingji Chen, Qinyi Luo, Yanzhi Wang, Hailong Yang, Depei Qian, and Xuehai Qian. SympleGraph: Distributed Graph Processing with Precise Loop-Carried Dependency Guarantee. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 2020.

[122] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. In *Advances in Neural Information Processing Systems*, Nuerips '19, 2019.