FACILITATING EMERGING APPLICATIONS ON MANY-CORE PROCESSORS

A Dissertation

Presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

DA LI

Dr. Michela Becchi, Supervisor

JULY 2016

The undersigned, appointed by the dean of the Graduate School, have examined the dissertation entitled

FACILITATING EMERGING APPLICATIONS ON MANY-CORE PROCESSORS

presented by Da Li,

a candidate for the degree of doctor of philosophy

and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Michela Becchi

Dr. Tony Han

Dr. Zhihai He

Dr. William Harrison

Dr. Jason Xu

ACKNOWLEDGEMENTS

First and foremost, I would like to deeply thank my adviser, Dr. Michela Becchi, for her valuable guidance and advice and for her vast reserve of patience and knowledge. I am fortunate and grateful to have worked with her during my studies. I also appreciate her generosity and flexibility in offering me opportunities to interact with other scientists and explore various projects through internships from industries.

I would like also to express my sincere gratitude to Dr. Tony Han, Dr. Zhihai He, Dr. William Harrison and Dr. Jason Xu for their teaching, mentoring and assistance. I want to thank Dr. Ziliang Zong and Xinbo from Texas State University for their expertise, feedback and overall support. I would like to thank all the current and previous members of the Networking and Parallel System laboratory, especially Kittisak, Huan, Ruidong and Henry who helped me with my research.

Finally, I would like to acknowledge my family and friends for their encouragement and devotion. I am very grateful to have a supportive family who puts my education as the first priority. I would express my deepest appreciation and thanks to my wife, Xin Tong, who is better at science than me, for her love and support.

This work has been supported by National Science Foundation awards CNS-1216756, CCF-1452454, CNS-1305359, and by gifts and equipment donations from NEC Labs America and Nvidia Corporation.

Table of Contents

ACKNOWLEDGEMENTS ii

LIST OF FIGURES vii

LIST OF TABLES xi

ABSTRACT xii

Chapter 1 Introduction 1

1.1 Introduction 1

1.2 Contributions 4

1.3 Dissertation Organization 5

Chapter 2 Background 6

2.1 GPU Architecture and Programming Model 6

2.2 Needleman-Wunsch Algorithm 9

2.3 Irregular applications 11

2.4 Convolutional neural networks 12

Chapter 3 Bioinformatics Applications on CPU-GPU Clusters 15

3.1 Motivation 15

3.2 Related Work 16

3.2.1 Early Works on Sequence Alignments 16

3.2.2 Acceleration of Bioinformatics Application on GPUs 17

3.2.3 Sequence Alignments on GPUs 18
3.3.4 Rodinia-NW: Needleman-Wunsch on GPU 19
3.3 Design of GPU-workers 21
3.3.1 TiledDScan-mNW: Multiple alignments with tiling 21
3.3.2 DScan-mNW: Single-kernel diagonal scan 22
3.3.3 RScan-mNW: Row scan via single CUDA core 24
3.4 Experimental Evaluation 28
3.4.1 Experimental setup 28
3.4.2 Performance on single GPU 28
3.5 Other Applications with Similar Computation Pattern 35
Chapter 4 Irregular Applications on Many-Core Processors 40
4.1 Related Work 41
4.2 Unified Programming Interface 44
4.2.1 General Design & Methodology 46
4.2.2 Programming API 47
4.2.3 Case Study 50
4.2.4 Compiler Design 54
4.2.5 Runtime Library Design 58
4.2.6 Performance Evaluation 61
4.3 Adaptive Computing 68

4.3.1 Motivation 69
4.3.2 Exploration Space 74
4.3.3 Implementation 81
4.3.4 Adaptive Runtime 86
4.3.5 Experimental Evaluation 91
4.4 Parallelization Templates 99
4.4.1 Motivation 100
4.4.2 Irregular Nested Loop 101
4.4.3 Experimental Evaluation 105
4.5 Workload Consolidation 113
4.5.1 Dynamic Parallelism 114
4.5.2 Application Characterization 115
4.5.3 Motivation 116
4.5.4 Methodology 121
4.6.4 Experimental Evaluation 132
Chapter 5 Deep Neural Network on CPU-GPU 141
5.1 Related Work 141
5.2 Energy Efficiency 142
5.2.1 Motivation 142
5.2.2 Methodology 144

5.2.3 Overall Results on CPU & GPU 146
5.2.4 Effect of NN and Batch Size Configuration 151
5.2.5 Effect of Hardware Settings 153
5.3 CNN on CPU-GPU heterogeneous architecture 158
5.3.1 Motivation 158
5.3.2 HetNet 159
5.3.3 HybNet 161
5.3.4 Experimental Evaluation 162
5.4 Virtual Memory for CNN on GPU 167
5.4.1 Motivation 167
5.4.2 Design 168
5.4.3 Experimental Evaluation 169
Chapter 6 Conclusion 173
References 176
VITA 191

LIST OF FIGURES

Figure 1: Fermi GPU architecture and the SM 7

Figure 2: Kepler GPU architecture and SMX v.s. Fermi's SM 8

Figure 3: Calculation of matrix in NW 10

Figure 4: Example of Deep Convolutional Neural Network (LeNet) 13

Figure 5: TiledDScan-mNW and the mapping to GPU cores and SMs 22

Figure 6: DScan-mNW and the mapping to GPU cores and SMs 23

Figure 7: RScan-mNW and of the mapping to GPU cores and SMs 24

Figure 8: Evaluation of memory optimizations on Rodinia-NW 30

Figure 9: TiledDScan-mNW Kernel speedup over Rodinia-NW 31

Figure 10: DScan-mNW Kernel speedup over Rodinia-NW 32

Figure 11: Kernel speedup of RScan-mNW on sequences of various lengths 33

Figure 12: Speedup of DScan-mNW over an 8-threaded CPU implementation 34

Figure 13: Tiled horizontal-vertical scan (CW-TiledHV) 36

Figure 14: Data flow of CW-TiledHV implementation 37

Figure 15: WF-Tiled implementation 38

Figure 16: Proposed graph processing system 45

Figure 17: Graph programming API 48

Figure 18: A-DFA compression algorithm 51

Figure 19: PageRank algorithm 52

Figure 20: DFA (subset) construction algorithm 53

Figure 21: Hierarchical-CSR with level-2, level-3 nodes/edges arrays 55

Figure 22: A-DFA compression –Speedup of GPU over serial CPU implementation62

Figure 23: A-DFA compression – Speedup of Xeon Phi over serial CPU implementation. 63

Figure 24: PageRank on dynamic graphs with hierarchical CSR.............................. 64

Figure 25: DFA construction – Speedup of multi-core CPU, GPU and Intel Phi over serial CPU code.. 65

Figure 26: DFA construction using different allocators and GPUs........................... 66

Figure 27: Outdegree distributions of *CO-road*, *Amazon* and *CiteSeer* networks 71

Figure 28: Unordered SSSP – size of the working set during the execution (*CO-road*, *Amazon* and *CiteSeer* networks) 72

Figure 29: Exploration space 75

Figure 30: Ordered and unordered BFS algorithms... 76

Figure 31: Ordered and unordered SSSP algorithms... 77

Figure 32: Working set: (a) bitmap vs. (b) queue.. 80

Figure 33: Compressed sparse row graph representation 82

Figure 34: Generic CPU pseudo-code for BFS and SSSP... 83

Figure 34: Pseudo-code of kernel functions (computation and workset_gen) 84

Figure 36: Overview of our adaptive framework 87

Figure 37: Design space.. 91

Figure 38: Processing speed of bet implementation 94

Figure 39: Performance under different T_3 settings (SSSP) 96

Figure 40: Performance under different sampling rates (SSSP)................................ 97

Figure 41: Performance of our adaptive runtime on BFS (to the left) and SSSP (to the right) – baseline: best static solution ... 98

Figure 42: Parallelization templates for irregular nested loops ... 102

Figure 43: *SpMV*: Speedup of load balancing code variants over basic thread-mapped implementation under different settings ... 108

Figure 44: *SSSP*: Speedup of load balancing code variants over basic thread-mapped implementation ... 109

Figure 45: Speedup of load balancing code variants over basic thread-mapped implementation under different lb_{THRES} settings ... 111

Figure 46: Basic-dp code template and sample codes ... 118

Figure 47: Workload consolidation – illustration ... 122

Figure 48: Kernel transformation flow ... 124

Figure 49: Example of use of our workload consolidation compiler directive... 129

Figure 50: Performance of different buffer implementations (SSSP) ... 134

Figure 51: Performance of different kernel configurations (TD) ... 135

Figure 52: Overall speedup over basic dynamic parallelism ... 137

Figure 53: Warp execution efficiency across benchmarks ... 138

Figure 54: SMX occupancy (achieved hardware utilization) ... 139

Figure 55: DRAM transactions ratio over basic dynamic parallelism... 140

Figure 56: Comparison between native GPU implementation and cuDNN v3 library in Caffe... 146

Figure 57: Comparison among different frameworks on K20m and TitanX GPUs. 149

Figure 58: Comparison among CNN frameworks on CPUs... 150

Figure 59: Breakdown of energy consumption of AlexNet and OverFeat on K20 and CPU using Caffe 151

Figure 60: Power and energy consumption on K20m, Titan X and CPU with different batch sizes using Caffe........ 153

Figure 61: Experimental results with different Hyper-Threading settings 154

Figure 62: Power and energy consumption using different memory and core frequencies 157

Figure 63: An illustration of HetNet........ 160

Figure 64: An illustration of HybNet........ 161

Figure 65: Performance of HetNet........ 163

Figure 66: Performance of HybNet with different neural networks 164

Figure 67: GPU memory footprint of HybNet with different neural networks 165

Figure 68: Performance of HybNet with AlexNet under different batch sizes........ 166

Figure 69: GPU memory footprint of HybNet with AlexNet under different batch sizes........ 167

Figure 70: Performance of vDNN with 1G GPU memory 170

Figure 71: Performance of vDNN with full GPU global memory........ 171

LIST OF TABLES

Table 1: Characteristics of the GPUs used in our evaluation 29

Table 2: Summary of applications and speedup over serial code 68

Table 3: Dataset characterization. 70

Table 4: Speedup of BFS (GPU implementation over serial CPU baseline)............. 92

Table 5: Speedup of SSSP (GPU code over serial CPU baseline) 93

Table 6: Profiling data collected on SSSP (lb_{THRES}=32) 111

Table 7: Warp execution efficiency (*dbuf-shared*) 112

Table 8: Clauses of our workload consolidation compiler directive 127

Table 9: Memory and core frequencies supported on K20m GPU.......................... 156

ABSTRACT

Over the last decade, many-core Graphics Processing Units (GPUs) have been widely used to accelerate a variety of applications. Meanwhile, Intel has released its Xeon Phi Coprocessor, which is equipped with more than fifty x86 cores, each supporting four hardware threads. Despite their widespread use, many-core processors are still considered relatively difficult to program, in that they require the programmer to be familiar with both parallel programming and the hardware features of these devices.

Due to their massive computational powers, many-core processors have been successfully used to parallelize a wide variety of dense matrices and vectors based applications. These extensively investigated problems are mainly from linear algebra, stencil computations, image processing and so on. However, many established and emerging problems have not yet been fully explored. Some of these applications use irregular algorithms/operations (e.g., dynamic programming), while others are based on irregular data structures, such as graphs. It has been shown that these emerging applications do exhibit certain degree of static and runtime parallelism, but are relatively hard to parallelize.

My research focuses on addressing important issues related to the deployment of emerging applications on many-core processors. In particular, we proposed efficient GPU implementations for large-scale pairwise sequence alignment and integrated proposed GPU kernels into a hybrid MPI-CUDA framework for CPU-GPU clusters. we also targeted graph- or tree-based applications and proposed: (1) unifying programming interfaces for many-core processors (2) runtime support for efficient execution on

irregular datasets and (3) compiler support for efficient mapping of applications onto hardware. Finally, we conducted a comprehensive study of performance, memory footprint and power consumption on various platforms and extended existing central processing units (CPU) only or graphic processing units (GPU) only CNNs learning methods to CPU-GPU cooperative ways. We also implemented a virtual memory and integrated into Caffe to facilitate training large CNN models with limited GPU memory.

Chapter 1 Introduction

1.1 Introduction

Over the last decade, many-core graphics processing units (GPUs) have been widely used to accelerate a variety of applications. Meanwhile, Intel has released its new Intel® Xeon Phi™ coprocessors, which are equipped with more than fifty x86 cores, each supporting four hardware threads. The peak double-precision performance of high-end many-core devices from Nvidia, AMD and Intel are well above 1 teraflops. Due to heterogeneity in both hardware features and application characteristics, it is often quite hard to choose the hardware platform that can guarantee good performance to a given application. In addition, performance optimization has increasingly become more hardware and application specific and an optimization designed for a hardware platform might not work at all on others.

Besides the increased complexity on the hardware side, the many-core era has also lent to significant software challenges. Although many-core processors are commonly used, they are still relatively difficult to program since they require programmers to be familiar with both parallel programming and with the features and operation of these hardware platforms [1]. To achieve good performance, the programmers need to tune their code and sometimes even redesign their algorithms to better fit the underlying hardware. This is not at all an easy job. This complexity is aggravated by the variety of software stacks used by the various many-core platforms. For instance, Nvidia's GPUs adopt both CUDA and OpenCL as their programming interface. However, in order to program the Intel Xeon Phi, one needs to master its customized OpenMP directives as well as other programming tools like Intel TBB and Cilk. This increasing variety of

programming models does not make the use of many-core processors any easier, not to mention that it also requires the programmer to become familiar with the debugging and profiling tools associated with each programming interface.

Due to their massive computational powers, many-core processors have been successfully used to parallelize a wide variety of dense matrix- and vector-based applications. These extensively investigated problems come mainly from linear algebra, stencil computations, image processing, and other applications with regular computational and memory access patterns. However, many established and emerging problems have not yet been fully explored. Some of these applications use less regular algorithms/operations (e.g., dynamic programming), while others are based on irregular data structures, such as graphs. Examples can be drawn from different application domains such as bioinformatics, social networking, machine learning, electrical circuit modeling, discrete event simulation, compilers, and computational sciences. It has been shown that these emerging applications do exhibit a certain degree of static and runtime parallelism, but are relatively hard to parallelize.

Among emerging applications, three categories of applications are eye-catching: bioinformatics, graph processing and deep neural network based applications.

With the development of fast and cheap genome sequencing techniques, bioinformatics plays a more and more important role in biology. From sequence alignment to genome editing, from structural biology to personalized health, scientists are using modern technology to produce a huge amount of data and powerful machines to analyze these data. New algorithms to accelerate scientific discoveries are in continuous development. A popular example is the Needleman-Wunsch algorithm, which is a widely

used global sequence alignment tool with applications in the analytics of complex microbial communities and the inference of the tree of life, to name a couple of use cases. The goal of this algorithm is to find the alignment of two strings (generally protein or DNA) that maximizes a cost function.

Graph applications are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence, and a degree of parallelism that is known only at runtime (rather than at compile time). In fact, the amount of parallelism within graph and other irregular applications depends on the characteristics of the dataset, rather than solely on its size. Yet, many established and emerging applications are irregular in nature, being based on irregular data structures, such as graphs and trees. Graph and trees are powerful representations used in many practical applications. Examples of such applications include adaptive meshes, web search, networking, online marketing and social network analysis. With the increased popularity of social and web network analysis, there is an increasing demand for accelerating these applications.

Neural network is nowadays an important machine learning method. The history of neural network research can be traced back to the second half of the last century and neural networks were successfully applied to recognize handwritten checks and ZIP codes in mail in the 90s. Before being used in classification, neural networks need to be trained. Although the high computational requirements of the training phase continue to be a key factor hindering the advancement of algorithms and applications based on neural networks, recent advancements in software and hardware have dramatically promoted their use in both academia and industry. Nowadays, neural network is a driving force for computer vision, natural language processing and speech recognition. Based on recent

breakthroughs in these fields, many exciting applications and technologies like autonomous driving are ready to change our world.

As we can see, these emerging applications are rapidly evolving and computational power is the key to this revolution. Although hardware and software complexities in the many-core era create a number of challenges for us to address, they also bring many opportunities for us to explore.

1.2 Contributions

In this dissertation, we explore the design of effective software systems for many-core platforms and make the following contributions:

- In the context of application-specific acceleration, we explore different GPU implementations of the Needleman-Wunsch bioinformatics algorithm. Many optimization methods discussed in this dissertation can be extended to algorithms that have similar computation and memory access patterns and are used in other domains (e.g., *integral histogram* used in computer vision and pattern recognition).
- In the context of graph processing, we demonstrate that the design of easy-to-use programming models and effective compiler and runtime techniques can hide the hardware details from programmers and dramatically simplify the use of many-core processors without sacrificing performance. We also explore a new hardware feature for GPUs – called *dynamic parallelism* – suitable for graph processing and other applications with irregular computation and memory access patterns. We propose a novel method to reduce runtime overhead and improve hardware

resource utilization when using this feature. The techniques and insights discussed in this dissertation are mostly transferable to many other irregular applications.

- We conduct a comprehensive study on the power behavior and energy efficiency of neural networks on CPUs and GPUs. We propose novel hybrid CPU-GPU solutions to reduce memory footprint and improve resource utilization, as well as overall performance, of neural network computations. Our insights can facilitate the design of high performance and energy efficient software solutions for neural networks.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides some background on the GPU architecture and the three types of applications covered in this dissertation. Chapter 3 describes different implementations of the Needleman-Wunsch algorithm on GPU. Chapter 4 describes our work on the deployment of graph and other irregular computations on many-core processors. Specifically, this chapter includes (1) unifying programming interfaces for many-core processors (Chapter 4.2), (2) runtime support for efficient execution of applications on irregular datasets (Chapter 4.3) and (3) compiler support for the efficient mapping of irregular applications onto parallel hardware (Chapter 4.4). Chapter 5 describes our workload characterization and improvement on training deep convolutional neural networks. Chapter 6 concludes our discussion.

Chapter 2 Background

2.1 GPU Architecture and Programming Model

Nvidia GPUs have evolved for four generations: the pre-Fermi, the *Fermi*, the *Kepler* and *Maxwell*. In pre-Fermi and Fermi architectures, the GPUs comprise a set of Streaming Multiprocessors (SMs) and GPUs with different compute capabilities are distinguished by the numbers of SMs. Each SM contains a set of simple in-order cores. These in-order cores execute instructions in a SIMD manner. Figure 1 shows the internal micro-architecture of one SM in Fermi GPU. The "cores" are actually different SIMD lanes. Multiple lanes (8 or 16) shared one set of instruction fetch unit (e.g. scheduler and dispatcher). Usually, a SM has multiple instruction fetch units (2 or 4) and the number of cores varies from 16 to 48. Starting from Kepler, the cores in Streaming Multiprocessors have increased dramatically and are named after SMX. Figure 2 shows the comparisons between SM of Fermi architecture and SMX of Kepler architecture. The new Kepler architecture is design for high throughput as well as power efficiency. It slows down the clock speed of SMX but integrates much more CUDA cores (192 v.s. 32/48), which achieve 2x performance per watt. With more cores, both the bandwidth and capacity of the register file in each SMX are doubled.

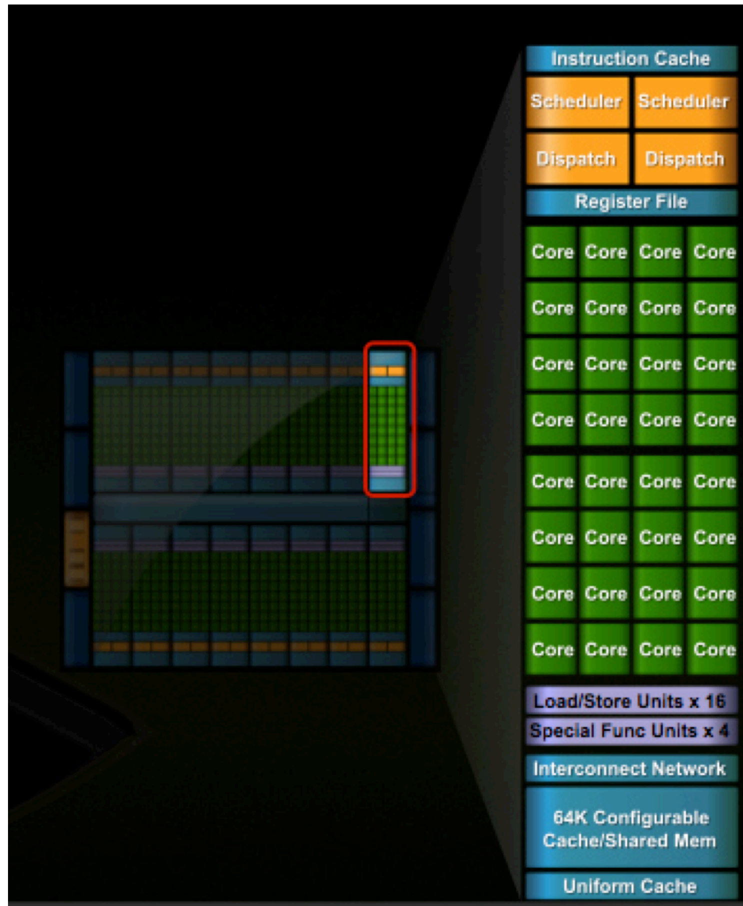


Figure 1: Fermi GPU architecture and the SM

GPUs have a heterogeneous memory organization consisting of high latency off-chip global memory, low latency read-only constant memory (which resides off-chip but is cached), low-latency on-chip read-write shared memory, and texture memory. GPUs adopting the Fermi and Kepler architecture, such as those used in this work, are also equipped with a two-level cache hierarchy. Judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieve good performances. In particular, the utilization of the memory bandwidth can be optimized by performing regular access patterns to global memory. In this situation, distinct memory accesses are automatically coalesced into a single memory transaction, thus limiting the memory bandwidth used.

The GPU can support thousands of parallel threads and the overhead of context switch is pretty low. Thus, unlike CPU, which utilize large cache to achieve high performance, GPU rely on massive parallel thread and fast context switch to high the long memory access latency. This design makes GPU have higher density of ALU than CPU and become the rising star in high performance computing.

However, programming massive parallel threads is hard, especially for GPUs. The advent of CUDA has greatly increased the programmability of GPUs. With CUDA, the programmers are required to write the kernel functions, which are executed on the GPUS and the computation is organized in a hierarchical fashion, wherein threads are grouped

Figure 2: Kepler GPU architecture and SMX v.s. Fermi's SM

into thread-blocks. The CUDA provides several built-in thread identifiers and block identifiers and assign them to different threads. Each thread-block is mapped onto a different SM(X), whereas different threads are mapped to simple cores and executed in SIMD units, called warps. The presence of control-flow divergence within warps can decrease the GPU utilization and badly affect the performance. Threads within the same block can communicate using shared memory, whereas threads within different thread-blocks are fully independent. Therefore, CUDA exposes to the programmer two degrees of parallelism: fine-grained parallelism within a thread-block and coarse-grained parallelism across multiple thread-blocks.

2.2 Needleman-Wunsch Algorithm

The goal of the Needleman-Wunsch algorithm (NW) is to find the alignment of two strings (generally protein or DNA) that maximizes a cost function. That cost function consists of two parts. The first is a match and mismatch scoring matrix that gives the cost of aligning matching or mismatching sequence elements (hereafter $S(x_i,y_j)$). For DNA alignments, simple schemes such as rewarding matches (+4) and penalizing mismatches (-5) are often used. For protein alignments, it is more common to use an empirical scoring matrix [e.g., BLOSUM; 2]. The second part of the function is a cost for “gaps”: i.e., regions of one sequence not aligned against regions of the other. Here, we will apply a linear gap cost G. As input data, NW takes two sequences of length m and n. The optimal alignment is then computed within a 2-D matrix M of size $(m+1)*(n+1)$. Note that this matrix can be virtual: there are linear space memory implementations of the

algorithm (e.g. Hirschberg's algorithm [3]). Each element in M is then computed according to equation (1).

$$M(i,j) = \max \begin{cases} M(i-1, j-1) + S(x_i, y_j) \\ M(i-1, j) + G \\ M(i, j-1) + G \end{cases} \quad (1)$$

Here, *M(i,j)* is the alignment score in the i_{th} row and j_{th} column of *M*. The first row and column of M are initialized as gaps of increasing length [4]: once this initialization is complete, the remaining positions can be computed given the values above them, to their left and to their left diagonal (Figure 1).

It is apparent from this description that the memory and computing requirements of a naïve implementation of the algorithm can be significant, as they scale as *O(mn)* (often spoken of as $O(n^2)$). For instance, in our experiments, we use database of roughly 25,000 unique 16S rDNA genes from the Ribosomal Database Project [5]. Performing all possible pairwise alignments involves roughly 300 million comparisons. Moreover, the computation itself is somewhat memory intensive: as equation (1) indicates, computing each new element in the alignment matrix requires three reads from memory and one write to store the new value. On the other hand, the computation is relatively trivial, requiring three additions and a comparison.

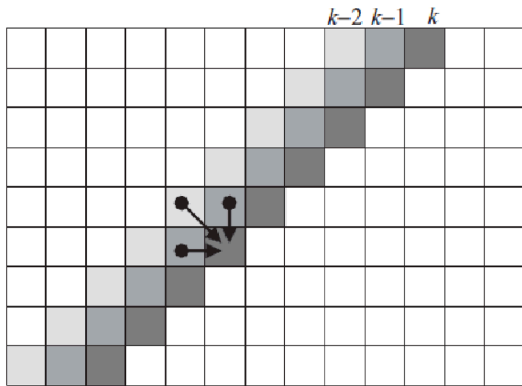


Figure 3: Calculation of matrix in NW

The NW algorithm can be broken into two phases: (1) the computation of the alignment matrix M (described above, Figure 3), and (2) the trace-back operation, which uses the alignment matrix to reconstruct the

sequence alignment itself. Unless linear-space implementations of NW [3] are adopted, the trace-back is a linear-time operation accounting for a small fraction of the overall execution time. In chapter 4, the GPU-works mainly focus on the computation of the alignment matrix.

2.3 Irregular applications

If the effective deployment of regular applications on many-core processors has been extensively investigated, the one of irregular applications is still far from understood. Irregular applications are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence, and dynamic parallelism at runtime (rather than static parallelism at compile time). Different from regular applications, whose parallelism is determined solely by the size of dataset, the amount of parallelism within irregular applications depends on the characteristics of the dataset.

Graphs, as a powerful representation used in many practical applications (e.g. networking, social networking, online marketing, webpage search, citation networks, among others), are intrinsically irregular. In the past decades, the size of real world datasets has rapidly increased, thus exposing higher amount of parallelism for many graph algorithms. It has been shown that graphs used in real-world applications exhibit significant topological differences. The topology of the graphs dictates the amount of parallelism that can be extracted at runtime, thus affecting the performance of specific GPU implementations. This heterogeneity makes it difficult to design a GPU implementation of a graph algorithm that is optimal on a large variety of datasets.

From the point view of computational patterns, irregular loops and parallel recursion are two important categories of irregular applications. Irregular loops are characterized by

an uneven work distribution across loops iterations. For example, nested loops where the number of iterations of inner loops varies across the iterations of outer loops. The degree of parallelism within irregular loops is typically data dependent and known only at runtime. Parallel recursion is also considered as an important form of irregular applications because recursive calls may spawn different amount of parallel work. As a consequence of this uneven work distribution, simple parallelization code handling all the recursive calls in the same way may lead to hardware underutilization.

2.4 Convolutional neural networks

At a high level, convolutional neural networks simulate the way in which human brains process and recognize images. They belong to the family of multi-layer perceptrons (MLP) [6]. A MLP is a multi-layer neural network consisting of an input layer, an output layer and multiple hidden layers between the input and output layers. Each hidden layer represents a function between its inputs and outputs that is defined by the layer's parameters.

Convolutional neural networks mainly consist of three types of layers: *convolutional layers* (Conv), *pooling layers* (Pooling) and *fully connected layers* (FC). Each layer may contain thousands to millions of neurons. A single neuron takes some inputs, computes their weighted sum, and sends the output to the neurons in the next layer. In this way, distinct layers apply different operations to their inputs and produce outputs for the layers that follow. Figure 4 shows an example of convolutional neural network (LetNet [7]), which consists of alternated convolutional and pooling layers followed by a few fully-connected layers.

Convolutional layers: These layers apply convolutions to the input with several filters and add a bias term to the results. Very often, a nonlinear function (called *activation function*) is also applied to the results. Convolutional layers exploit spatial connectivity and shared weights. The parameters of a convolutional layer are reduced dramatically compared to a typical hidden layer of a MLP. Convolutional layers are the most computational intensive layers in CNNs.

Pooling layers: These layers perform a nonlinear down-sampling operation on the input. They partition the input into a set of sub-regions and output sampled results from these sub-regions. Based on their sampling method, pooling layers can be categorized into: *maximum pooling*, *average pooling* and *stochastic pooling*. Pooling layers progressively reduce the amount of parameters as well as control model over-fitting. Pooling layers are usually placed between two convolutional layers.

Fully connected layers: Unlike in convolutional layers, neurons in FC layers have full connections to all output from the preceding layers. As a consequence, a FC layer has many more parameters than a convolutional layer. Nonetheless, since convolution operations are replaced by multiplications, fully connected layers require less computational power.

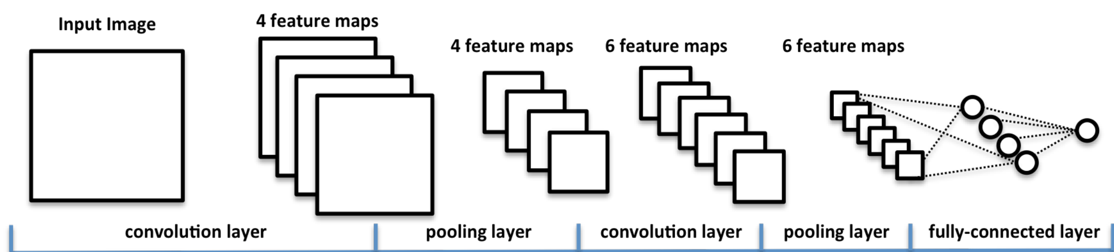


Figure 4: Example of Deep Convolutional Neural Network (LeNet)

Using CNNs for machine learning tasks involves three steps: (1) designing the CNN architecture, (2) learning the parameters of the CNN (also called "training"), and (3) using the defined CNN for inference. Since CNNs are back-propagation learning algorithms, their learning phases can be divided into: *forward propagation*, *backward propagation* and *weight update*. In the forward propagation phase, input data are sent to the neural network to generate the outputs. In the backward propagation phase, the errors between the standard outputs and the produced outputs are propagated in a backward fashion to compute the errors in each layer. These errors (also called *gradients*) in each layer will be used in every weight update. However, for inference, the parameters of the networks are given and there is only forward propagation to produce the prediction.

Chapter 3 Bioinformatics Applications on CPU-GPU Clusters

3.1 Motivation

The pairwise sequence alignment algorithms, both local and global [4, 8], are in many ways the core technology for the study of biological sequences. They have key roles in multiple sequence alignment [9], phylogenetics[10], and molecular evolution studies [11]. It adopts the dynamic programming approach to alignment which is $O(n^2)$ in time complexity. The biologists often wish to make millions or even billions of such comparisons [12], which are extremely time consuming. To reduce the computational complexity, some heuristic methods are proposed to improve the basic dynamic programming approach. Examples can be found in sequence database search programs such as FASTA [13] and BLAST [14, 15] and various forms of genome assembly algorithms [16, 17]. Such acceleration is useful in some cases. However, these heuristics depend on the assumption that the vast majority of the sequence pairs being compared have essentially no similarity and that, once this fact has been demonstrated for a sequence pair, the computation of the alignment itself is unnecessary.

Increasingly a second class of problem is becoming relevant. In this case, there is a requirement to compare very large numbers of sequences that are all evolutionarily related. As a result, it is not possible to omit the computation of any of the alignments, making approaches such as that of BLAST inappropriate. One example is the computation of very large multiple sequence alignments for analyses such as inference of the “tree of life” [18-20]. A similar problem motivates my work in this thesis, namely the analysis of complex microbial communities through the sequencing of a particular microbial gene, the 16S rDNA gene. Biologists have discovered that many microbes

cannot be cultured under laboratory conditions but that it is possible to assess their presence through the direct sequencing of the DNA in an environment [21-25]. To compare microbial communities across environments, it is helpful to survey a single gene: the 16S gene is useful in this regard as it is essentially ubiquitous across prokaryotic life. However, the sequencing of the gene is only a first step: it is then necessary to compare the sequences generated to each other and to other known 16S sequences to assess the taxonomic diversity present in the sample. As there are hundreds of thousands of 16S sequences in sequence databases and tens of thousands of unique sequences among those [5], this analysis can be daunting.

The problem as stated is clearly highly parallel. To address this problem, the massively parallel computing potential of GPUs is brought in. General-purpose graphics processing units (GPGPUs) are advancements of hardware originally developed to accelerate complex graphical rendering for applications like 3D gaming. However, the state-of-the-art GPUs can be programmed for various general applications. The programming interfaces include the CUDA framework proposed by Nvidia and OpenCL, which is an open standard for parallel programming heterogeneous system. As GPGPUs are increasingly becoming part of HPC clusters, how to levering GPUs in a distributed clusters for large-scale sequence alignment becomes an emerging research problem.

3.2 Related Work

3.2.1 Early Works on Sequence Alignments

To study the biological sequences, pairwise sequence alignment algorithms are heavily used in various applications (e.g. sequence database search). These algorithms can be

categorized into (1) local sequence alignment (e.g. Smith-Waterman algorithm [8]) and (2) global sequence alignment (e.g. Needleman-Wunsch [4]). SW is used when the optimal common subsequence between two sequences is needed to identify. NW, on the contrary, is the algorithm to align two sequences.

Both SW and NW are based on dynamic programming. The algorithms start with an initialized matrix and the calculation is performed from left-top to right-bottom. Due to the computation intensive nature of this approach, some heuristic solutions are proposed to address similar problem, at the cost of reducing accuracy. FASTA [13] and BLAST [14, 15] are two famous methods among those heuristic solutions.

3.2.2 Acceleration of Bioinformatics Application on GPUs

In recent years GPUs and other accelerator devices have been widely used to accelerate a variety of scientific applications from many domains [26-28]. In particular, a number of biological applications, including BLAST [29], hidden Markov models [30-32] and structure comparisons [33], have been ported to GPU or FPGA architectures. Among GPU-enabled HPC Bioinformatics software, CUDA-BLASTP [34] is designed to accelerate NCBI BLASTP for protein sequence databases search. MSA-CUDA [35] is a parallel multiple sequence alignment program accelerating all three stages of ClustalW [9] processing pipeline. In next-generation sequencing technologies, there are already different proposes [36-38] using GPUs to accelerate various applications. Most relevant to my work in this thesis are several sequence alignment algorithms implemented on GPU [27, 39-41]. In chapter 3, we will provide more background on one of these: the NW implementation in the Rodinia benchmark suite [27].

3.2.3 Sequence Alignments on GPUs

Among the alignment implementations, Liu et al., [42] present an optimized sequence database search tool based on the Smith-Waterman (SW) local alignment algorithm (in contrast to the NW global alignment problem considered here). Compared to other implementations [39, 43, 44], their tool provides better performance guarantees for protein database searches. Li et al., [45] offer a GPU acceleration of SW intended for a single comparison of two very long sequences; we focus on accelerating many pairwise alignments of shorter sequences. We are interested in the NW problem, which rather than being used for database search is more commonly applied to situations where all possible pairwise alignments are required (e.g., alignments for phylogenetics or metagenomics as described above). In their first phase (computation of the alignment matrix), NW and SW share similar computation patterns, so optimization techniques can be reused between the two methods.

There are also distributed CPU-based implementations of NW: for example ClustalW-MPI [46] aligns multiple protein, RNA or DNA sequences in parallel using MPI. Biegert et al., [47] have introduced a more general MPI bioinformatics toolkit in the form of an interactive web service that supports searches, multiple alignments and structure prediction. Our tool differs from these in combining MPI and CUDA to allow deployment on CPU-GPU clusters where multiple GPUs may be employed simultaneously.

3.3.4 Rodinia-NW: Needleman-Wunsch on GPU

The Rodinia benchmark suite [27] offers a GPU parallelization of NW (hereafter, Rodinia-NW), that is used as baseline in this thesis. Rodinia-NW operates as follows. Since each element in the alignment matrix depends on its left-, upper- and left-upper-neighbors, a way to exploit parallelism is by processing the matrix in minor diagonal manner. Each minor diagonal depends on the previous one, thus leading to the need for iterating over minor diagonals. However, at every iteration, all the (independent) elements in the same minor diagonal line can be calculated simultaneously. If the matrix is laid out in global memory in row-major order, the involved memory access patterns are uncoalesced, potentially leading to performance degradation. Since each element in the alignment matrix is used for calculating three other elements, performance can be improved by leveraging shared memory and dividing the alignment matrix in square tiles (each of them fitting the shared memory capacity). Rodinia-NW performs tiling and exploits two levels of parallelism: (i) within each tile elements are processed in minor diagonal manner, and (ii) different tiles in the same minor diagonal line can also be processed concurrently by distinct thread-blocks. Threads within the same thread-block manipulate the data and store elements in shared memory temporarily. After the computation of a tile completes, all of the data are moved to global memory using coalesced accesses. For square alignment matrices and tiles of width N and T, respectively, Rodinia-NW's parallel kernel is invoked   times (once for each minor diagonal of tiles). After carefully analyzing Rodinia-NW, we found the following limitations.

First, Rodinia-NW is designed for a single pairwise comparison. Applications such as those above require hundreds to thousands of comparisons. As such, they introduce a second exploitable level of parallelism, especially as each pairwise comparison is independent. Moreover, the sequences generally differ in length but Rodinia-NW only supports sequences of equal length, requiring padding to handle more general cases.

Second, Rodinia-NW requires three data transfers for each alignment, an approach that can be improved. Before kernel launch, the alignment matrix is initialized (with the gap information) on the CPU. Next, alignment matrix and score matrix are copied from CPU to GPU. The alignment matrix is processed on the GPU, and finally copied back to the CPU. We note that the two copies of the alignment matrix are $O(nm)$ each. However, the first data transfer of the alignment matrix can be avoided by initializing its 1st row and 1st column directly on the GPU.

Finally, CUDA does not support global barrier synchronization among thread-blocks within a parallel kernel (an implicit global synchronization takes place at the end of each kernel execution). Since in Rodinia-NW each tile is mapped to a thread-block and tiles must be processed in diagonal strip manner, a global synchronization among thread-blocks operating on the same diagonal is required before proceeding to the next diagonal. This is accomplished by invoking multiple kernel launches from the host side. This approach has two limitations: (i) each kernel launch has an associated overhead (that depends on the GPU device), and (ii) the GPU is underutilized by kernel launches that process small numbers of tiles (i.e., those corresponding to the first and the last diagonals).

3.3 Design of GPU-workers

In this Section we describe three alternative implementations of multiple pairwise alignments using NW on GPUs: TiledDScan-mNP, DScan-mNP and RScan-mNP [48-50]. All these implementations, exemplified in Figure 4, Figure 5 and Figure 6, aim to overcome the limitations pointed out above.

3.3.1 TiledDScan-mNW: Multiple alignments with tiling

The first method (TiledDScan-mNW) is a directed extension of Rodinia-NW to multiple pairwise alignments. This approach still uses tiling and operates in diagonal strip manner, performing multiple kernel invocations to compute the alignment matrices. However, for each kernel invocation, multiple alignment matrices are concurrently processed using different thread-blocks (and SMs). This is illustrated in Figure 5, where we concurrently perform three pairwise comparisons: (seq_1, seq_2), (seq_1, seq_3) and (seq1, seq4). In the first iteration, the top-left tiles of the three matrices are processed in parallel by three thread-blocks, and thus mapped onto three streaming multiprocessors: SM_1, SM_2 and SM_3. In the second iteration, the tiles of the second minor diagonal of the three matrices are processed in parallel by six thread-blocks, and thus mapped onto streaming multiprocessors SM_1-SM_6. Note that, for *m* pairwise comparisons, the number of kernel invocations of TiledDScan-mNW is reduced by a factor *m* (as compared to Rodinia-NW); for each kernel call, the number of thread-blocks is increased by a factor *m*. This has two advantages: (i) a limited kernel invocation overhead, and (ii) an improved GPU utilization. Execution configurations with a large number of threads allow not only exploiting all the SMs and cores available on the GPU, but also hiding the global memory

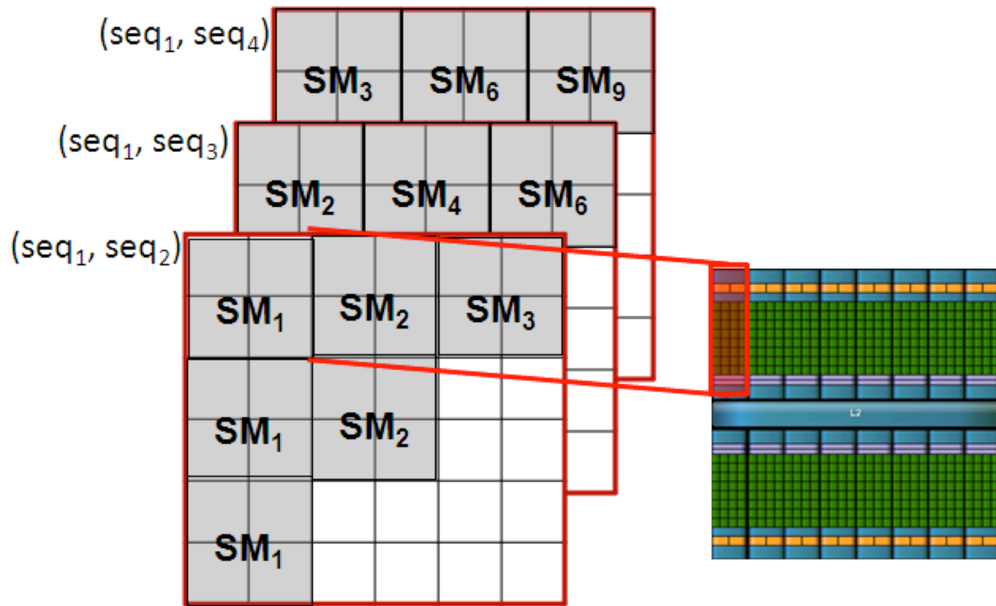


Figure 5: TiledDScan-mNW and the mapping to GPU cores and SMs

access latencies (and NW is a memory-intensive application). Being an extension of Rodinia-NW, TiledDScan-mNW retains its advantages: regular computational patterns and coalesced memory access patterns when storing alignment data from shared memory to global memory.

3.3.2 DScan-mNW: Single-kernel diagonal scan

TiledDScan-mNW still requires multiple kernel invocations to perform *m* pairwise alignments. Even if the parallelism within each kernel call is improved by a factor *m* compared with Rodinia-NW, some kernel invocations still exhibit limited parallelism (and limited opportunity to hide memory latencies). Our second implementation – DScan-mNW – performs a diagonal scan with a single kernel call. As illustrated in Figure 6, in this case each alignment matrix is assigned to a thread-block (and mapped

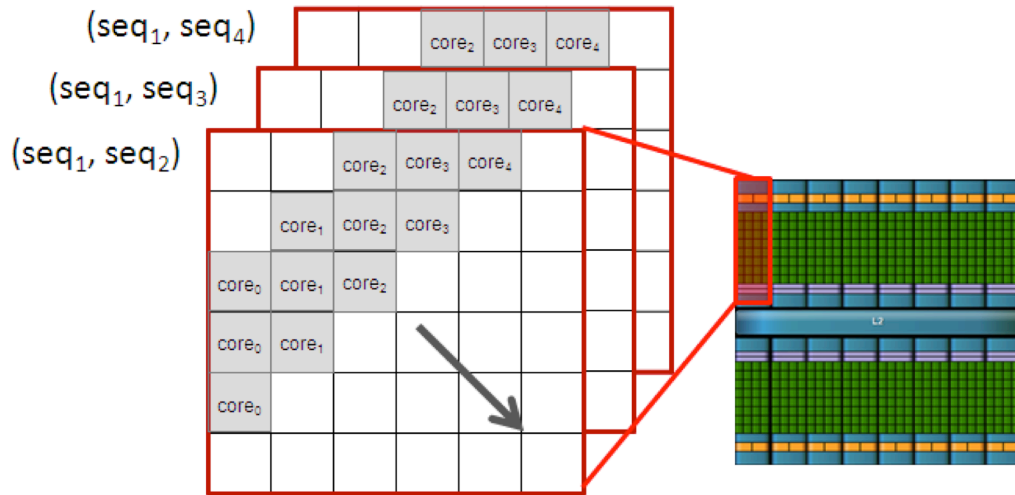


Figure 6: DScan-mNW and the mapping to GPU cores and SMs

onto a SM). No tiling is performed. The computation iterates over diagonals. For each diagonal, every element is processed by a thread (and mapped onto a core).

To limit the number of expensive accesses to global memory, the computation is fully performed in shared memory. The alignment matrix is stored in row-major order in global memory and in minor diagonal order in shared memory. According to equation (1), at each iteration three diagonal lines are required: the first two diagonal lines cache previous data and the third one contains the newly computed elements. Once computed, this third line can be copied from shared to global memory. At that point, the first diagonal line can be discarded and the shared memory reused for the next iteration. To summarize, the matrices are created in shared memory and moved to global memory diagonally. The main disadvantage of this approach is the uncoalesced memory accesses required to store diagonal data to global memory. We found that the latencies of such irregular access patterns can be effectively hidden by using large numbers of threads.

The computational pattern of our DScan-mNW is similar to the SW intra-task parallelization proposed by Liu et al. [42]. However, [42] avoids uncoalesced memory

accesses by storing the alignment matrix in global memory in minor diagonal order. We found that, when using large thread-blocks to hide memory latencies (e.g., 512 threads/block), the overhead due to uncoalesced memory access patterns is reduced to 10% and 7% of the execution time on Fermi and Kepler GPUs, respectively (the exact percentage depends also on the clock-rate of the memory system). On the other hand, storing the alignment matrix in row-major facilitates the trace-back operation (which is not considered in [42]) in two ways: first, it avoids the need for complex index translation; second, the more regular data layout leads to better caching properties.

3.3.3 RScan-mNW: Row scan via single CUDA core

Our third method – RScan-mNW – uses a *fine-grained matrix-to-core mapping* and a *row-scan* approach. First, each alignment matrix is computed by a single GPU core. Second, to allow regular compute and memory access patterns, each alignment matrix is computed row-wise (rather than diagonal-wise). This computational pattern is illustrated

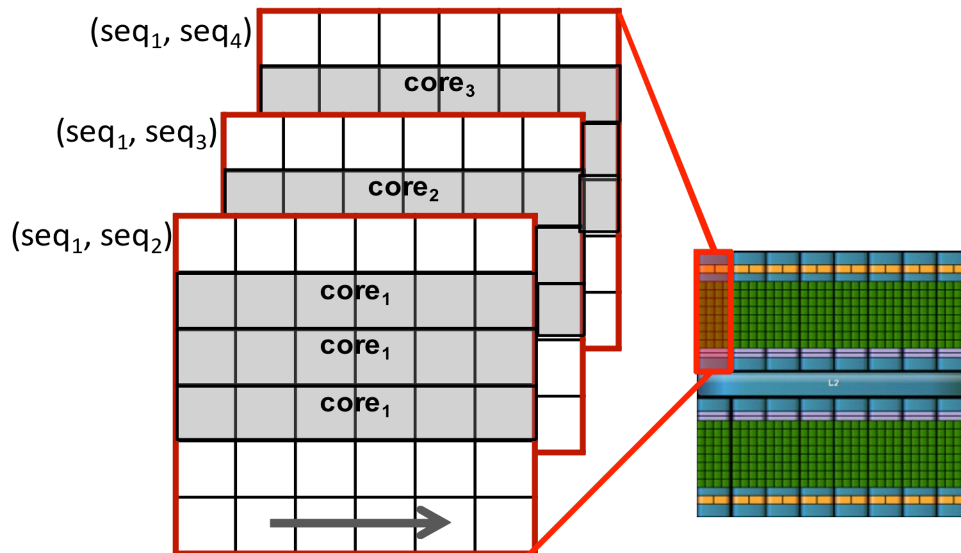


Figure 7: RScan-mNW and of the mapping to GPU cores and SMs

in Figure 7.

This method leverages shared memory in order to allow data reuse and minimize the global memory transactions. The parallel kernel iterates over the rows of the alignment matrices. At every iteration, only two rows per matrix must reside in shared memory: the previously computed one and the one containing newly computed elements. Only the left-most element of the new row must be loaded from global memory; for the rest, the computation happens solely in shared memory. Once the new row has been computed, it is copied from shared to global memory. The previously computed row can be discarded, and the new one can be cached for use in the next iteration. The kernel has two phases: computation and communication. In the computation phase, the threads within a thread-block operate fully independently: each thread computes the data corresponding to the row of an alignment matrix and stores them in shared memory. In the communication phase, threads belonging to the same thread-block cooperate to transfer row data from shared to global memory in a coalesced fashion (that is, each alignment matrix is transferred cooperatively by multiple threads). In case of very long sequences, rows are split into sections so as to fit into shared memory. The size of these sections is configurable. Large sections require more shared memory, which in turn limits the number of active threads on each SM. Small sections (e.g. sections with less than 32 elements) lead to warp underutilization in the communication phase, which in turn can hurt the performance. The usage of shared memory is a major concern in the kernel configuration process. The per-block shared memory can be calculated using the following formula:

$$shmem = 3*\text{sizeof(int)}*\textit{BLOCK_SIZE}*\textit{SECTION_SIZE}$$

Each thread stores three sets of data: the sequence data and two sections of the alignment matrix. Each thread-block performs *BLOCK_SIZE* pairwise alignments using sections of size *SECTION_SIZE*. By setting the *BLOCK_SIZE* and the *SECTION_SIZE* to 32, we use 12KB of shared memory with no warp underutilization. With this setting, each SM can concurrently run up to four thread-blocks.

The advantages of this approach are twofold. First, the computational pattern is extremely regular: unlike diagonals, rows are all of the same size. Second, data transfers between shared and global memory are naturally coalesced. The main drawback to this approach is that the parallelism is limited by the GPU memory capacity. For example, if the sequences to be compared are of length 2,000 and the alignment matrices contain 4-byte integers, then each matrix will be of size 32MB. To fully utilize the cores of typical GPUs (say 480 cores), we should allow 480 parallel pairwise comparisons, requiring a total of roughly 15GB of memory. This number considerably exceeds the 1-5GB of memory present on most GPUs. Therefore, on long sequences RScan-mNW will tend to underutilize the GPU resources. On the other hand, this approach is very promising for short sequences (e.g. <500). For long sequences, an alternative optimization would be to break the alignment matrices into smaller strips to reduce the memory footprint, and use dual-buffering to move previously computed strips to the CPU while computing new ones. Finally, we note that certain scoring schemes allow for linear memory NW algorithms of minimal complexity: under these limited and less-commonly used schemes, highly efficient parallelism could be achieved using RScan-mNW.

The computational pattern of our RScan-mNW is similar to the SW inter-task parallelization proposed by Liu et al. [42]. However, their proposal does not use shared

memory in the kernel and adopts a different data layout in global memory. Specifically, to avoid uncoalesced global memory accesses, Liu et al. place data corresponding to different alignment matrices into continuous global memory space. For instance, the i_{th} element of global memory is from the ith alignment matrix, while the $(i+1)_{th}$ element is from the $(i+1)_{th}$ alignment matrix. This memory layout leads to poor data locality during the trace-back phase. As mentioned above, trace-back is not considered in [42], but is a necessary operation in the problem we consider.

3.4 Experimental Evaluation

In this section, we present two sets of experiments: (i) single GPU experiments, and (ii) cluster experiments. The former are meant to evaluate our GPU implementations of multiple sequence alignment with NW and the latter are to evaluate our distributed framework. Particularly for the cluster the experiments, we focus on the scalability issues.

3.4.1 Experimental setup

Hardware setup – Single GPU experiments have been performed on a variety of low-end and high-end GPUs, listed in Table 1.

Software setup – The CUDA 5.0 driver and runtime are installed in all the machines used. The OS in use is CentOS5.5/6 with g++4.1.2. Each data point represents the average across 3 executions.

Dataset – Our reference dataset consists of about 25,000 unique 16S rDNA genes from the Ribosomal Database [5]. The sequences are on average 1,536 bases long.

3.4.2 Performance on single GPU

Our first set of experiments is meant to evaluate our GPU implementations and compare them with Rodinia-NW. In Chapter 3.3.2, we noted two limitations in Rodinia-NW: unnecessary memory transfers from CPU to GPU and inefficiencies in the computational kernel and its invocations. Below, we will show how we improve performance with respect to both limitations.

Table 1: Characteristics of the GPUs used in our evaluation

GPU	Type	Values
Low-end GPUs	*Quadro 2000*	4 SM x 48 cores ~1 GB Global memory
	GTX 460	7 SM x 48 cores ~1 GB Global memory
	GTX 480	15 SM x 32 cores ~1.5 GB Global memory
High-end GPUs	*Tesla C2050*	14 SM x 32 cores ~2.6 GB Global memory
	Tesla C2070/C2075	14 SM x 32 cores ~5 GB Global memory
	Tesla K20	13 SM x 192 cores ~4.7 GB Global memory

Memory Transfers: As explained in Chapter 3.3.2, Rodinia-NW initializes the alignment matrix on CPU and copies it to GPU. Also, to simplify memory access during computation, it creates a temporary substitution score table of size m x n during CPU initialization. For problems of the size considered, data transfer consumes considerable amount of time. An obvious optimization is to move the initialization from CPU to GPU. In addition, by omitting the creation of the temporary substitution table, more alignment matrices can be accommodated on the GPU, thus allowing for increased parallelism. In Figure 8 we show the effect of these optimizations on different GPUs. In all experiments, 64 pairwise alignments are performed. The optimized version initializes the alignment matrices on GPU and avoids the initial CPU-to-GPU data transfer. On top of this, the optimized + pinned memory version uses pinned memory. As can be seen, the proposed memory optimizations lead to a 5-10% and a 20-25% decrease in execution time on low-end and high-end GPUs, respectively. In addition, the combination of the memory

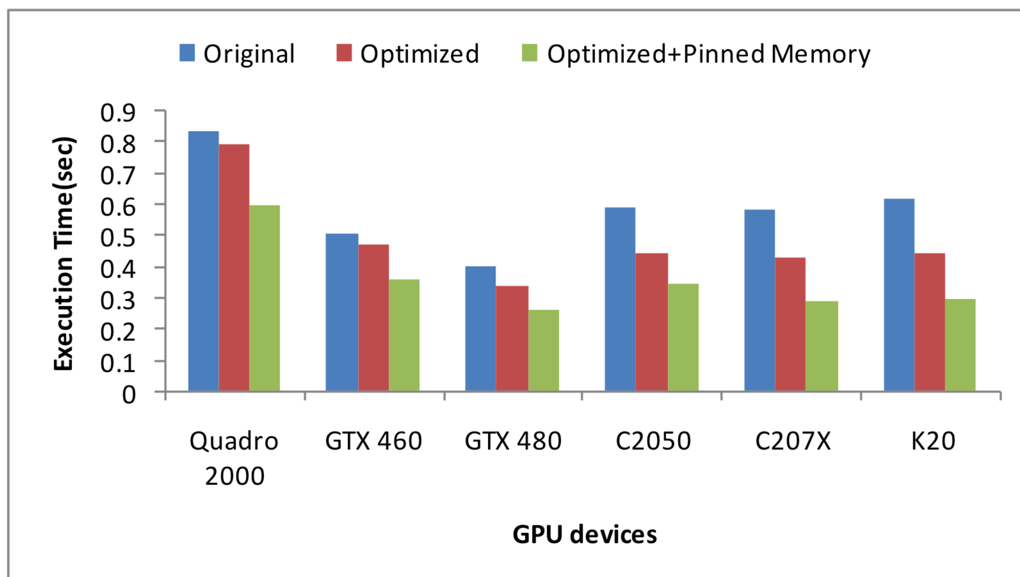


Figure 8: Evaluation of memory optimizations on Rodinia-NW

optimization with the use of pinned memory leads to a decrease in execution time in excess of 30% and 50% on low-end and high-end GPUs, respectively.

Kernel computation: We now focus on the performance of our compute kernels. Our analysis has two goals: (i) evaluating the performance improvements over Rodinia-NW, and (ii) devising criteria for selecting the optimal GPU implementation depending on the underlying GPU device. In Figure 9 and 10, we show the relative speedup in kernel computation time of DScan-mNW and TiledDScan-mNW over Rodinia-NW (the speedup is computed as the ratio between the compute time of Rodinia-NW and that of our GPU implementations). We performed experiments on all available GPUs and varied the number of pairwise comparisons performed from 8 to 64. Given its fine-grained alignment-to-core mapping, on these datasets RScan-mNW underutilizes the GPUs and reports poor performance. This, in general, holds when comparing long sequences on GPUs with 1-5GB device memory. Therefore, we focus on the other schemes.

Figure 9 reports the speedup of TiledDScan-mNW over Rodinia-NW. Note that TiledDScan-mNW performs fewer kernel calls (and therefore, has less kernel overhead) and involves more per-kernel computation (thus leading to increased parallelism). This motivates the performance improvement achieved by TiledDScan-mNW over Rodinia-NW. Note that the speedup increases with the computational power of the GPU (from 1.2x on the Quadro2000 to 2x on the K20). In fact, the increased parallelism in the TiledDScan-mNW kernel can be better serviced by GPUs with more SMs and compute cores.

As can be seen in Figure 10, DScan-mNW also outperforms Rodinia-NW on all devices and datasets. Its performance is also generally better than that of TiledDScan-mNW, except on Tesla C207x cards. It is somewhat surprising that our approach does not show substantial speedup over Rodinia-NW on this device. It must be said that NW is an integer application, and Tesla GPUs are optimized for larger memory capacity (5GB vs.

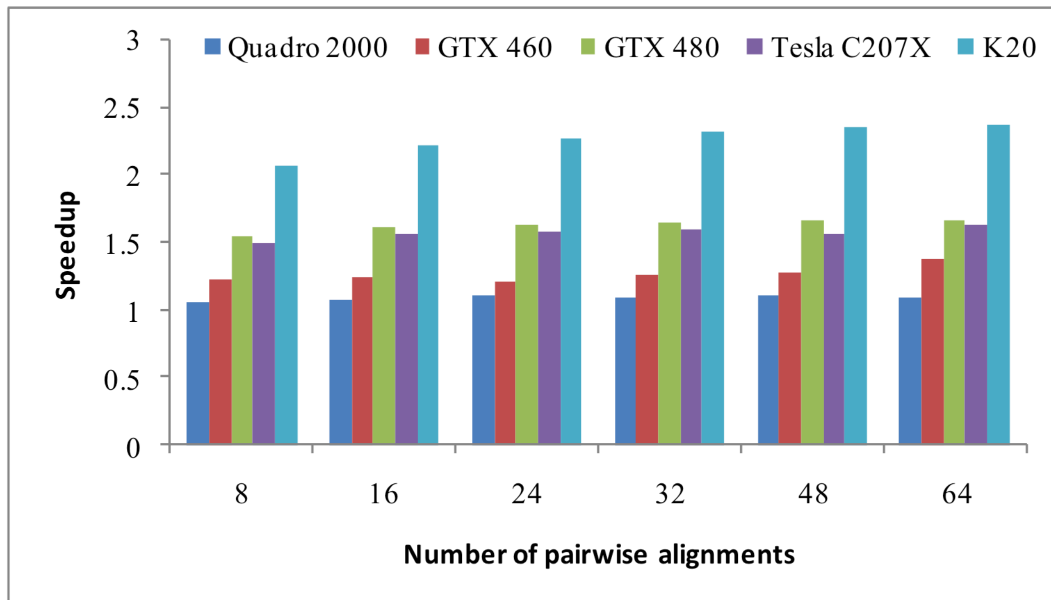


Figure 9: TiledDScan-mNW Kernel speedup over Rodinia-NW

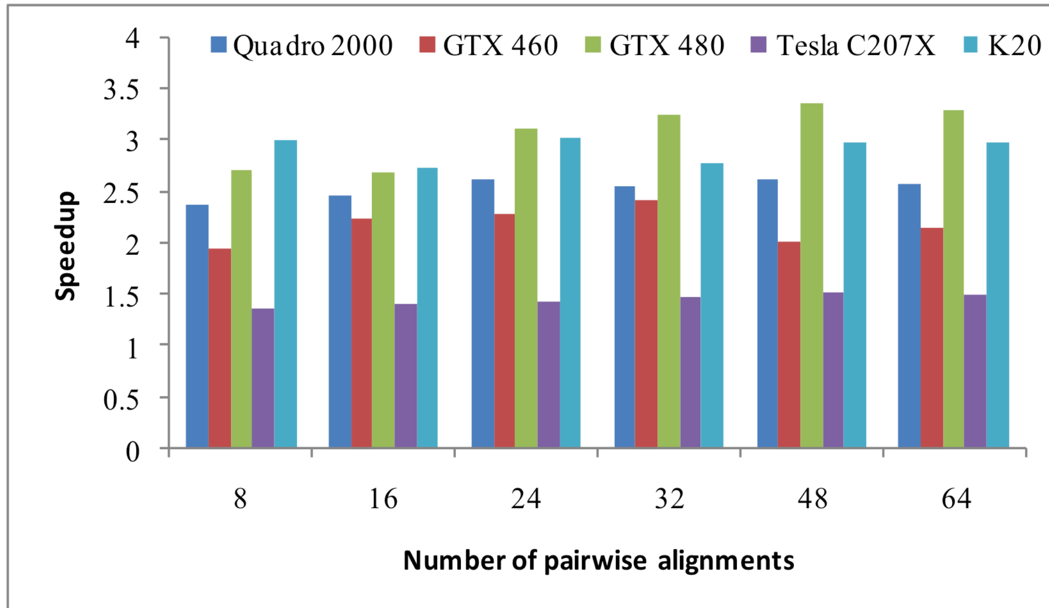


Figure 10: DScan-mNW Kernel speedup over Rodinia-NW

1GB) and improved support for double precision floating point operations, but have a reduced clock rate (1.15GHz vs. 1.4GHz in GTX 480 cards, for example). We believe that the high number of uncoalesced memory accesses performed by DScan-mNW may motivate the poor performances on Tesla C207x cards, which have a slower memory clock.

Figure 11 reports the speedup of RScan-mNW over Rodinia-NW on sequences of different lengths. The number of pairwise comparisons performed in each experiment is reported on top of each bar (all experiments have been configured so to use 70% of the global memory capacity). As mentioned in Chapter 3.3.3, in RScan-mNW each core computes an alignment matrix: in order to fully utilize the computational resources of the GPU, RScan-mNW needs to perform a large number of parallel sequence alignments. This leads to pressure on the global memory capacity: for long sequences, the GPU global memory becomes the bottleneck, and the performance of RScan-mNW is penalized. RScan-mNW's performance improves when the length of the sequence

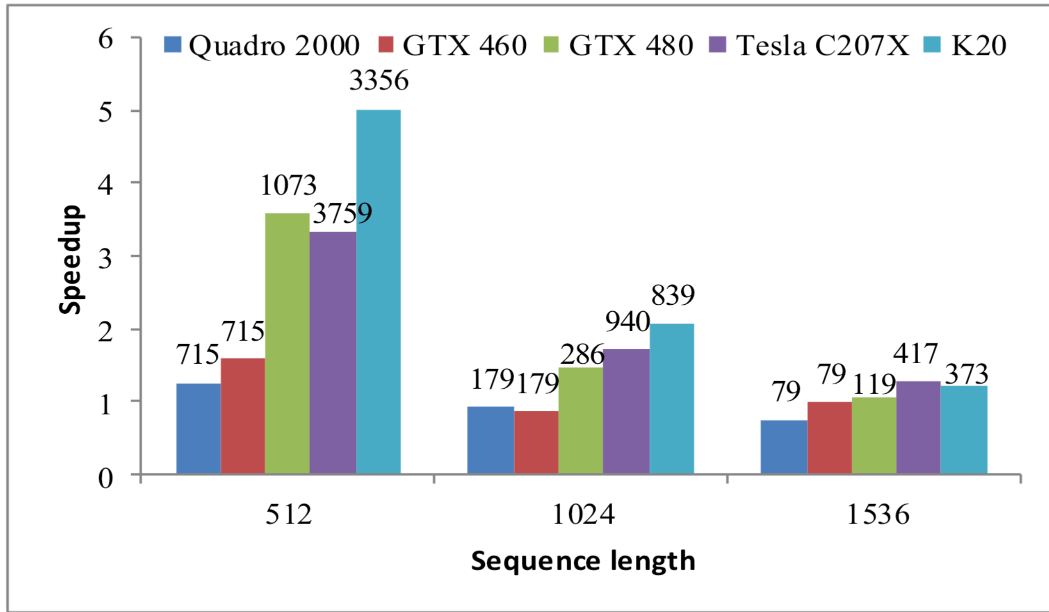


Figure 11: Kernel speedup of RScan-mNW on sequences of various lengths

decreases: in the case of shorter sequences, the global memory can accommodate more alignment matrices, thus leading to higher utilization of the GPU cores. In particular, on 512-base sequences, RScan-mNW gives a speedup over Rodinia-NW up to a factor 5x.

In general, Figure 9, 10 and 11 show that DScan-mNW and TiledDScan-mNW are preferable to RScan-mNW on the 1,536-base sequences in the 16S rDNA gene dataset. In addition, these results show that our methods overcome inefficiencies of Rodinia-NW, and suggest that DScan-mNW is preferable on all devices except Tesla C207x. On such cards, TiledDScan-mNW provides better performance. This finding will be used to configure our GPU-workers. As next step, we want to determine how to size the amount of work that each GPU-worker should pull from the GPU-dispatcher to operate at full capacity. In fact, we want to fully utilize the GPUs present in the system. The number of pairwise comparisons that can be performed concurrently on each GPU is limited by its memory capacity. We configured each GPU to operate with its global memory 75% full.

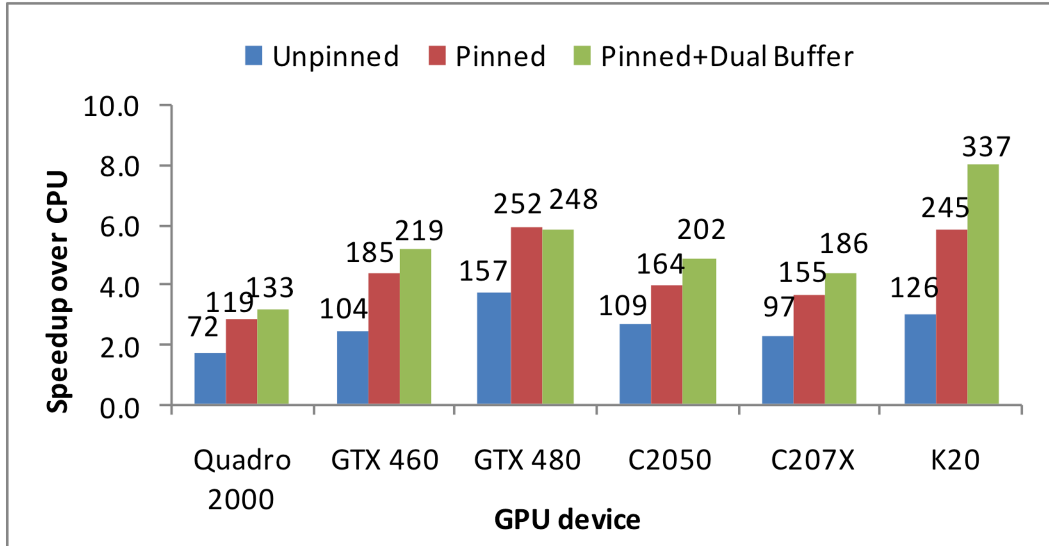


Figure 12: Speedup of DScan-mNW over an 8-threaded CPU implementation

For the sequence lengths being considered, this leads to 79, 79, 119, 208, 417, and 372 parallel alignments on Quadro2000, GTX460, GTX480, Tesla C2050, Tesla C207x and K20 GPU, respectively.

Figure 12 shows the speedup reported by Dscan-mNW over an 8-threaded OpenMP implementation running on the 8-core CPU on Node-4 (see Table 2). The numbers on each bar represent the throughput in number of pairwise alignments/sec. For each GPU, we performed three experiments: one using unpinned memory, one using pinned memory, and one using double buffering. We first define an "optimal batch size" b_{SIZE} for a particular GPU to be the number of simultaneous alignments that can be performed given the device memory (as above). For the first two versions, we ran analyses consisting of a number of sequences equal to 3 b_{SIZE} in order to effectively time the computation. For double buffering, only half of the GPU memory performs alignments at one time, so 6 batches of size $b_{SIZE}/2$ were timed. The performance was measured as the number of sequence pairs compared per second.

As can be observed from Figure 12, switching to pinned memory offers a gain of roughly 1.6x, consistent with previous findings [51]. Not using the OS's virtual memory system could in principle limit the number of sequences that can be processed concurrently. However, our observation is that problem sizes are instead generally limited by the amount of physical memory on the GPU, so we do not consider this CPU-based memory limitation to be a significant disadvantage. The application of double-buffering along with pinned memory offers an additional average 1.2x speedup, with the exception of the GTX480 system, which does not show significant speedup. We speculate that the reason for this lack of improvement is that the GTX480 has a more restricted handling of CUDA streams, which does not allow the same level of overlapping of memory transfers and kernel computations possible on other devices. In general, it can be observed that even cheap low-end GPUs (like the GTX460 and GTX480) offer throughput in the order of 200-250 pairwise alignment/sec.

3.5 Other Applications with Similar Computation Pattern

In many computer vision tasks ranging from image search to multi-target tracking, feature probability maps represented as histograms play a critical role in the overall computation. The integral histogram for images is an efficient preprocessing method for speeding up diverse computer vision algorithms. Similarly to the Needleman-Wunsch algorithm, the integral histogram computation follows a dynamic programming pattern. We have explored different techniques to efficiently compute integral histograms on GPU and propose two GPU implementations: *CW-TiledHV* and *WF-Tiled*.

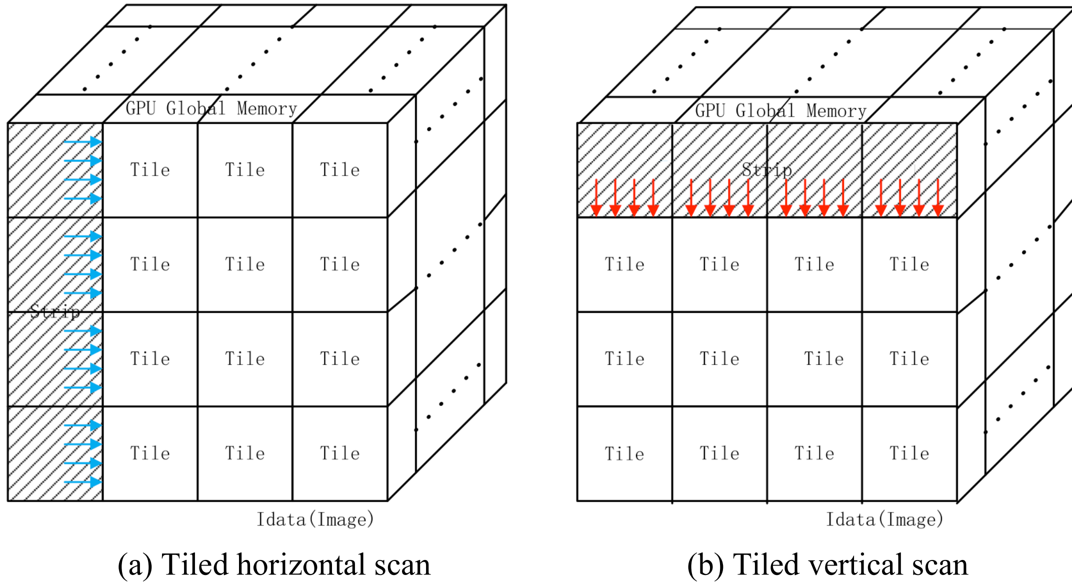
Cross-wave Tiled Horizontal-Vertical Scan (CW-TiledHV): As represented in Figure 13, the CW-TiledHV approach operates as follows. First, each of the *b* matrices of size (*h* x *w*) corresponding to the different bins is divided into tiles. Each tile must be small enough to fit in shared memory and large enough to contain sufficient amount of data for computation work. In our implementation, we use squared tiles. The processing is divided into two stages: the horizontal scan (Figure 13(a)) and the vertical scan (Figure 13(b)). In each stage, the computation is performed on strips with the width of the tile. A kernel call operates on a strip; the computation is performed strip by strip until the whole matrix has been processed. The total number of image tiles or blocks processed is given by:

$$Tiles = (W_{Image}/W_{Tile}) \times (H_{Image}/W_{Tile})$$

In horizontal scan, the number of vertical strips is equal to:

$$VStrips = W_{Image}/W_{Tile}$$

and during the vertical scan the number of horizontal strips is equal to:



(a) Tiled horizontal scan (b) Tiled vertical scan

Figure 13: Tiled horizontal-vertical scan (CW-TiledHV)

$$HStrips = H_{Image}/W_{Tile}$$

We expect the image sizes to be evenly divisible by the tile sizes other wise the image is appropriately padded. In the kernel implementation, each tile is assigned to a thread-block, and each row/column is assigned to a thread. Shared memory is used to allow efficient and coalesced memory accesses. Threads belonging to the same block push the wave-front forward (either from left to right or from up to bottom in Figure 13). Since each thread-block consists of warps, in order to avoid thread divergence within warps and GPU underutilization, the tile size must bet set to be a multiple of the warp size (32).

Wave-front Tiled Scan (WF-Tiled): The use of separate horizontal and vertical scan kernels in the CW-TiledHV method has a drawback: it causes each tile to be transferred multiple times between global and shared memory. In fact, in both scan kernels, each tile is first moved from global into shared memory, then processed, and then moved back to global memory. This fact is exemplified in Figure 14. As a consequence, combining the horizontal and vertical scans into a single kernel will allow accessing global memory only twice per tile (once in read, and once in write mode). Actually, for the horizontal scan, the data in each row rely on the data on their left; for the vertical scan, the data in each column rely on the data on their upper position. This data access pattern is quite

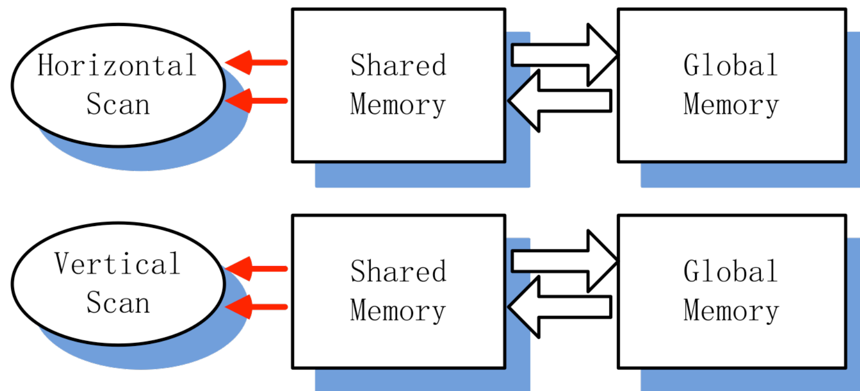


Figure 14: Data flow of CW-TiledHV implementation

similar to the Needleman-Wunsch algorithm in this chapter. Therefore, we can arrange the computation in a similar fashion, and compute the integral histogram using a front-wave scan. This approach, that we call Wave-front Tiled Scan (WF-Tiled), is shown in Figure 15. Similarly to the CW-TiledHV implementation, we divide the h x w matrix into different tiles. Again, each tile should be small enough to fit in shared memory, and large enough to contain non-trivial amount of computation work. All the tiles lying on the same diagonal line are considered part of the same trip and processed in parallel. Within the parallel kernel, each thread block will process a tile, and each thread will process a row (during horizontal scan) and a column (during the vertical scan) of the current tile. The tricky part of this implementation is the following: after the horizontal scan, the last column of each tile (that would otherwise be overwritten during the vertical scan) must be preserved. In fact, the last column must be used in the horizontal scan of the next strip. This can be easily achieved by storing the extra data in global memory (the additional

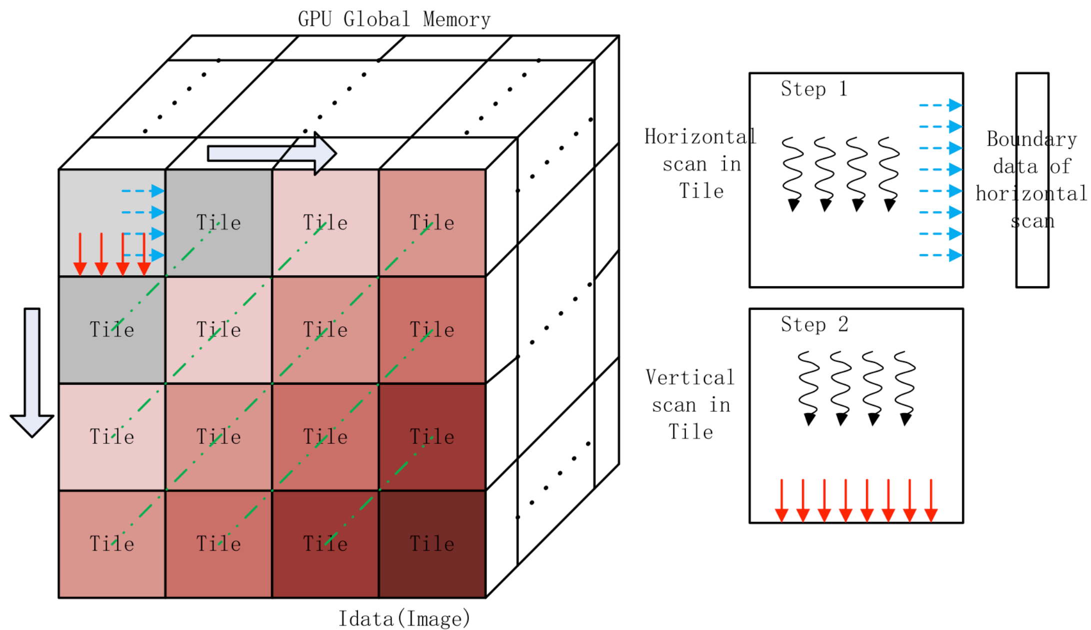


Figure 15: WF-Tiled implementation

memory requirement correspond to a single array of *h* elements). By eliminating unnecessary data movements between shared and global memory, the WF-Tiled method can potentially be preferable to the CW-TiledHV.

Chapter 4 Irregular Applications on Many-Core Processors

Parallelization of regular of regular applications (such as those operating on dense matrices and vectors) on many-core processors has been extensively investigated. However, parallelization of irregular applications continues to be challenge. Irregular applications are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence, and a degree of parallelism that is known only at runtime (rather than at compile time). In fact, the amount of parallelism within irregular applications depends on the characteristics of the dataset, rather than solely on its size. Yet, many established and emerging applications are irregular in nature, being based on irregular data structures, such as graphs and trees.

This chapter focuses on addressing important issues related to the deployment of irregular computations on many-core processors. Specifically, my contributions are in three directions:

(1) ***Unifying programming interfaces for many-core processors***. We proposed a compilation and runtime framework that generates efficient parallel implementations of generic graph applications for multi-core CPUs, Nvidia GPUs and Intel Xeon Phi coprocessors. Applications are implemented with a unified, platform-agnostic programming API and then our source-to-source compiler performs platform-specific code transformations and optimizations.

(2) ***Runtime support for efficient execution of applications on irregular datasets***. We analyzed the computational patterns of several irregular applications and found that the dynamic nature of the extracted parallelism makes it impossible to find an optimal

solution at compile time. So we proposed a runtime system able to dynamically transition between different implementations with minimal overhead, and investigated heuristic decisions applicable across algorithms and datasets.

(3) ***Compiler support for efficient mapping of applications onto hardware***. We proposed different parallelization templates for efficient code generation across various irregular applications and GPU architectures. In addition, we proposed a compiler-assisted workload consolidation method to enhance the efficiency of kernels with dynamic parallelism on GPUs.

4.1 Related Work

Irregular applications are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence, and a degree of parallelism that is known only at runtime (rather than at compile time). In fact, the amount of parallelism within irregular applications depends on the characteristics of the dataset, rather than solely on its size. Yet, many established and emerging applications are irregular in nature, being based on irregular data structures, such as graphs and trees.

There has been a rich body of work on the design of parallel algorithms to solve various graph problems (e.g., breadth-first-search [52-54], shortest paths [55, 56], minimum spanning trees [52, 57, 58], connected components [59-63]) on many-core platforms. Recent works (i.e. Parallel BGL [64], ParGraph [65], STAPL [66] and GraphLab [67]) have proposed parallel graph libraries for multi-core processors and distributed systems. For single node system, GraphChi [68] efficiently computes large graphs on a single CPU node. Green-Marl [69] offers a domain-specific language for

graph analytics [70-72]. The Galois system [73, 74] includes a programming model and a runtime component to dynamically extract parallelism from irregular applications by leveraging speculative parallelization.

As GPUs have become more general-purpose, the interest of research community has moved toward effectively deploying irregular applications on these many-core platforms. In particular, there have been several efforts [75-81] focusing on the acceleration of graph processing algorithms on Nvidia GPUs. Harish and Narayanan [75] were the first to perform this operation; their proposed implementations, however, are pretty basic and ineffective on sparse graphs used in practice. Better results have been reported through subsequent efforts, which focused on specific algorithms (breadth-first search [76, 77, 80], inclusion-based points-to analysis [78], strongly connected components [79, 82]). The optimizations introduced by these proposals are somehow orthogonal to our work, and can be integrated with it. Because of their higher generality, the proposals closest to ours are those by Hong et al [80, 81]. Hong et al. [80] proposes a virtual warp-centric programming model to allow datasets with different characteristics to more efficiently use the GPU hardware. This idea can be integrated with our work. Hong et al. [81] considers an adaptive solution that alternates CPU and GPU execution. We, on the other hand, focus on the automatic selection of different GPU solutions and on the conditions that make this beneficial.

Mapgraph [83], VertexAPI2 [84] and Gunrock [85] are GPU-targeting tools for graph analytics, not compiler frameworks to generate multiple GPU code variants for generic graph. They provide library implementations of specific graph algorithms and APIs to customize graph analytics. Users still require hand-coding in CUDA/OpenCL and

implementing predefined callback functions (gather, scatter). TOTEM [86] and Medusa [87] are more general programming frameworks for graph algorithms operating on static datasets (i.e., they do not include support for dynamic memory operations). Our work aims to automatically generate different code variants for GPU and Intel Phi devices starting from a platform-agnostic interface and to also support dynamic datasets. Unlike TOTEM, we do not consider graph partitioning across devices and cooperating CPU-GPU execution. Medusa also explores different graph storages (e.g., edge-oriented storage), graph-aware buffer schemes and multi-GPU execution. These mechanisms, however, can be incorporated in our framework.

Recent studies have analyzed the strengths and limitations of Nvidia's dynamic parallelism (DP). Although the effectiveness of dynamic parallelism has been demonstrated on certain applications (such as clustering algorithms [88], computation of the Mandelbrot set [89] and a particle physics simulation [90]), it has been shown that, because of the non-negligible overhead of this feature, the naïve use of DP can actually slow down the performance [91-93]. Wang et al. [93] have performed a characterization of DP-based implementations of unstructured applications, focusing on the analysis of their control flow behavior, their memory access patterns and the DP overhead. Yang and Zhou [91] have proposed a compiler framework to support nested thread-level parallelism without using DP. Their solution, which leads to spawning a massive number of threads, does not apply to recursive computations and is less effective on applications that exhibit high degrees of thread-level parallelism even before the proposed code transformations.

In our work, we propose code parallelization templates – with and without DP – to facilitate the efficient execution of irregular nested loops on GPUs. Besides that, we also propose generic code transformation techniques that apply equally to irregular loops and recursive applications to facilitate efficient use of dynamic parallelism, and we have automated these code transformations through compiler integration. Wang et al. [94] have proposed hardware architecture to support lightweight block execution of dynamically launched kernels. Conversely, our method is purely software-based, and is thus applicable on any GPU that supports dynamic parallelism, requiring no modification to the architecture.

4.2 Unified Programming Interface

Although many-core processors are widely used, they are relatively difficult to program, since they require programmers to be familiar both with parallel programming and with the features and the operation of these hardware platforms. This complexity is aggravated by the variety of software stacks used by the various many-core platforms.

In this work [95, 96], we intend to fill this gap and propose a compilation and runtime framework (Figure 16) for the effective deployment of *generic* graph applications on many-core processors (GPUs and the Intel Xeon Phi). Note that our framework also produces multi-threaded code for multi-core CPUs. Quite unlike previous work [86, 87], we consider graph applications that use *static* or *dynamic* datasets, and we free the programmer from the need to write specific parallel kernels for GPUs and the Intel Xeon Phi. Our framework hides the complexity and heterogeneity of the underlying hardware and software stack from the programmer. The programming API exposed to the user is

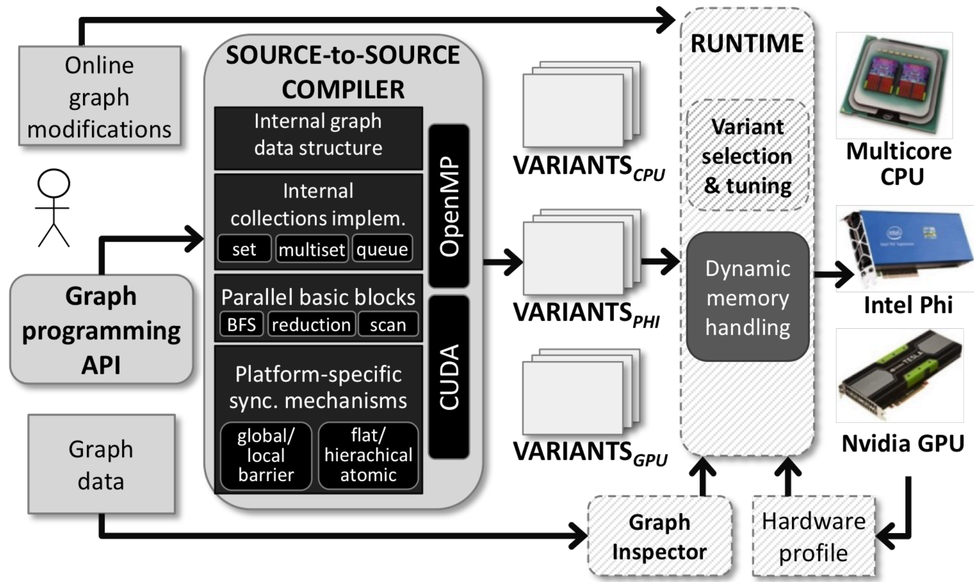


Figure 16: Proposed graph processing system

platform-agnostic, and includes a set of platform-independent sequential and parallel constructs. Our source-to-source compiler converts the graph and the containers (sets, multi-sets and queues) into internal, platform-specific data structures for multi-core CPUs, Intel Xeon Phi and NVIDIA GPUs. It then uses iterator-based templates to parallelize graph processing. The compiler generates different functionally-equivalent code variants for the target platforms. These variants differ in the parallelization strategy and in the optimized data structures on which they rely. Our runtime system, designed to support automatic selection of code variants and parameter tuning (based on profiling), also includes support for efficient dynamic memory management.

Unlike previous work [75-80, 97-100], we do not aim to optimize a specific algorithm on a particular many-core processor, but to automate the development of high-performance graph applications on many-core platforms. By leveraging our platform-agnostic programming API, the application developer delegates the complex task of tailoring the application to a particular platform to our tool-chain.

4.2.1 General Design & Methodology

Graph algorithms can be represented as a sequence of iterative steps. At each step, the algorithm performs some work on the elements of a *working set* and updates the working set (typically, by visiting neighbors of an element) by adding or removing elements. In every iteration, the elements of the working set may be processed in parallel (although synchronization mechanisms may be required to control concurrent data accesses). This computational pattern maps naturally to the *bulk synchronous* [101] style of parallelism.

Figure 16 describes our proposed graph processing system. The programming interface consists of a high-level graph API and a set of platform-agnostic, sequential and parallel constructs allowing the user to define generic graph applications. The source-to-source compiler generates different code variants for multi-core CPUs, Intel Phi coprocessors and NVIDIA GPUs. These code variants may differ in several aspects: from the type of parallelization performed, to the implementation of the underlying data structures [102], to the handling of nested parallelism, and more. The generated code is written in OpenMP and CUDA, and it uses the offload execution model on the Intel Phi. During code generation, the graph and the containers (sets, multi-sets and queues) are converted into internal, platform-specific data structures. In addition, existing parallel basic blocks are used for common primitives such as reduction, sort and scan. Parallelization is enabled by the presence of parallel iterators, which must be explicitly inserted in the code by the programmer. The compiler automatically handles synchronizations associated with the graph, the iterators and the containers. Synchronizations associated with custom data structures must be explicitly indicated by

the programmer using high-level, platform-independent synchronization primitives, which are converted into platform-specific synchronization mechanisms. Finally, the runtime system supports two important functions: (i) selecting the most suitable code variant depending on the characteristics of the application, the dataset and the underlying platform, and (ii) supporting dynamic memory allocation. In this paper, we focus on the second function and propose a runtime library for dynamic memory management. In addition, in Chapter 4.2.4 we provide guidelines to efficiently match the application to the hardware platform.

4.2.2 Programming API

We aim to provide the programmer with a familiar and easy-to-use programming interface, similar to existing ones designed for CPUs. To this end, we extend Green-Marl's API [69] with dynamic memory allocation and runtime primitives to support applications that use dynamic data sets (Green-Marl assumes static data sets), and we borrow some containers (ordered and unordered sets) from the Galois's system [73].

The resulting programming interface is summarized in Figure 17.

Graph API: The graph API includes the abstract data structure and high-level primitives that can be used by the programmer to define and manipulate graphs. Graphs, nodes and edges have *default* attributes, which are part of the API. For example, each graph consists of a set of *nodes* and *edges*, can be directed, and may have a root node. Each node has a set of neighbors and of (outgoing and incoming) edges, and a *level*. Each edge has a *left* and *right* vertex and potentially a *weight*. These basics data structures can be extended via user-defined, *application-specific* attributes. Such attributes can be

```
GRAPH API
graph/node/edge
Default attributes
graph: nodes, edges, root, num_nodes, num_edges, directed
node: (in_/out_)neighbors, (in_/out_)edges, (in/out)degree,
      level
edge: left, right, weight; primitive: node mate(node)
Methods to define/manipulate application-specific attributes
void addAttr(graph/node/edge, attr_name, type, default_value);
void setAttr(attr_name, value);
value getAttr(attr_name);

CONTAINER DATA STRUCTURES
set: void add(item), void remove(item), bool include(item),
     bool empty(), int size(), void clear(), bool equal(set)
oset: primitives of set; item first(), item next(item)
multiset, omultiset: primitives of set/oset, int occurrences(item)
queue: void push(item), item pop(), item front(), int size(),
       bool empty(), item next(item), void clear()

ITERATORS
sequential:
   while(condition [; dynamic_update(set)])
   for(datatype item:domain [; dynamic_update(set)])(filter)
parallel:
   foreach(datatype item:domain [; clear domain])(filter)
   inBFS(var: domain from source_node)

DYNAMIC MEMORY MANAGEMENT PRIMITIVES
newGraph
addNode/deleteNode
addEdge/addDirectEdge/deleteEdge
new/delete

PARALLEL PRIMITIVES
item reduction(container, operator)
void scan(in_container, out_container, operator)
void sort(in_container, out_container)

SYNCHRONIZATION PRIMITIVES
barrier
critical{}

RUNTIME PRIMITIVES
void commit(bool) - commits a set of changes to the graph and,
                    if parameter is true, to the working set
void rebalance() - rebalance an extended CSR representation
```

Figure 17: Graph programming API

defined using the *addAttr* primitive, and their value can be set and queried using the *setAttr* and *getAttr* methods, respectively.

Container data structures: A variety of containers (*unordered* and *ordered sets* and *multisets*, and *queues*) can be used to build graph algorithms (for example, to represent

the working set). These containers come with a set of access and manipulation primitives (*add*, *remove*, *empty*, etc.) and can apply to generic objects. Internally, containers operate on numeric data types and pointers to objects and are mapped to platform-specific, thread-safe data structures.

Iterators: Iterators provide the ability to define loops. They can be of two kinds: sequential (*for* and *while*) and parallel (*foreach* and *inBFS*). Parallel iterators allow the programmer to expose parallelism within the application. The *inBFS* iterator (from Green-Marl [69]) modifies the *level* attribute of the nodes.

Dynamic memory management primitives: Dynamic memory allocation primitives can be graph-specific (e.g., *newGraph*, *add*/*deleteNode*, *add*/*deleteEdge*) or general-purpose (e.g., *new* and *delete*). The former map to the internal, optimized graph representation; the latter can be used for containers or application-specific, user-defined data structures. All these primitives are internally mapped to a *custom_malloc* function and handled within our runtime system.

Parallel primitives: Commonly used parallel primitives on containers (*reduction*, *scan* and *sort*) are internally mapped to platform-specific, optimized, thread-safe implementations.

Synchronization primitives: The source-to-source compiler automatically handles synchronization related to the graph data structure, containers and iterators. However, application-specific, user-defined data structures may also require synchronized access. This kind of synchronization must be explicitly indicated by the programmer through high-level primitives (*barrier* and *critical*). These primitives are internally mapped to platform-specific synchronization mechanisms.

Runtime primitives: Runtime primitives can be invoked when the graph data structure is modified by external intervention. Specifically, *commit* is used to commit a set of graph modifications to the runtime system; after a *commit*, the required modifications are applied to the internal graph and possibly incorporated in the working set. The rebalance primitive allows re-optimizing the internal layout of the graph data structure.

4.2.3 Case Study

We consider two kinds of applications: graph analytics and general purpose graph processing. Specifically, we target three categories of workloads.

Graph analytics on static datasets: These *read-only* algorithms perform analysis tasks on graphs that do not change over time or that change so infrequently to justify rerunning the algorithm on the whole graph when this happens.

Graph analytics on dynamic datasets: These *read-only* algorithms perform analyses on graphs that may change over time. The graph itself is not modified by the algorithm, but by external events. For instance, in social network, graphs constantly change due to "friend" and "unfriend" acticities.

General purpose graph processing: These *read-write* algorithms perform various types of general-purpose computations that may modify the structure of the underlying graph. For example, subset construction [103], which transforms a non-deterministic finite automaton (NFA) into a deterministic one (DFA), is a general purpose read-write algorithm.

```
1.  Procedure A-DFA (graph dfa, int d){
2.    inBFS(n: dfa.nodes from dfa.root){
3.      int max = 1;
4.      n.setAttr('default', UNDEFINED);
5.      foreach(node p:dfa.nodes)
6.        if (p.level < n.level & n.level-p.level ≤ d){
7.          int cn = common_neighbors(n,p);
8.          if (cn > max){
9.            max = cn;
10.           n.setAttr('default', p);
11. } } } } }
```

Figure 18: A-DFA compression algorithm

We briefly describe two algorithms used as example workloads and illustrate our programming API on them.

A-DFA: A-DFA [104] is a compression algorithm used to reduce the memory footprint of DFA accepting large sets of regular expressions. For simplicity, in this paper we focus on the computation of the application-specific *default* transition attribute. Thus, the A-DFA algorithm shown in Figure 18 is *read-only* and operates on a static graph. Specifically, the algorithm visits a DFA graph in BFS manner (line 2). It compares each node with every other node at lower depth (skipping *d* levels) and selects the node with more transition commonality as default target state. Compared to BFS, A-DFA presents a non-trivial computation phase. In fact, the work executed at every step of the BFS traversal (lines 3-11) includes control-flow operations and scattered memory accesses to the whole DFA graph. We also notice that this algorithm exhibits a two-level parallelism (*inBFS* at line 2 and *foreach* at line 5).

```
1.  Procedure PageRank(graph g, double delta){
2.    for(set ws=g.nodes; !ws.empty(); dynamic_update(ws)){
3.      foreach(node n: ws){
4.        n.setAttr('nr',0);
5.        for(node m: n.in_neighbors)
6.          n.incAttr('nr', m.getAttr('rank')/m.outdegree);
7.      }
8.      foreach(node n: ws; clear ws){
9.        if (abs(n.getAttr('nr')-n.getAttr(rank))>delta)
10.         ws.add(n.out_neighbors);
11.       n.setAttr('rank',n.getAttr('nr'));
12. } } }
```

Figure 19: PageRank algorithm

PageRank: PageRank (Figure 19) is commonly used in search engines to rank webpages based on their significance. The *rank* attributes are initialized to a default value upon creation. In each iteration of the main loop (line 2), the ranks are updated according to the *outdegree* of connected nodes. Specifically, every node distributes its rank evenly to its outgoing neighbors (lines 5-6). PageRank terminates after all ranks converge (i.e., their variation falls below a given threshold *delta*). PageRank is a *read-only* algorithm that can operate on either static or dynamic graphs since webpages can be created and modified by user intervention. The user notifies the runtime system by issuing a sequence of *addNode* and *addEdge* (and corresponding *delete* operations) followed by a *commit*. The *commit* causes the involved nodes to be added to the working set *ws*. External modifications to *ws* are processed in the next iteration. The *dynamic_update* primitive in the iterator at line 2 indicates that external updates should be incorporated in *ws* at the beginning of each iteration. This can happen while PageRank is running or when it is terminated (in which case it will be reactivated). We note that the ranks are usually double precision floating point numbers, thus requiring the use of floating point arithmetic (on GPUs, floating point is slower than integer arithmetic).

DFA construction: DFA are typically used by applications performing regular expression matching. In this context, regular expressions are initially compiled into a NFA. Then, the NFA is transformed into DFA through subset construction [103] (Figure 20). Like A-DFA, DFA construction proceeds in BFS manner and exhibits a two-level parallelism (*inBFS* at line 5 and *foreach* at line 7). Again, the work performed in each iteration (body of *inBFS* loop at lines 6-20) contains control-flow operations and irregular memory accesses. However, DFA construction involves an additional complexity: it modifies the DFA graph (in fact, it creates it). As can be seen, DFA construction involves dynamic memory operations on the DFA graph (lines 2, 4, 14 and 18), on sets (lines 4, 8 and 17) and on the custom *subset* data structure (lines 3, 4 and 14). The latter has a complexity hidden within its lookup and update primitives. Briefly, *subset* is a double linked-list data structure used to verify in linear time if a subset belongs to a power set.

```
1.  Procedure DFA_construction (graph nfa, graph dfa){
2.    dfa = newGraph();
3.    subset mapping = new subset();
4.    dfa.root = dfa.addNode(mapping.update(new set(nfa.root)));
5.    inBFS(n: dfa.nodes from dfa.root){
6.      set *nfa_subset = mapping.reverse_lookup(n);
7.      foreach(char c: alphabet){
8.        set *target = new set();
9.        for(node m: nfa_subset)
10.         for(edge e: m.out_edges)
11.           if (e.getAttr('symbol') = c) target.add(e.mate(m));
12.       if (!mapping.lookup(target)){
13.         critical{
14.           node state = dfa.addNode(mapping.update(target));
15.       } }
16.       else
17.         delete target;
18.       edge ne =dfa.addDirectEdge(n,state);
19.       ne.setAttr('symbol',c);
20. } } }
```

Figure 20: DFA (subset) construction algorithm

The programmer can make the code parallelizable by using a *critical* section (lines 13-15): this is an example of *coarse-grained synchronization*. In alternative, the programmer can provide a thread-safe implementation of the *subset* data structure optimized for different platforms (thus allowing custom *fine-grained synchronization*).

4.2.4 Compiler Design

Once graph algorithms are expressed by using our platform-agnostic API, our source-to-source compiler generates different code variants for GPU, Intel Phi, and multi-core CPU. We describe the main aspects of the compilation process.

A. Data Structure Design

Our reference graph encoding scheme is the *compressed sparse row form* (CSR), a format commonly used to represent sparse data structures [75]. To support dynamic graphs, we extend the basic CSR data structure into a hierarchical-CSR (Figure 20). Initially, we overprovision both the level-1 *nodes* and *edges* arrays. In the edges array, we pre-allocate a default number of edges per node (blank slots in the level-1 *edges* array in Figure 20). The optimal initial provisioning size depends on the characteristics of the graph. Reserving larger space can improve the insertion performance at the cost of wasting memory. At each deletion, we invalidate elements. Upon insertion, we first use the free elements of the *nodes* and *edges* arrays. When the level-1 *nodes* array overflows, we allocate a level-2 *nodes* array. Similarly, when the portion of the *edges* array allocated to a node *n* overflows, we allocate a block of cells in a level-2 *edges* array and insert the new edges of *n* in it. We repeat this operation recursively: if the level-2 edges/nodes array overflows, a level-3 array is allocated. For each node, one extra variable is needed to

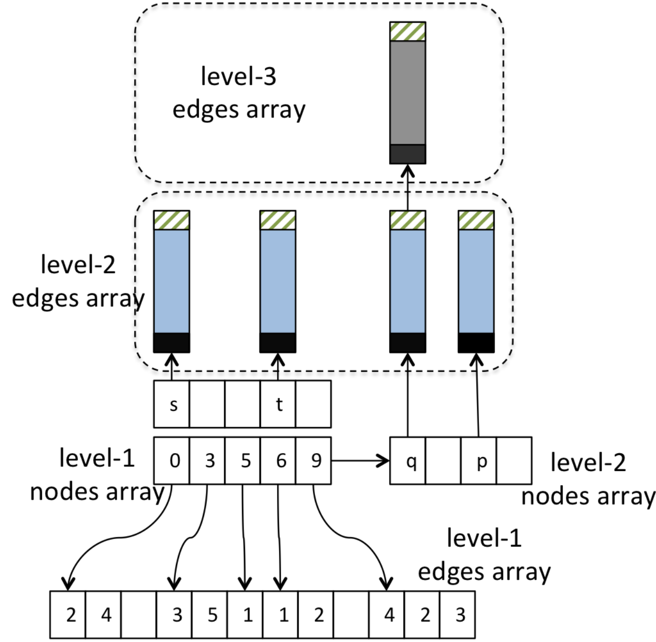


Figure 21: Hierarchical-CSR with level-2, level-3 nodes/edges arrays

store the "pointer" to the corresponding block within the level-2 edge array (s, t, p, q in Figure 21). In addition, the last element of each level-x block (the dashed area in Figure 21) stores a "pointer" to the *level-(x+1)* block (null-pointer if such block has not been allocated). Also, for fast insertion into any level-*x* block, the first element (solid black area in Figure 21) records the used space in the block. We fix the size of the level-2 blocks (on GPU, for example, 128B blocks allow aligned memory accesses to entire blocks), and increase the block-size quadratically from level to level (to take into account the fact that commonly used small-world networks present only a few nodes with a substantial number of neighbors). The resulting hierarchical-CSR allows efficient dynamic insertions and deletions but, due to its nested structure, is less efficient to traverse than a flat. Our runtime system periodically calls the *rebalance* primitive to transform the hierarchical-CSR into pure CSR form. The allocation process relies on the dynamic memory management mechanism provided by our runtime (Section VI.A),

which fosters data locality. For container data structures, we use basic blocks proposed and discussed in previous work [75, 77, 97].

B. Code Generation

We now describe our code generation process along with the platform-specific transformations we perform.

General Design – Users encode functions corresponding to the "hot spots" in their applications using our programming API. Our source-to-source compiler then transforms these user-defined functions into C++ wrapper functions containing platform-specific code. In the case of multi-core CPUs, code regions associated with parallel iterators are translated into parallel regions through the insertion of OpenMP directives. GPUs and Intel Phi coprocessors require handling also the data transfers between host and device. In particular, the compiler generates data transfers for graphs and user-defined data structures declared outside the parallel regions and referenced inside them. In the case of the Intel Phi, parallel regions are handled using OpenMP directives and placed inside offload regions surrounded by the "*#pragma offload*" directive. The primitives to allocate variables on the coprocessor and move data between host and device are inserted along with the offload directives. In the case of GPUs, regions of code associated to parallel iterators are translated into parallel kernels. In this process, elements of the working set are mapped onto threads or thread-blocks (i.e., thread- and block-based mapping [102]) producing different code variants. The compiler then generates code for data transfers (using the *cudaMalloc* and *cudaMemcpy* primitives) and kernel launches.

Handling of Nested Parallel Iterators – The presence of nested parallel iterators enables different code variants and can be handled differently on various accelerators.

The Intel Phi has a flat hardware parallelism. However, the two-level nesting of A-DFA and DFA construction leads to different alternatives on the placement of the offload and OpenMP parallel directives. Specifically, we can: (i) place a synchronous offload at the level of the sequential for and the parallel directive at the level of the outer *inBFS* iterator; (ii) place both the synchronous offload and the parallel directive at the level of the outer *inBFS* iterator; (iii) use an asynchronous offload and a parallel directive on one of the parallel iterators (thus launching several parallel offloads concurrently). We experimentally found that the offload overhead makes the first code variant preferable.

NVIDIA GPUs present a two-level hardware parallelism. Therefore, two-level nested iterators (as in 18 and 20) can be naturally handled by using block- and thread-based mapping for the outer and inner parallel iterators, respectively. On the other hand, using thread-based mapping on the outer-loop will cause serialization of the inner-loop. In the presence of three nested parallel iterators, an additional level of parallelism can be achieved by using multiple streams and the parallel kernel execution feature available starting from the Fermi architecture [105]. Given its massive hardware parallelism and its Hyper-Q technology, code variants using concurrent streams are particularly suited to Kepler GPUs. Kepler GPUs also allow an additional level of nesting through dynamic parallelism. However, we experimentally found that the overhead for launching nested kernels is significant, making it difficult to achieve good performance by using this feature.

Other Accelerator-specific Optimizations – To reduce the communication overhead, we leverage the compiler analysis techniques proposed by [106] to identify data reused by subsequent parallel kernels with no intermediate CPU read or write access. These data

can be stored persistently on the coprocessor, thus avoiding unnecessary data transfers between host and device. On the Intel Phi, we use the *alloc_if* and *free_if* clauses to control the allocation of data and the *in*, *out*, and *nocopy* modifiers to avoid unnecessary data transfers.

On the Intel Phi, the use of vectorization can greatly help performance. Vectorizable code can come from either *foreach* or *for* iterators. In the PageRank algorithm (Figure 19), for example, the for loop at line 6 is a good candidate for vectorization. In the case of for loops, however, data dependences must be resolved (since these iterators are sequential). To this end, we rely on a feature of the Intel compiler: inserting a "*#pragma ivdep*" before a loop allows the Intel compiler to resolve conservatively assumed data dependences and possibly generate vectorized code.

4.2.5 Runtime Library Design

The runtime system has essentially two functions (Figure 16): dynamically selecting and tuning the code variant that better fits the characteristics of the target dataset and the hardware profile, and handling dynamic memory allocation. Variant selection can be done based on profiling information and by monitoring the size of the working set, and dynamically selecting the code variant that better fits that level of parallelism [102]. Due to space constraint, in this paper we only describe dynamic memory management

A. Dynamic Memory Management

NVIDIA GPUs lack of operating system support and of an efficient mechanism to handle dynamic memory allocation within parallel kernels. Starting from the Fermi Architecture NVIDIA has added support for the *malloc* call. However, the use of *malloc* on GPU has

two limitations: first, it leads to inefficient code for small size allocation; and second, it fails in the presence of large numbers of *malloc*. We observed that, when using system *malloc*, DFA construction fails even on small graphs (about 30k nodes). To circumvent this problem, we have introduced a custom memory management scheme for GPU. We have ported this mechanism to multicore CPU and Intel Phi, and have compared our proposed scheme with the direct use of the system *malloc*.

General design: Our basic idea is to use a memory pool with fixed-size blocks. Specifically, we start by pre-allocating a single block with a handle pointing to the next free position within the block. Each *custom_malloc* call will obtain the requested number of bytes from the block, and will cause the handle to be incremented accordingly. When the current block fills up, a new block is allocated by the runtime.

Our proposed solution uses multiple locks (one per thread-group) to reduce lock contention. We use two types of blocks (with separate handles): *permanent* and *temporary* blocks. Permanent blocks have the application lifetime, whereas temporary ones have the lifetime of an iteration of the algorithm. Variable de-allocations within the temporary blocks are deferred until the end of the corresponding iteration, when temporary blocks are cleared by resetting their handle. Compile-time code analysis determines which kind of *custom_malloc* to perform for each dynamically allocated variable and the runtime moves temporary data to permanent blocks when needed. This significantly reduces fragmentation and de-allocation cost, which distinguishes our dynamic memory management from other proposals.

Intel Phi/CPU implementation: Since the Intel Phi and the CPU have a flat thread organization, thread grouping is done based on the thread identifiers. Our experimental

results show that on CPU the system *malloc* and *custom_malloc* have similar performance. On the Intel Phi, however, the system *malloc* outperforms the *custom_malloc* (independent of the number of regions). We experimentally verified (by forcing locks also on system *malloc* calls) that this inefficiency is not due to the synchronization overhead. We believe that pre-allocating a large memory buffer on the Intel Phi may affect the performance in two ways. First, the Intel Phi operating system performs lazy memory allocation (i.e., it progressively allocates memory as needed). This mechanism may make large allocations costly. Second, large pre-allocations may have bad interference with caching.

GPU implementation: The GPU implementation of the above memory management mechanism requires addressing two issues: using a deadlock-free locking mechanism and reducing the synchronization cost. Ramamurthy [105] pointed out that, due to the SIMT architecture and the unfairness of GPU warp schedulers, common spin-lock implementations based on atomic *compare_and_swap* may cause deadlock. To avoid this problem, we use the deadlock-free implementation described in [105]. To reduce the synchronization cost on GPU, we associate a buffer region to each thread-block, thus limiting contention to threads belonging to the same block. This design also allows storing handles in shared memory for fast access. However, atomics on shared memory generally result in serialization and are costly [107]. We experimental verified that storing these handles in global memory leads to better performances. Finally, we note that the blocks are stored in global memory, and can therefore be accessed by all threads. In summary, each thread-block can *allocate* data only in its assigned regions, but can access data located in regions mapped to other thread-blocks.

4.2.6 Performance Evaluation

In this section we evaluate the performance of the code generated by GRapid on three platforms: an 8-core Xeon E5-2609 CPU, an NVIDIA Tesla C2075 GPU, and an Intel Xeon Phi 5110P. This process will provide guidelines on the mapping of applications onto these platforms. We compiled CPU and Phi code through the Intel C++ compiler (icpc 13.1.2) and used the CUDA 5 toolkit to compile and run the GPU code. Data transfer times are included in the coprocessor results. In all cases, we show the speedup reported by the parallel code over a serial code running on a Xeon E5 processor.

To cover all the categories of graph applications, we implemented BFS, A-DFA, PageRank and DFA Construction using GRapid. For BFS and PageRank, we used datasets from DIMACS competitions and Stanford-Large-Data Collection (see [102] for more detail). The largest graph has 4M nodes and 34.5M edges. For A-DFA and DFA construction, we use DFA graphs with 30k-700k nodes and 70M-180M edges (typical sizes in regex matching paper [104]). Due to limited space, we omit a detailed discussion of the results reported on BFS.

A. A-DFA

Figure 22 and 23 report the performance of A-DFA. The datasets are DFAs with number of states varying from 30k to 400k, as reported on the x-axes. We recall that A-DFA makes the default transition of each DFA state n point backward to the state that has the highest number of transitions in common with n. The distance parameter d affects the amount of work (lines 3-10 of Figure 18): for larger d, the algorithm performs more state comparisons (and memory accesses). In the GPU case, we use three of the code variants

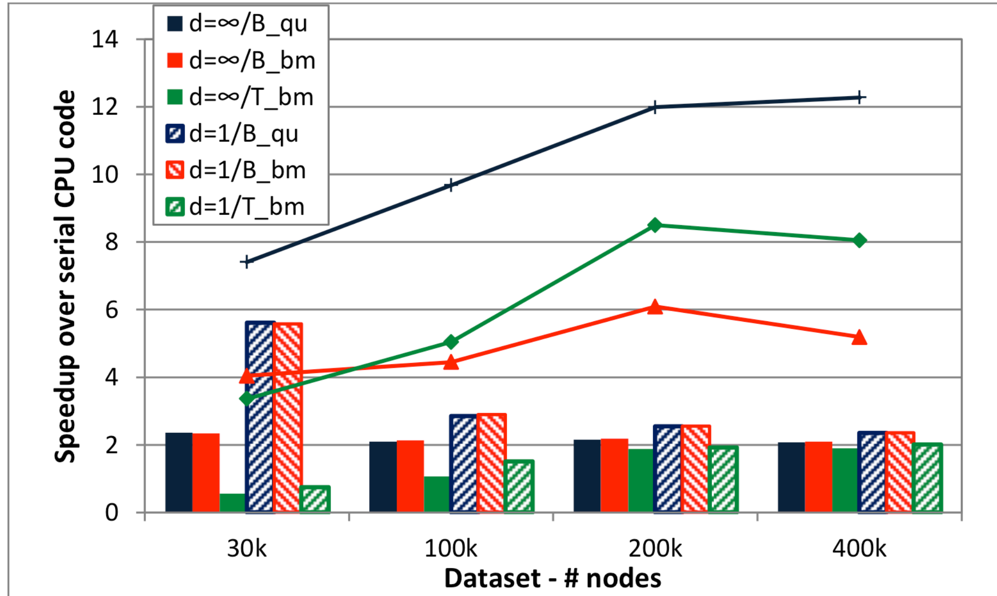


Figure 22: A-DFA compression –Speedup of GPU over serial CPU implementation

and also show the performance reported by BFS. In the Intel Phi case, we use a bitmap-based working set and a variable number of threads.

We note the following observations. First, due to the irregularity of the work performed, the speedup of A-DFA on GPU is substantially inferior to that of BFS using the same code variants. Second, the GPU speedup of A-DFA decreases when the amount of work increases (that is, for larger d and datasets). Third, the relative performance of the code variants differs between BFS and A-DFA. Fourth, due to the more general-purpose nature of its hardware, the behavior of the Intel Phi differs from the GPU. In fact, the speedup of A-DFA on the Intel Phi is far greater than that of BFS and increases with the amount of work (that is, with increasing *d*) from 9~15x to 43~67x. On the other hand, the speedup does not scale with the dataset size. In fact, larger graphs put more pressure on the cache, thereby limiting the performance. In addition, in contrast to BFS, A-DFA can effectively leverage all 60 cores available on the Intel device: the performance scales almost linearly until 120 threads (e.g., two threads per core). However, using all four

hardware threads in each core does not provide further benefits. Finally, due to the complexity of the work, the multi-threaded CPU implementation of A-DFA achieves ideal speedup: roughly 8x on 8 cores.

B. PageRank

Figure 24 shows the speedup (and, for values < 0, the slowdown) of PageRank on dynamic datasets when using our hierarchical-CSR over performing a full CSR rebuild. We evaluate different methods for initial overprovisioning of the level-1 arrays: zero (0) *provision* (reserve no extra space), *even provision* (reserve the same amount of extra space to all nodes) and *ratio provision* (reserve different amounts of extra space to different nodes according to their outdegree). The experiments are conducted on the Google weblink graph [102], which consists of 0.7M nodes and 2.5M edges. In case of even overprovisioning, we set the extra allocation to half of the average node outdegree. In case of proportional overprovisioning, we overprovision the edge array so that each

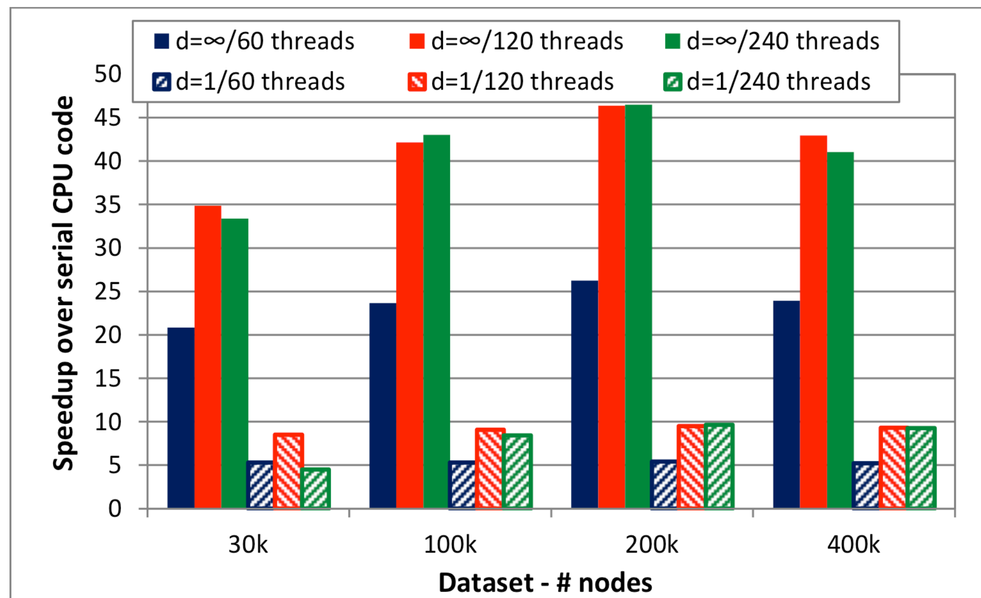


Figure 23: A-DFA compression – Speedup of Xeon Phi over serial CPU implementation.

node is allocated 150% of the space required by its outdegree. We dynamically add an increasing number of nodes and edges (evenly distributed among nodes) to the original graph (x-axis). While doing so, we keep the average outdegree unchanged: the number of added edges is roughly 7 times that of added nodes. In the hierarchical-CSR implementation, the PageRank kernel is slightly more complex because it must handle the hierarchical graph data structure. However, the static approach requires a full rebuild of the CSR representation and, consequently, large data transfers between the CPU and the coprocessor. As can be seen, since the amount of added nodes is small (< 3.5% compared to whole graph), the incremental, hierarchical approach is preferable to a full rebuild on both CPU and GPU for most of the cases. On CPU, even overprovisioning achieves the best speedup. On GPU, however, the performance of the 0 provision and even provision methods are very close. Ratio provision introduces blank slots in the

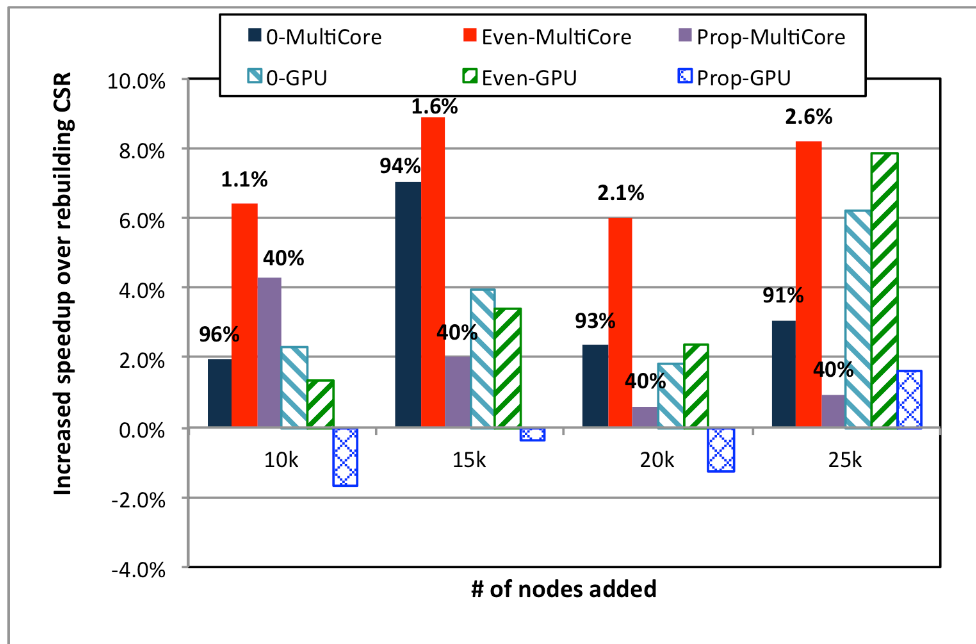


Figure 24: PageRank on dynamic graphs with hierarchical CSR

level-1 arrays, leading to underutilization of the GPU hardware. By using more extra-space, ratio provision leads to the worst performance on GPU. We don't show the performance of the Intel Phi, because it is poor (always 10%~20% slow down). We believe that the memory allocation mechanisms within the Intel Phi's OS may interfere with our dynamic memory management scheme and cause this performance loss. We need to get more insights on the operation of the Intel Phi OS to improve the performance of our memory management scheme.

C. DFA Construction

Figure 25 shows the speedup of DFA construction on multi-core CPU, GPU and Intel Phi over a serial CPU implementation. We tested two code variants: one using coarse-grained and the other using fine-grained synchronization. We recall that the coarse-grained code variant has a critical section around the *subset.update* primitive, while the fine-grained version includes a thread-safe subset implementation that associates a fine-grained lock

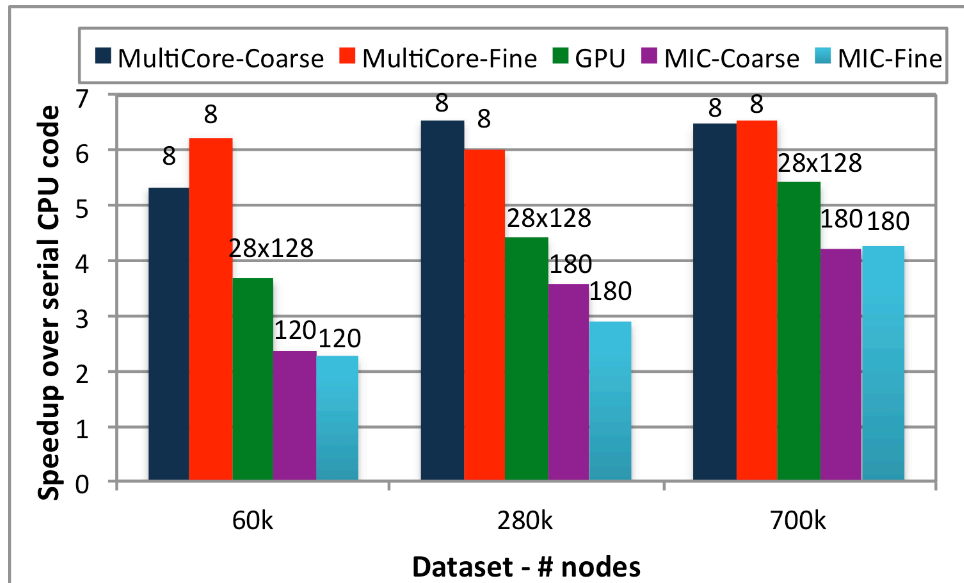


Figure 25: DFA construction – Speedup of multi-core CPU, GPU and Intel Phi over serial CPU code

to each node of the double linked-list. On GPU, we use block-based mapping. In the kernel configuration, we set the number of blocks to twice the number of SM on the GPU and the block size to 128 threads. For the Intel Phi, we show the thread configuration reporting the best performance (the corresponding number of threads is indicated on top of the bars). We test DFA construction on four datasets with increasing size (x-axis). The multi-core CPU and the Intel Phi report the best and worst performance, respectively. Interestingly, the two code variants show comparable performance on all platforms (recall that the coarse-grained version uses the high-level *critical* primitive and requires minimal programming effort). The GPU and Intel Phi are consistently slower than the 8-threaded CPU code; the performance gap between the multi-core CPU and the many-core devices, however, decreases as the dataset size (and the runtime parallelism) increases. The GPU performance suffers from the irregular memory access patterns, cost of locking and branch divergence. The modest performance of the Intel Phi is in part due to the less efficient memory management module within the thin-OS running on this coprocessor.

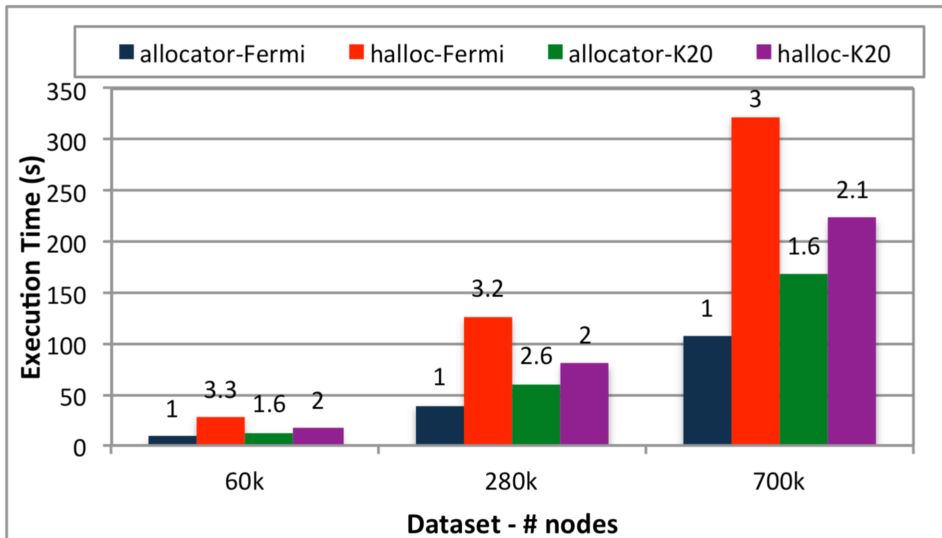


Figure 26: DFA construction using different allocators and GPUs

We compare our memory allocator with Halloc [108], a state-of-the-art dynamic memory allocator for NVIDIA Kepler GPUs. For completeness, we also port Halloc to Fermi GPUs. To this end, we remove the "__*shfl*" instructions from its code. This instruction allows distributing the register values from one thread to the other threads within the same warp, but is available only on Kepler GPUs. On Fermi GPUs, we use shared memory to exchange the value of register variables among the threads within a warp. This workaround, however, requires additional synchronization instructions, leading to performance degradation compared to the original version relying on the "__*shfl*" instruction. Figure 26 shows the performance comparison between our allocator and Halloc for DFA construction on Fermi (C2075) and Kepler (K20) GPUs. As can be seen, on Fermi GPUs our allocator leads roughly to a 3x speedup over Halloc. In addition, our allocator achieves a 21%, 25% and 28% performance improvement over Halloc on the 60k, 280k and 700k node DFA, respectively. This performance improvement can be explained as follows. Recall that our allocator uses two distinct memory pools – a permanent and a temporary one – and leverages compile-time analysis to determine where to perform allocations. This design decreases the deallocation cost and increases the data locality of temporary data.

D. Summary

Table 2 summarizes the characteristics of the four applications (BFS, A-DFA, PageRank, DFA Construction) and the speedup reported on multi- and many-core platforms. In the 2nd column we indicate whether the graph topology is static or dynamic, and, in the latter case, whether it is modified by the application (*read-write* applications) or by external intervention (*read-only* applications). The 3rd column shows the application-specific

Table 2: Summary of applications and speedup over serial code

Application	Application & Graph Type	Attributes	Comp. pattern & arith. intensity	Best Speedup		
				m-CPU	GPU	Phi
BFS	read-only static	Level (int)	Set level (simple& low)	2.5x	50x	3.5x
PageRank	read-only static/dynamic	Rank (double)	Calculate Rank (intermediate & intermediate)	6.3x	6.5x	9x
A-DFA	read-only static	Default Trans (int)	Compare trans. (complex, low)	8x	6x	45x
DFA construction	read-write dynamic	Trans Table (int)	Compare Subset (complex, low)	6.5x	5.5x	4.3x

attributes. The 4th column reports an indication of the complexity of the parallel work and its arithmetic intensity. As can be seen, many-core platforms outperform multi-core CPUs on static datasets. The Intel Phi is preferable to GPU for more complex computational patterns, whereas the arithmetic intensity is not a big discriminating factor. The three platforms report similar speedups on DFA construction; however, the multi-core CPU is in this case a slightly better choice, due to the presence of frequent dynamic memory allocation and synchronization. We note that manually generating code for the three considered devices would be a daunting and time consuming task: by automatically generating different versions of the code, GRapid allows the programmer to quickly identify the platform most suited to his application.

4.3 Adaptive Computing

It has been shown that graphs used in real-world applications [109-111] exhibit significant topological differences. The topology of the graphs dictates the amount of parallelism that can be extracted at runtime, thus affecting the performance of specific GPU implementations. This heterogeneity makes it difficult to design a GPU implementation of a graph algorithm that is optimal on a large variety of datasets. In this

work, we argue for an adaptive solution that takes into account the topological characteristics of the dataset to dynamically select the most suitable alternative among a set of available GPU implementations.

4.3.1 Motivation

In this section, we present some motivating facts that show the potential of GPUs as accelerators of graph algorithms and give an intuition of why a dynamic solution can be preferable to a static one. First, we characterize some graph datasets used in real-world applications. Second, we introduce the main architectural features of modern GPUs, and discuss their suitability to the deployment of graph algorithms.

A. Characterization of Graph Datasets

Graphs are a powerful representation used in many practical applications, where the relationships among the nodes in some network are relevant. Some examples drawn from different application domains are: the road network, the web link network and the social network. The road network is typically extracted from GPS maps and used to calculate the optimal route (or shortest path) between two endpoints. The web link network contains links between web pages, and its connectivity is typically used by search algorithms to rank the results of queries. The social network contains relationships between individuals, and is used to compute a variety of connectivity properties (in applications like Facebook, for instance, such relationships are used to suggest new friends).

We use graphs from the 9th and the 10th DIMACS implementation challenges [109, 110] and from the Stanford Large Data Collection [111]. In particular, we consider

datasets used in different application domains: the Colorado road network [109], a paper co-citation network (from the CiteSeer library) [110], a p2p networking network [111], the Amazon co-purchase network [111], the Google webpage link network [111] and a SNS network (from Live-Journal) [111]. All but the road network and the paper co-citation network are directed graphs. Table 3 shows a characterization of these datasets, in terms of total number of nodes, total number of edges and node degree (that is, number of edges per node). We observe the following facts.

- The graph size varies considerably across the datasets: from the small *p2p* network (with about 36.6 K nodes and 183.8 K edges) to the large *SNS* network (with about 4.3 M nodes and 34.5 M edges).
- The average node outdegree also varies considerably: from 2.4 in the *CO-road* network, to 73.9 in the *CiteSeer* network. Four networks (*CiteSeer*, *p2p*, *Google* and *SNS*) exhibit a considerable outdegree variance, leading to large outdegree values. The other networks (*CO-road* and *Amazon*) have a more regular structure.

Figure 27 shows the outdegree distribution of the *CO-road*, the *Amazon* and the *CiteSeer* network. As can be seen, these networks exhibit different characteristics. The *CO-road* graph is pretty sparse: most of its nodes have an outdegree from 1 to 4, and the maximum outdegree is 8. This is because most towns are usually directly connected to a

Table 3: Dataset characterization.

Network	# Nodes	# Edges	Node Outdegree		
			min	max	avg
CO-road	435,666	~1 M	1	8	2.4
CiteSeer	434,102	~16 M	1	1,188	73.9
p2p	36,692	~0.18 M	0	1,383	10.0
Amazon	396,803	~1.7M	0	10	8.4
Google	739,454	~2.5 M	0	456	6.9
SNS	4,308,452	~34.5 M	0	20,293	16.0

handful of other towns, whereas few bigger cities serving as transportation hubs have as many as 7-8 intercity roads. The Amazon network is very regular: 70% of the nodes have 10 outgoing edges, and the remaining nodes have an outdegree uniformly distributed between 1 and 9. The *CiteSeer* network is far less regular: about 90% of the nodes have less than 200 outgoing edges. On the other hand, the outdegree range is very wide for the remaining nodes (up to 1,188). The outdegree distribution of the *p2p*, the *Google* and the *SNS* networks is similar to that of the *CiteSeer* graph.

This fact has a practical significance. Most graph algorithms proceed iteratively. In each iteration, they visit the local neighborhood of a working set consisting of nodes or edges, remove elements from the set and add new elements to it. Intuitively, large outdegree lead to large working sets, and thus to potentially high amounts of parallelism. However, unbalanced outdegree distributions can cause work imbalances during graph traversals. An adaptive solution may therefore better support a wide variety of graphs, including those with irregular topologies.

B. Imbalanced Work of Graph Algorithms

In this work, we focus on breadth-first search (BFS) and single-source shortest path (SSSP), two fundamental graph problems. BFS computes the depth of each node n, that is, the minimum number of nodes visited when moving from a given source node to n. SSSP computes the minimum cost paths from a given source node to any other node in the

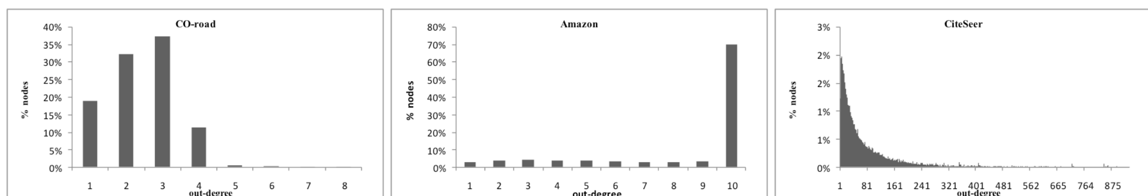


Figure 27: Outdegree distributions of *CO-road*, *Amazon* and *CiteSeer* networks

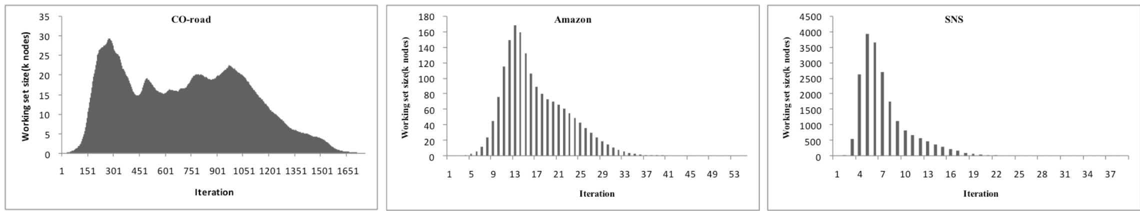


Figure 28: Unordered SSSP – size of the working set during the execution (*CO-road*, *Amazon* and *CiteSeer* networks)

graph. These problems are solved through an iterative graph traversal. Initially, the working set consists of the source node. In each traversal step the local neighborhood of the working set is processed. The traversal terminates when the working set becomes empty. During execution, two kinds of work imbalance can take place.

- *Inter-iteration work imbalance* - The size of the working set typically changes from iteration to iteration. For example, Figure 28 shows how the size of the working set varies during the execution of SSSP on three datasets (*CO-road*, *Amazon* and *SNS*). As can be seen, the work is generally limited at initial stages, when the traversal is restricted to the neighborhood of the source node. When enough nodes have been processed, the working set starts growing and keeps growing until a large fraction of the nodes have been visited. At that point, the working set starts shrinking. The working set size and the convergence speed depend on the specific algorithm and on the characteristics of the dataset. For instance, on the datasets of Figure 28, BFS has working set sizes from 2 to 20 times smaller than those reported by SSSP.
- *Intra-iteration work imbalance* - Different nodes can have different outdegrees. As a consequence, each node in the working set can be potentially associated with a different amount of work. This fact affects the performance of the GPU design.

For example, if a node-to-thread mapping is adopted on a graph with an irregular topology, thread divergence may arise during execution, and the performance will be limited by the node with the largest outdegree.

C. Architectural pros & cons of GPUs

GPUs are known for the massive hardware parallelism that they offer. NVIDIA GPUs consist of several SIMT processors, called Streaming Multiprocessors (SMs), each containing a set of in-order CUDA cores. In the Fermi architecture, each SM comprises either 32 or 48 cores (depending on the compute capability of the device). The CUDA programming model [112] facilitates writing parallel algorithms for GPUs. In CUDA, the computation is organized in a hierarchical fashion: threads are grouped into thread-blocks; at runtime, each thread is mapped onto a core and each thread-block is mapped onto a SM. In CUDA 4, as many as 64K*64K*64K blocks with at most 1,024 threads each are allowed. This parallelism can clearly be advantageous for graph applications that operate on large datasets consisting of millions of nodes and edges.

Another characteristic of the GPU architecture is its memory hierarchy. GPUs are equipped with a relatively large off-chip, high-latency, read-write global memory; a smaller low-latency, read-only constant memory (which is off-chip but cached); and a limited on-chip, low-latency, read-write shared memory. The global memory can be accessed via 32-, 64- or 128-byte transactions and has a high access bandwidth (up to 144 GB/sec). Multiple memory accesses to contiguous memory locations are automatically coalesced into a single memory transaction, thus saving memory bandwidth. The graph algorithms in consideration are not computation intensive, but – especially when running on large datasets - can be memory bound. In fact, when processing hundreds of nodes in

parallel, it is necessary to access their neighbors in an efficient way. The GPU high memory bandwidth can be beneficial for these memory intensive applications.

Two GPU architectural features are particularly problematic when deploying graph algorithms on GPUs. First, SMs are SIMT-processor. During execution, threads are grouped into 32-element SIMT units, called warps. In every clock cycle, threads belonging to the same warp must execute the same instruction. Branches are allowed through the use of hardware masking. In the presence of branch divergence within a warp, both paths of the control flow operation are in principle executed by all CUDA cores. Therefore, the presence of branch divergence within a warp leads to core underutilization. Unfortunately, the irregular nature of graph algorithms leads to relatively frequent branch operations. Second, to fully utilize its high memory bandwidth, the GPU requires regular memory access patterns. In fact, contiguous memory accesses can be coalesced into few memory transactions when accessing global memory, and allow avoiding bank conflicts when accessing shared memory. However, the memory access patterns within graph algorithms are often irregular and hard to predict. Even if this effect can be limited by representing graphs with ad-hoc data structures (e.g. adjacency matrices in compressed sparse row form), unpredictable and irregular memory accesses cannot be fully avoided.

4.3.2 Exploration Space

In this section, we present and discuss a possible exploration space for implementing graph algorithms on GPU. Our study focuses on the BFS and SSSP problems. However, we believe that our analysis can be extended to other amorphous graph algorithms with

similar computational patterns. We consider a 3-dimensional exploration space (Figure 29), which is built according to the following questions.

- Is the working set used by the algorithm ordered or unordered?
- What is the granularity of the mapping of the work to the GPU hardware? Two obvious alternatives consist of mapping each element of the working set to a thread (fine-grained mapping) or to a thread-block (coarse-grained mapping).
- How is the working set implemented? We will consider a bitmap-based and a queue-based implementation.

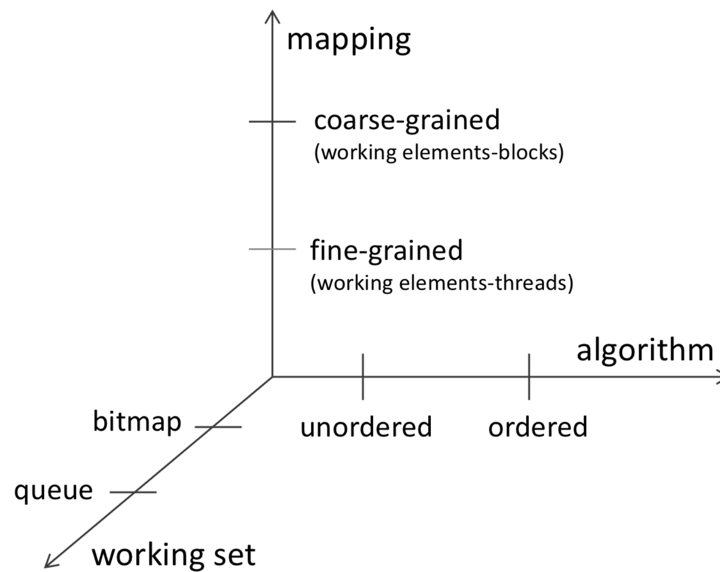


Figure 29: Exploration space

```
1.  Graph g;
2.  WorkingSet ws;
3.  g.root.level = 0;
4.  ws.add(g.root);
5.  foreach(Node n:ws){
6.     int level = n.level + 1;
7.     for (Node m: g.neighbors(n)){
8'.       if (m.level == INF){      //ORDERED VERSION
8".       if (m.level > level){     //UNORDERED VERSION
9.           m.level = level;
10.          ws.add(m);
11. }  }  }
```

Figure 30: Ordered and unordered BFS algorithms

A. Ordered vs. unordered algorithms

In this work, we consider the distinction between unordered and ordered graph algorithms introduced by Hassaan et al. [113]. The basic idea is the following. Most graph algorithms operate iteratively over a working set consisting of nodes or edges; in unordered graph algorithms, the elements can be extracted from the working set and processed in any order; conversely, in ordered algorithms, an ordering relation over the working set imposes a constraint on the processing sequence of the elements in it.

Figure 30 shows the pseudo-code of an ordered and an unordered BFS algorithm, which compute the level (or depth) of the nodes in a graph. The two algorithms differ in the nature of the working set (ordered vs. unordered, respectively) and in instruction 8. The ordered version processes each node exactly once, and adds it to the working set the first time it is visited (that is, when its level is undefined). The unordered algorithm may add the same node to the working set multiple times, as long as its level decreases when the node is visited. The ordered version clearly terminates when all nodes have been

processed. Since the node level is a monotonically decreasing function, the unordered version is also guaranteed to terminate.

Figure 31 shows the pseudo-code of an ordered and an unordered SSSP algorithm (Dijkstra and Bellman-Ford [114], respectively). In the ordered algorithm, the working set is ordered by distance, and the distance of each node is updated only once. In the unordered version, such attribute may be updated multiple times (as long as its value decreases). The ordered algorithm terminates when all node distances have been set. Since the distance is a monotonically decreasing function, the unordered algorithm is also guaranteed to terminate. Note that, in the ordered version, the same node can appear multiple times in the working set with different weight values. However, the ordered

```
1.  Graph g;              -- ORDERED VERSION
2.  WorkingSet ows;
3.  for (Node m: g.neighbors(g.root)){
4.    ows.add(<m, g.edgeWeight(g.root,m)>);
5.  }
6.  foreach(Pair <n, d>: ows)
7.    if (n.distance == INF)){
8.      n.distance = d;
9.      for (Node m: g.neighbors(n)){
10.       if (m.distance == INF){
11.         ows.add(<m, d + g.edgeWeight(n,m)>);
12. } } } }
```

```
1.  Graph g;              -- UNORDERED VERSION
2.  WorkingSet ws;
3.  ws.add(g.root);
4.  foreach(Node n: ws){
5.    for (Node m: g.neighbors(n)){
6.      int d = n.distance + g.edgeWeight(n,m);
7.      if (d < m.distance){
8.        m.distance = d;
9.        ws.add(m);
10. } } }
```

Figure 31: Ordered and unordered SSSP algorithms

nature of the working set ensures that the node distance is updated only once with the minimum weight value.

In general, ordered algorithms are more work efficient than their unordered counterparts (in that they process each element a minimum number of times), but take more iterations to converge. However, unordered algorithms may exhibit higher degrees of parallelism. In fact, unordered algorithms can process all the nodes in the working set at the same time, whereas ordered algorithms can process in parallel only elements that are equivalent in terms of the underlying order relation (in other words, at every iteration, ordered algorithms effectively process only a subset of the working set). Intuitively, ordered algorithms are better candidates for serial implementation, whereas unordered ones can be more suitable for parallel implementation.

B. Coarse- vs. fine-grained mapping

The second dimension of exploration has to do with the mapping granularity of the work to the GPU hardware. As mentioned, a graph algorithm can be expressed as a sequence of iterations over a working set. In each iteration, a subset of elements are extracted from the working set, they are processed (e.g., their depth level or their distance from a source node is computed), their neighborhood is queried, and possibly new elements are added to the working set for the next iteration. The per-element work consists on the node processing and on the visit to its neighborhood. In each iteration, the elements extracted from the working set can be processed in parallel. One question must be addressed: how to map the work of each node onto the GPU?

Two basic mapping strategies can be devised: fine-grained (or thread-based) and coarse-grained (or block-based) mapping. In thread-based mapping, each element in the

working set is mapped to a GPU-thread; each thread processes such element and visits its neighbors. In block-based mapping, each element in the working set is mapped to a thread-block. Different threads within the same block handle different neighbors of the element. Block-based mapping exhibits two levels of parallelism: (i) active elements are processed in parallel by different blocks, and (ii) neighbors are visited in parallel by different threads.

These mapping strategies have advantages and disadvantages. Fine-grained mapping is suitable for graphs with a regular topology (that is, low outdegree variance across the nodes) and in case of large working sets. In fact, a large outdegree distribution may cause work imbalances across threads during the neighborhood visit, thus leading to thread divergence. In addition, small working sets may lead to idle cores. The two-level parallelism of coarse-grained mapping is naturally suited to the GPU hardware. However, in the presence of nodes with small outdegree (i.e., less than 32 neighbors) this mapping strategy will keep some cores idle, thereby underutilizing the hardware. In addition, the GPU has a limited number of SMs. Therefore, block-based mapping is more suitable for dense graphs (i.e., graphs with high average outdegree) and for small working sets. Since the amount of per-node computation varies from application to application, block-based mapping may also be preferable when BFS and SSSP are building blocks of complex applications, and more work is associated to each element of the working set. In the pseudo-code in Figures 30 and 31, for example, if we exclude the neighborhood visit, the processing associated to each node is limited to setting the level/distance value. In more generic situations where additional work must be performed, block-based mapping brings an extra level of parallelism and allows distributing the work within the thread-block.

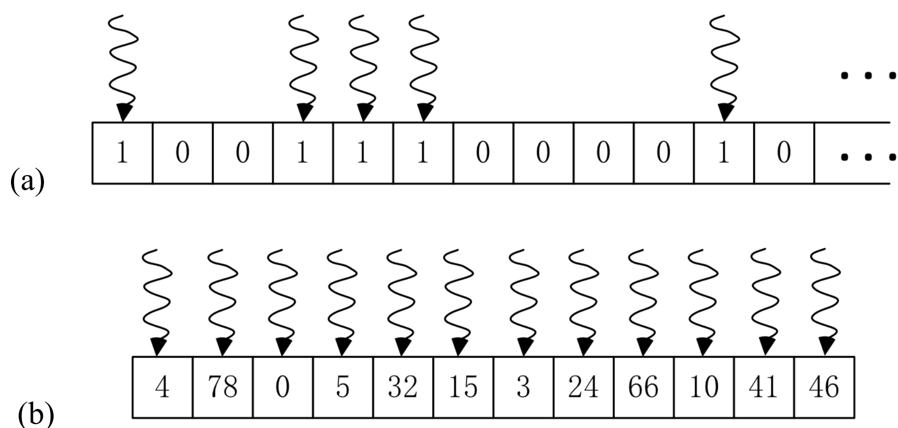


Figure 32: Working set: (a) bitmap vs. (b) queue

Both strategies allow a one-to-one and a many-to-one element-to-thread/block assignment. In the case of thread-based mapping, for example, each thread can be assigned either a single, or multiple elements of the working set. This choice affects the configuration of the kernel launches. In this work, we adopt a one-to-one mapping, which maximizes the number of threads (or thread-blocks) instantiated at every kernel call. We note that the thread- and block-based mappings are not the only options, and intermediate solutions can be devised. For example, when performing the neighborhood visit, nodes with a high outdegree can be split across multiple threads or thread–blocks. In this work, we limit ourselves to the two basic mapping strategies, and do not perform this form of load balancing.

C. Working set: bitmap vs. queue

The third dimension of exploration has to do with the working set representation. Previous work has adopted two representations: bitmap-based [75] and queue-based [77] working sets (Figure 32). The former consists of a 1-dimension array of bits, each indicating whether the corresponding element is in the working set and must be processed in the current iteration. The latter consists of a queue containing the identifiers of the

elements to be processed in the current iteration. Bitmaps are generally used in combination with thread-based mapping [75].

Both representations have advantages and disadvantages. The bitmap solution is simple to implement, update and access. In fact, it requires only a one-dimensional array of bits, and can be accessed with minimal synchronization. However, bitmaps are inefficient when sparse, that is, when the working set is small. This is especially true if the number of threads launched is equal to the number of elements in the graph (that is, nodes in case of BFS and SSSP). In this case, most threads will be idle, leading to high GPU underutilization. This inefficiency is avoided when representing the working set as a queue, which contains only elements that must be effectively processed, and can be efficiently accessed by contiguous threads. However, a queue is more difficult to implement and especially to update, since it requires more synchronization mechanisms.

4.3.3 Implementation

In this section, we discuss our implementation in details.

A. Data Structures

On both CPU and GPU, we store the graph in *compressed sparse row* (*CSR*) form, which is an efficient encoding scheme also adopted in previous work [75]. CSR represents the nodes and the edges in the graph through two one-dimensional arrays, the node vector and the edge vector. The ith entry of the node vector contains an index to the edge vector. Specifically, such index points to the start of an adjacency list containing the neighbors of node *i*. The entries in the edge vector store node identifiers, which in turn can be used to

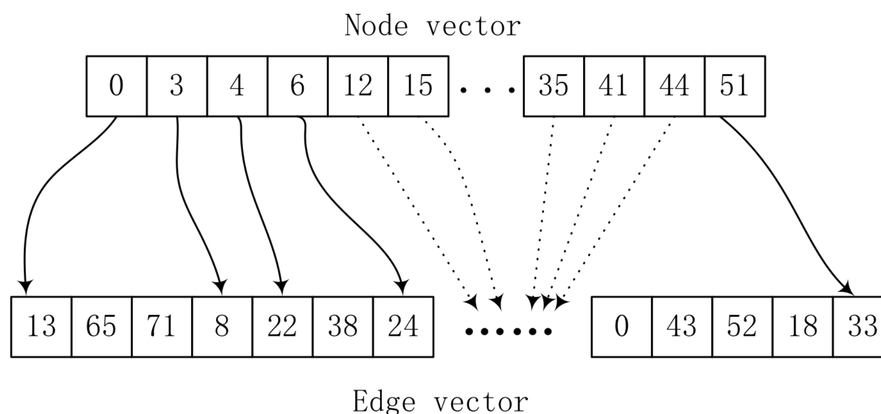


Figure 33: Compressed sparse row graph representation

index the node vector. The size of the node vector is equal to the number of nodes in the graph (+1); that of the edge vector is equal to the number of edges.

An example is shown in Figure 33, which assumes that the nodes are numbered starting from 0. The neighbors of nodes 2, for example, can be retrieved by first querying the node vector, and extracting the element at position 2 and its successor (that is, values 4 and 6, respectively). These values represent the starting and ending index of the neighbors of node 2 within the edge vector. As can be seen, the elements stored in the edge vector from position 4 to position 6 (excluded) are 22 and 38. The node vector requires an extra element to point to the end of the edge vector.

Besides the node and the edge vectors, BFS and SSSP require additional data structures, which we also represent in array form to allow for easy and efficient implementation. In particular, BFS and SSSP need an array to store the level and the distance information, respectively, which is node-specific. In addition, SSSP requires an array to record the edges' weights. Finally, some of our implementations require a node-

specific update variable to indicate whether a node needs to be updated in the current iteration.

B. Parallel Kernels

Figure 34 shows the CPU implementation framework of BFS and SSSP. In the first three steps (lines 1-3), the data structures are created and initialized on both CPU and GPU, and the required data transfers are performed. The loop in lines 4-7 represents the graph traversal, which terminates when the working set becomes empty. Each loop iteration consists essentially of the invocation of two GPU kernels: *CUDA_computation* and *CUDA_workset_gen*. The former processes the elements in the working set, computes their level/distance value, and visits their neighborhood, adding the elements that should be processed in the next iteration to an update vector. Since multiple active nodes can have common neighbors, modifications to the update vector are performed through atomic operations (thus introducing some serializations). The *CUDA_workset_gen* kernel generates the working set by transforming the update vector to bitmap or queue form. The computation and the working set generation are split into two kernels because CUDA does not offer primitives for global synchronization inside kernels.

Framework of BFS and SSSP

1: Create data structures on CPU and GPU
2: Initialize working set on CPU
3: Transfer working set and support data from CPU to GPU
4: **while** working set is not empty **do**
5: Invoke *CUDA_computation* kernel
6: Invoke *CUDA_workingset_generation* kernel
7:**end while**

Figure 34: Generic CPU pseudo-code for BFS and SSSP

The *CUDA_computation* kernel can be implemented according to all possible combinations of the alternatives in the exploration space of Figure 29. The ordered/unordered property determines which elements to extract from the working set and how to process them (Figure 30 and 31). The mapping strategy and the working set implementation affect how the work is distributed among threads and thread-blocks. The pseudo-code in Figure 34 summarizes the body of the two kernels using different mappings and working set representations. The 1st and 2nd lines of the pseudo-code partition the work among threads or thread-blocks. The 4th line represents the neighborhood visit. In the case of thread-based mapping, a single thread visits all the neighbors of the current node; in the case of block-based mapping, each thread visits a single neighbor. In the *CUDA_workset_gen* kernel, each thread processes one element in the update vector and, if necessary, adds it to the working set. The queue-based implementation requires atomic operations to avoid race conditions while adding nodes to the queue.

```
CUDA_computation kernel
1':  id = getThreadId()                        // thread mapping
1":  id = getBlockId()                         // block mapping
2':  if (id<nodeNumber && bitmap[id ])         // bitmap
2":  if (id<queue length)                      // queue
3:      process current node                   //compute level or distance
4':     current thread visits all neighbors    // generate update vector
4":     each thread in block visits a neighbor // generate update vector
5:   end if
CUDA_workset_gen kernel
1:   id = getThreadId()
2:   if (id<nodeNumber && update[id])
3':     generate bitmap working set            // bitmap
3":     generate queue working set             // queue
4:   end if
```

Figure 35: Pseudo-code of kernel functions (computation and workset_gen)

The ordered and unordered implementations of BFS are very similar. The ordered SSSP has the added complexity of finding the minimum distance value in the working set (*findmin*), operation which is not required by the unordered version of SSSP. A CPU implementation of ordered SSSP usually uses a heap data structure to ensure fast insertions in the working set and accesses to it. We implemented the findmin operation on GPU by parallel reduction (which is faster than maintaining a heap on CPU).

C. Working Set on GPU

The bitmap representation of the working set was first adopted in [75] in combination with thread-based mapping. The advantage of this representation is its simplicity: since each node has an own entry in the bitmap and is handled by a different thread (or thread-block), no synchronization is required when accessing the working set or generating it from the update vector.

The queue representation has an added complexity. As explained above, the *CUDA_workset_gen* kernel requires atomic operations to convert the update vector into queue form. In this work, we adopt the basic implementation described in [115]. Specifically, each thread uses an atomic operation only to get a unique insertion index within the queue. Thus, threads get indexes sequentially, but insert nodes into the queue in parallel. Once the queue-based working set is created, accesses to it within the *CUDA_computation* kernel are coalesced, and do not require any additional synchronization mechanism.

There are several ways to improve the performances of work queues on GPU. Luo et al [76] propose a hierarchical queue implemented using both shared and global memory. The use of shared memory provides faster accesses and lighter synchronizations. To

avoid serialization while generating indexes into the queue, Merril et al [77] replace atomic operations with prefix scans. These optimizations are orthogonal to this work, but can certainly be applied to our reference implementations.

4.3.4 Adaptive Runtime

As discussed in the previous sections, the heterogeneity of the graphs used in practical applications makes it impossible to determine a single GPU implementation which is optimal across all datasets and algorithms. This fact is supported by the experimental evaluation that we will present in Chapter 4.3.5. To tackle this problem, we design an adaptive runtime that allows dynamically selecting the GPU implementation that better suits the characteristics of the dataset. Specifically, our runtime can perform coarse- and fine-grained decisions. First, given a graph and an algorithm (e.g. BFS or SSSP), it can select the best GPU implementation according to the graph topology and the underlying GPU hardware. Second, while processing a graph, our runtime can dynamically switch between different implementations of the same algorithm in different phases of the traversal.

A. Overview

The structure of our adaptive framework is shown in Figure 36. We expose to the user an API consisting of an abstract graph data type. Such API provides primitives to define and instantiate graphs, as well as functions to run the SSSP and BFS algorithms on them. At the low level, we have different GPU implementations for both SSSP and BFS, as defined by the exploration space described in Figure 28. Between the graph API and the algorithm implementation layers, we have a runtime layer. Such runtime consists of two

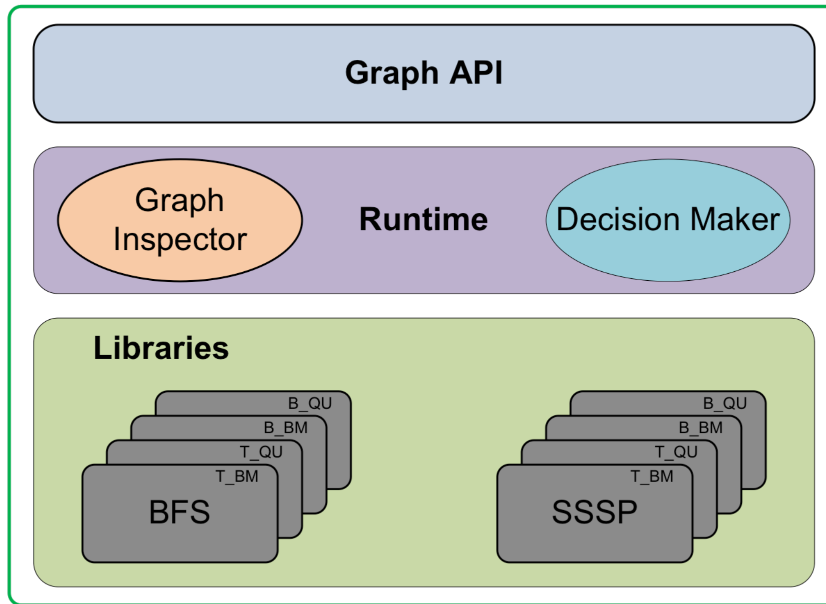


Figure 36: Overview of our adaptive framework

components: a *graph inspector* and a *decision maker*. The former inspects relevant characteristics of the graph (e.g., number of nodes, number of edges, minimum, maximum, average node degree), and monitors significant runtime attributes (e.g., working set size). The latter dynamically selects the most suitable implementation based on the values of these attributes and on the hardware characteristics of the underlying GPU.

In our experimental evaluation, we found that *unordered implementations of both BFS and SSSP generally perform better than their ordered counterparts*. This observation is coherent with [113]; this result is due not only to the larger amount of parallelism available in unordered graph algorithms, but also to the overhead due to applying the order relationship to the working set in case ordered versions. Therefore, our adaptive framework uses *only unordered versions of SSSP and BFS*, and makes decisions in two dimensions: mapping method and working set implementation. This leads to 4

combinations: (1) thread mapping + bitmap, (2) thread mapping + queue, (3) block mapping + bitmap, and (4) block mapping + queue. As discussed below, the selection mechanism takes the utilization of the GPU hardware into account: specifically, we consider the fraction of cores and SMs effectively used, as well as the amount of thread divergence introduced.

B. Selection of the Mapping Method

The first decision to be made by our runtime system is whether to use thread- or block-based mapping. This decision is based on two considerations: core/SM utilization and amount of thread divergence introduced.

Core/SM utilization - As mentioned in Chapter 4.3.1.C, in CUDA thread-blocks are mapped onto SMs and threads are mapped onto cores. In Fermi GPUs (used in this work), each SM consists of 32 or 48 cores. In each graph traversal step, only nodes belonging to the working set need to be processed. The working set size is an indicator of the amount of coarse-grained parallelism available within a graph traversal step. Thus, small working sets (for which thread-based mapping is unable to fully utilize the available GPU cores) make block-based mapping preferable. Thread-based mapping becomes a viable option when the working set size approximates the number of cores available on the GPU.

In case of large working sets, both thread- and block-based mapping are viable options, and an additional selection criterion is required. To this end, we consider the *average outdegree* of the nodes in the graph. In case of block-based mapping, the neighborhood visit is cooperatively performed by the threads within the block (that is, each thread will process one or more neighbors). The minimum practical size for a thread-block corresponds to the warp size (i.e., 32 threads). If block-based mapping is

used, an average outdegree well below the warp size causes cores within a SM to be unutilized. Therefore, a small average outdegree makes a thread-based mapping preferable to block-based mapping.

Thread divergence – Using the average outdegree to discriminate between thread- and block-based mapping helps also with another consideration: in case of thread-based mapping, better performances are achieved if the amount of warp divergence is limited. Since, in case of thread-based mapping, every thread processes all the neighbors of an active node, the performances of each warp will be limited by the node with the largest outdegree. In particular, large outdegree variance may cause warp divergence. As can be observed in Table 3, graphs with high average outdegree tend to exhibit uneven outdegree distributions. By using thread-based mapping only when the average outdegree is low, we limit the amount of thread divergence, which would originate by unbalanced outdegree distributions.

C. Selection of the Working Set Representation

The second decision to be made by our runtime is the working set representation, i.e., bitmap vs. queue. As discussed in Chapter 4.3.2.C, a queue implementation involves a larger number of synchronizations. In particular, the creation of the queue requires a number of atomic operations equal to the queue length; such atomic operations introduce serialization among threads, thus degrading performance. This suggests that bitmaps are preferable to queues in the presence of large *working sets*. This criterion is also coherent with another consideration. When using a bitmap representation, there will be a one-to-one mapping between threads (for thread-based mapping) or blocks (for block-based mapping) and nodes. Small working sets can cause many threads/blocks to be invoked

without performing any real work, leading to core/SM underutilization. Specifically, in case of a bitmap representation and a graph with |N| nodes, a working set of size |WS| leads to a fraction of wasted threads/blocks equal to 1-|WS|/|N|. In conclusion, small working sets will be implemented using a queue, and large ones using a bitmap.

D. Decision Space

The decision space resulting from the previous considerations is illustrated in Figure 36. We represent the size of the working set along the x-axis, and the average outdegree of the graph along the y-axis. This decision space is broken into 5 regions by three threshold values: T_1, T_2 and T_3. In particular, T_1 and T_2 correspond to the considerations made for the selection of the mapping method, and T_3 to the one for the choice of the working set representation.

The areas in the decision space represent different implementations. To the left of T_2, the implementation will always be B_QU (block-mapping + queue). Between T_2 and T_3, the working set is implemented with a queue while the mapping strategy depends on the average node outdegree (see T_1). To the right of T_3, the working set is represented as a bitmap, and the mapping still depends on the average outdegree (T_1). T_1, T_2 and T_3 are experimentally tuned, as discussed in Chapter 4.3.5.

E. Runtime Overhead

To understand the overhead introduced by our runtime, we must consider its two components: the decision maker and the graph inspector. The former has extremely low overhead since its logic (summarized by Figure 37) is straightforward. In order to allow the decisions described above, our graph inspector must monitor the working set size and the average outdegree of the nodes within the working set. This information can be

collected at runtime by running a separate kernel (parallel scan can allow a more efficient computation of the average outdegree). This overhead is much greater than that of the decision maker. In our implementation, we reduce this overhead in two ways: (i) by considering the average outdegree of the whole graph (which is a value computed only once when reading the graph) rather than the one of the current working set, and (ii) by sampling (that is, by not performing measurements in every traversal step). These design decisions represent a trade-off between execution efficiency and runtime overhead. The selection of the sampling rate and its effect on performances will be discussed in Chapter 4.3.5.

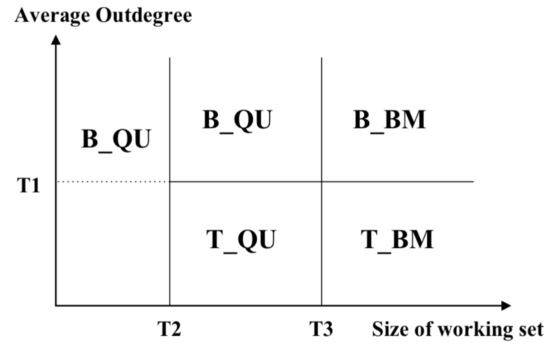


Figure 37: Design space

4.3.5 Experimental Evaluation

We present an experimental evaluation on the datasets of Table 3. We first evaluate the static implementations corresponding to Figure 29. We then study how to tune the parameters of our adaptive runtime. We finally compare the performance achieved through our runtime with those achieved through the static solutions.

Our testing platform consists of an Intel Core i7 CPU (running CentOS 5.5) and an Nvidia Tesla C2070 GPU, which contains 14 32-core SMs. We use gcc 4.1.2 and nvcc 4.0 compilers, both with –O3 optimizations. Our results include CPU processing, GPU

processing and CPU-GPU transfer times. We do not measure the time spent loading graph data from the hard drive.

A. Performance of Static Implementations

Tables 4 and 5 summarize the performances of the BFS and SSSP implementations covering the exploration space in Figure 29. In particular, the tables report the speed up of each GPU implementation over a serial CPU implementation. All the GPU solutions are named with three fields separated by underscore. The first field indicates whether the implementation is ordered (O) or unordered (U); the second field distinguishes thread-based (T) and block-based (B) mapping; the third field indicates the representation of the working set: bitmap (BM) vs. queue (QU). For instance, "O_B_BM" indicates that the implementation is ordered, uses block-based mapping and a bitmapped working set. For each dataset, grey cells show the best performance achieved.

The tables show the results reported using the best kernel configurations, which have been obtained by using the "CUDA Occupancy Calculator" and conducting experiments under different settings. When using thread-based mapping, we found that the best results can be achieved with 192 threads per block. When using block-based mapping, the optimal number of threads per block is the multiple of 32 closest to the average node outdegree in the graph.

Table 4: Speedup of BFS (GPU implementation over serial CPU baseline)

	O_T_BM	O_T_QU	O_B_BM	O_B_QU	U_T_BM	U_T_QU	U_B_BM	U_B_QU
CO-road	0.81	1.12	0.04	1.15	0.94	1.50	0.04	1.49
CiteSeer	24.39	15.63	12.35	49.22	24.68	15.04	12.48	48.94
p2p	3.79	3.34	0.93	3.22	3.66	3.44	0.95	3.37
Amazon	13.59	11.12	2.05	10.62	13.94	10.60	2.07	11.52
Google	20.76	18.82	2.94	18.90	21.57	18.03	2.96	20.36
SNS	20.39	16.33	8.43	24.02	24.04	18.00	8.64	24.30

For BFS, we can make the following observations. First, when using the same mapping strategy and working set representation, ordered and unordered algorithms achieve very similar performance. In ordered BFS, the nodes are processed level by level and each node is accessed exactly once. In unordered BFS, in principle each node may be updated multiple times. However, since in every iteration we process the entire working set, our unordered GPU implementation also proceeds level by level, unless the working set is initialized through some depth-based traversal. We experimentally verified that limited amount of initialization (e.g., depth-based traversal in 3-5 directions) does not substantially affect the results. Second, the GPU implementation does not outperform its CPU counterpart on all datasets. In fact, the GPU performance is poor for the *CO-road* network, whose average outdegree is only 2.6 (see Table 3), and whose diameter is relatively large (more than 1000 levels). Third, the best GPU implementation varies from dataset to dataset. For instance, the CO-road and CiteSeer networks favor U_B_QU, while the Amazon and *p2p* networks achieve best performance with U_T_BM.

Table 5: Speedup of SSSP (GPU code over serial CPU baseline)

	O_T_BM	O_T_QU	O_B_BM	O_B_QU	U_T_BM	U_T_QU	U_B_BM	U_B_QU
CO-road	0.02	0.01	0.0012	0.01	1.88	1.76	0.35	2.11
CiteSeer	136.23	112.77	11.81	139.53	126.47	118.91	483.24	867.91
p2p	1.29	1.16	0.22	1.22	135.65	127.38	49.17	131.88
Amazon	3.29	3.03	0.26	3.12	95.10	58.93	37.71	99.83
Google	2.27	2.16	0.18	2.19	96.57	58.12	32.94	89.32
SNS	25.37	24.33	1.87	24.81	174.82	140.45	136.08	276.23

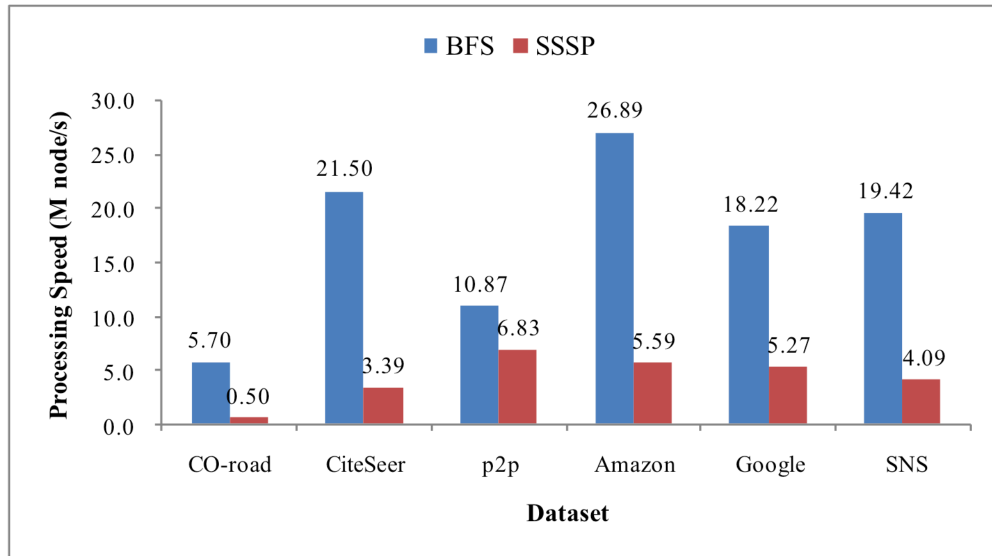


Figure 38: Processing speed of bet implementation

For SSSP, we can observe the following facts. First, unordered algorithms are significantly faster than their ordered version. This has two motivations: (1) unordered SSSP exhibits more parallelism than ordered SSSP, and (2) ordered SSSP suffers from the cost of implementing the findmin operation. Second, the ordered SSSP on GPU can achieve considerable speedup over its serial CPU version. In every iteration, nodes at the same distance can be processed in parallel. In addition, the parallel reduction on GPU is a good alternative to executing the findmin operation on CPU. Third, by concurrently processing the elements in each node's neighborhood, block-based mapping leads to high speedups on graphs with large average outdegrees (e.g., *CiteSeer* and *SNS*). Finally, we can again observe that the best implementation strictly depends on the dataset; for example U_B_BM performs very well on CiteSeer, but exhibits the worst performance on the other 5 datasets.

Figure 38 shows the processing speed (in millions nodes per second) of the best GPU implementation of BFS and SSSP across the considered datasets. BFS achieves better

performance than SSSP due to its faster convergence. Our experiments show similar results to previous work [76] and prove that GPU can be successfully used to run graph algorithms on large datasets. Since our results show that the best solution depends on the characteristics of the underlying dataset, we now evaluate the use of our adaptive runtime.

B. Parameter Tuning for our Adaptive Runtime.

Before evaluating the performance of our adaptive runtime, we study how to tune its parameters. In particular, we start with T_1, T_2, and T_3, the thresholds used by the decision maker and illustrated in Figure 37.

Recall that T_1 is related to the average outdegree of the graph, and allows discriminating between thread- and block-based mapping when the size of the working set would allow both alternatives. Since each thread-block must at least contain one warp (i.e., 32 threads), if the average outdegree is less than 32, block-based mapping will underutilize the hardware resources. Thus, we set T_1 to 32.

T_2 indicates the size of the working set below which block-based mapping should be always preferred to thread-based mapping. Its value is related to the kernel configuration and the number of SMs on the GPU. As mentioned in the previous section, we experimentally verified that good configurations for thread-based mapping are characterized by 192 threads per block. The GPU used in our experiments has 14 SMs. When the size of the working set is less than 192*14 = 2,688 nodes, thread-based mapping will leave some SMs idle, thus underutilizing the GPU. To confirm this analysis, we measure the kernel execution time of T_QU and B_QU across all iterations of BFS and SSSP. Our results show that B_QU outperforms T_QU for working set sizes smaller than ~3000. Therefore, we set T_2 to 2,688.

T_3 indicates the size of the working set above which a bitmap representation is preferable to a queue. In Figure 39, we report the results of experiments conducted to study how the performance changes with T_3. We recall that the ratio between the size of the working set and the number of nodes in the graph indicates, in case of a bitmap representation, the fraction of threads/blocks instantiated that will effectively perform some work. Therefore, in the x-axis we show the percentage ratio of T_3 over the number of nodes in the graph. As can be seen, for all datasets but *CiteSeer*, the execution time increases with T_3. This can be easily explained as follows: in the presence of large working sets, the queue generation incurs higher overheads due to atomic operations. When the ratio T_3/*#node* is less than 6%, the execution time increases very slowly. However, when it exceeds 7% or 8%, the execution time increases rapidly (*CO-road*, *p2p* and *SNS*). Although this trend is not exactly the same for all datasets, we set the value of T3 to 6%. It is worth explaining why, in the case of *CiteSeer*, the execution time decreases even when the ratio T_3/*#node* reaches 13%. The *CiteSeer* dataset is characterized by a high average outdegree, which leads to higher parallelism. In case of a

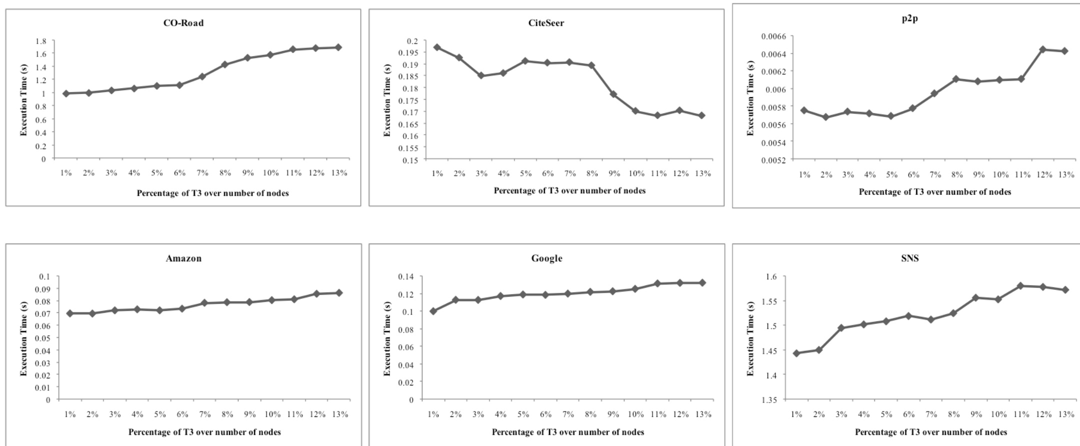


Figure 39: Performance under different T_3 settings (SSSP)

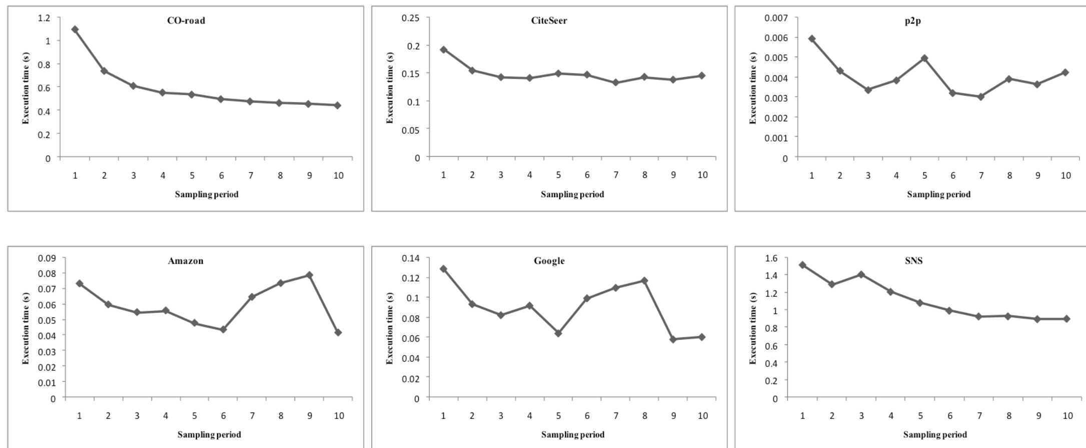


Figure 40: Performance under different sampling rates (SSSP)

queue implementation, the amount of work performed within a thread-block amortizes the overhead due to the atomic operations performed when generating the queue, making a queue preferable to a bitmap.

The graph inspector introduces a runtime overhead while monitoring the working set size. Such overhead can be reduced by performing this measurement task periodically, rather than at every iteration. Figure 40 shows the performance under different sampling rates. Due to lack of space and given the similarity between BFS and SSSP, we show only the SSSP results. We can observe that the performance generally benefits from a decreased sampling rate. However, the changes are not smooth and vary across datasets. In general, datasets characterized by a longer convergence time (e.g. *CO-road*, *CiteSeer*, and *SNS*) experience a slow and steady performance improvement as the sampling rate decreases. On the other hand, graphs that take only a few iterations to converge (e.g., *Google*, *p2p*) are more sensitive to changes in the sampling rate. To make a trade-off, we set the sampling rate of our adaptive runtime to 6. With this setting, we observed a runtime overhead varying from 10.6% (on *CiteSeer*) to 13% (on *p2p*).

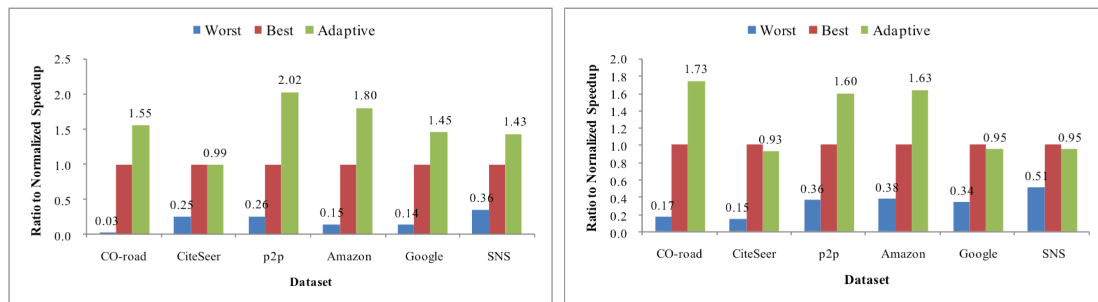


Figure 41: Performance of our adaptive runtime on BFS (to the left) and SSSP (to the right) – baseline: best static solution

C. Overall Performance of Adaptive Runtime

Figure 41 shows the speedup comparison between our adaptive runtime, the worst and the best static solution. In every case, the best static implementation is taken as baseline. Only unordered implementations are considered (also for the worst case). The values reported on the bars are the speedup numbers of the worst static and of the adaptive solution over the best static implementation. Since our goal is to argue that an adaptive system can capture dynamic parallelism and not to develop a highly tuned code, in our study we use basic kernels similar to those by Harish and Narayanan [75]. However, the performance achieved by our dynamic solution is in the same order of magnitude as that achieved by Merrill et al [77] (for example, our dynamic BFS computes 0.74 billion edges/sec on the *CiteSeer* network).

We can observe the following. First, the performance of the worst static solution can be as low as 3% (and as high as 52%) of that of the best static solution. Second, despite its overhead, our adaptive runtime achieves better performance than the best static implementation on most of these real world graphs. Specifically, the speedup over the best static solution ranges from 1.43 to 2.02. Although on *CiteSeer*, *Google* (SSSP) and *SNS* (SSSP) the adaptive runtime has no advantage compared to the best static solution, it

still achieves 95-99% of its performance (and avoids the penalty associated with possibly choosing a bad implementation).

4.4 Parallelization Templates

The effective deployment of applications exhibiting irregular nested parallelism on GPUs is still an open problem. A naïve mapping of irregular code onto the GPU hardware often leads to uneven work distribution. As a consequence, simple parallelization templates handling all the loop iterations in the same way may lead to hardware underutilization. On the other hand, adding simple load balancing mechanisms to the parallelization may allow a better mapping of work to hardware and, consequently, it may improve the application performance. In this work, we focus on simple load balancing schemes and study the use of dynamic parallelism as a means to achieve better hardware utilization. Specifically, we investigate mechanisms to effectively distribute irregular work to streaming multiprocessors and GPU cores. Some of our parallelization templates rely on dynamic parallelism, a feature recently introduced by Nvidia in their Kepler GPUs and announced as part of the OpenCL 2.0 standard. We propose mechanisms to maximize the work performed by nested kernels and minimize the overhead due to their invocation. Our parallelization techniques can be incorporated in compilers, thus freeing the programmer from the need to worry about the mapping of work to the hardware and to understand the complex semantics of GPU dynamic parallelism.

4.4.1 Motivation

GPUs present two levels of hardware parallelism: streaming multiprocessors and CUDA cores in Nvidia's parlance, and SIMD units and stream processors in AMD's. Commonly used programming models for many-core processors, such as CUDA and OpenCL, expose this two-level hardware parallelism to the programmer through a two-level multithreading model: at a coarse grain, CUDA thread-blocks and OpenCL work-groups are mapped onto streaming multiprocessors and SIMD units; at a fine grain, CUDA threads and OpenCL work-items are mapped onto cores and stream processors. An additional level of parallelism is provided by the opportunity to launch multiple kernels concurrently through CUDA streams and OpenCL command queues. In addition, Kepler GPUs allow nested kernel invocations; this functionality, called dynamic parallelism, has recently been announced as part of the OpenCL 2.0 standard and will soon be included in AMD devices. In the remainder of this paper, we will use CUDA terminology.

Despite the presence of hardware and software support for nested parallelism, finding the optimal mapping of algorithms exhibiting multiple levels of parallelism onto GPUs is not a trivial problem. Especially in the presence of irregular computation, a naïve mapping of irregular code onto the GPU hardware can lead to resource underutilization and, thereby, limited performance. Besides allowing recursion, dynamic parallelism has the potential for enabling load balancing and improving the hardware utilization. This is because nested kernels, each with a potentially different degree of parallelism, are dynamically mapped to the GPU cores by the hardware scheduler according to the runtime utilization of the hardware resources. Unfortunately, the effective use of this

feature has yet to be understood: the invocation of nested kernels can incur significant overhead [91] and may be beneficial only if the amount of work spawned is substantial.

In this work, we only consider applications containing parallelizable irregular nested loops. We propose and analyze different parallelization templates aimed to improve the hardware utilization and the performance – some of them leveraging the dynamic parallelism feature [116]. Our goal is to propose and evaluate simple and yet effective mechanisms that can be incorporated in GPU compilers.

4.4.2 Irregular Nested Loop

The hierarchical nature of the GPU architecture leads to two logical ways to map parallel loops onto the hardware: different loop iterations may be mapped either to GPU cores (thread-based mapping) or to streaming multiprocessors (block-based mapping) [102]. In the former case, loop iterations are assigned to threads; in the latter, they are assigned to thread-blocks. For brevity, we will use the terms thread- or block-mapped loops and kernels to indicate loops and kernels parallelized with one of these two approaches. In the presence of loop nesting, the mapping of inner loops depends on the mapping performed on the outer loops. For example, thread-based mapping on an outer-loop will cause serialization of all inner loops, while block-based mapping on an outer-loop will allow thread-based mapping on the inner loops. In addition, stream-based mapping (whereby different iterations of the loop are assigned to different CUDA streams) offers an additional degree of freedom to the parallelization process. During code generation, compiler analysis is required to identify the type of GPU memory where each variable must be stored and the need for synchronization and reduction operations to access

shared variables. For example, in the presence of a block-mapped outer loop and a thread-mapped inner loop, variables defined inside the outer loop but outside the inner loop will be shared by all threads in a block. Therefore, these variables will need to be stored in shared memory, and their access by threads within the inner loop will require synchronization.

```
(a) irregular nested loop
foreach (int i = 1 to N)
  foreach (int j = 1 to f[i])
    work(i,j);
(b) dual-queue
set_low = {i :: f[i] ≤ lb_THRES}
set_high = {i :: f[i] > lb_THRES}
thread-mapped_kernel(set_low)
block-mapped_kernel(set_high)
(c) delayed-buffer
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    buffer.add(i);
}
blk-mapped-exec(buffer, work);
(d) dynamic parallelism œ naïve
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    blk-mapped_nested-kernel_TH(i,work);
}
(e) dynamic parallelism œ optimized
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    buffer_SM.add(i);
}
blk-mapped_nested-kernel_BL(buffer_SM, work);
```

Figure 42: Parallelization templates for irregular nested loops

Simply relying on thread- and block-based mapping in the parallelization process is acceptable for regular nested loops, wherein the number of iterations of an inner loop does not vary across the iterations of the outer loop. However, this simple solution may lead to inefficiencies when applied to irregular nested loops, for which this property does not hold. Irregular nested loops have the structure of the code in Figure 42(a). As can be seen, the number of iterations of the inner loop is a function of i, the outer loop variable. In the case of irregular nested loops, the use of thread-based mapping on the outer-loop may cause warp divergence (i.e., different threads are assigned different amounts of work), while the use of block-based mapping will lead to uneven block

utilization, which in turn may cause GPU underutilization. Load balancing of irregular nested loops is one of the use cases for GPU dynamic parallelism. By launching nested grids of variable size, dynamic parallelism has the potential for improving the GPU utilization. However, despite its overhead has been reduced in recent CUDA distributions, the effective use this feature depends on the amount of work spawned in each kernel invocation.

We consider different parallelization templates aimed to perform load balancing in the presence of irregular nested loops. The proposed code variants trade off the advantages and disadvantages of thread- and block-based mapping, respectively. In particular, in Figure 42(b-e) we illustrate the load-balancing variants of the thread-based mapping template. Those variants rely on a load balancing threshold parameter (lb_{THRES}). The dual-queue template in Figure 42(b) divides the elements processed in the outer loop in two queues depending on the number of iterations they require in the inner-loop and processes those queues separately using thread- and block-based mapping, thus reducing the warp divergence of the former and the block-level work-imbalance of the latter. The *delayed-buffer* template in Figure 42(c) delays the execution of large iterations of the outer loop by queuing them in a buffer and then processing them using block-based mapping. We consider two versions of this template: one that stores the buffer in global memory (*dbuf-global*), thus requiring two kernel calls and allowing redistributing the work among thread-blocks in the second phase; the other that stores the buffer in shared memory (*dbuf-shared*), thus requiring a single kernel invocation but possibly leading to uneven work distribution among thread-blocks. The naïve dynamic parallelism (*dpar-naïve*) template in Figure 42(d) invokes a nested call for each "large" iteration (and

performs the dynamic parallelism calls at a thread level). Finally, the optimized dynamic parallelism template (*dpar-opt*) in Figure 42(e) delays spawning nested calls to a second-phase; by invoking a single dynamic parallelism call for each thread-block, this code variant spawns fewer and larger kernels.

We note that GPU dynamic parallelism has fairly elaborate semantics. For example, nested kernel calls are performed by threads, but their synchronization happens at the level of thread-blocks; registers and shared memory variables are not visible to nested kernels; memory coherence and consistency between parent and child kernels require explicit synchronization; and concurrent execution requires the use of CUDA streams. Fortunately, the proposed parallelization templates for nested loops are simple and can be incorporated in a compiler, allowing the programmer to write only the simplified code in Figure 42(a). The automatic generation of the code variants corresponding to the proposed parallelization templates by a compiler will therefore hide from the programmer not only the two-level hardware and software organization of the GPU, but also the execution and memory model of GPU dynamic parallelism.

Several factors may impact the effectiveness of the proposed parallelization templates. For example, the optimal load balancing threshold (lb_{THRES}) will depend on the underlying dataset and algorithm. In addition, the performance of each parallelization template will depend on the characteristics of the algorithm (that is, the nature of the work in Figure 42). We explore these aspects in our experimental evaluation.

4.4.3 Experimental Evaluation

A. Experimental Setup

Hardware and Software Setup: We run all our experiments on a server equipped with a Xeon E5-2620 CPU and an Nvidia K20 GPU. The machine uses CentOS release 6.4. We compiled and run our code using gcc v4.4.7 and CUDA v6. We used Nvidia Visual Profiler to collect the profiling data presented in our analysis.

Benchmark Applications: We evaluated the performance of the proposed parallelization templates on four applications, which include irregular nested loops (*SSSP*, *BC*, *PageRank*, *SpMV*). Whenever available, we used open-source implementations of these applications as baselines.

- **Single-Source Shortest Path (*SSSP*)**: For *SSSP*, we used the thread-mapped implementation described in [75] as baseline. *SSSP* is a memory intensive application with scattered memory accesses.
- **Betweenness Centrality (*BC*)**: A node's *BC* is an indicator of its centrality in a network, and its value is equal to the number of shortest paths from all nodes to all others that pass through that node. Our parallel implementation is based on Brandes' algorithm and operates in two phases. First, it constructs the shortest paths tree using BFS (we consider unweighted graphs); second, it computes the BC value by traversing the shortest path tree. Both phases present irregular nested loops and scattered memory accesses.
- **PageRank**: *PageRank* is a popular graph analysis algorithm to rank web pages. We consider the GPU implementation described in [117] as a reference. This algorithm contains a parallelizable, irregular nested loop: each iteration of the

outer loop processes a different webpage (node in a graph); the inner loop collects ranks from the neighbors of the considered node.

- **Sparse Matrix-Vector multiplication (*SpMV*)**: *SpMV* [118] calculates the product of a sparse matrix and a dense vector, and is an important building block for diverse applications in scientific computing, financial modeling and information retrieval. Since the sparse matrix is represented in Compressed Sparse Row format, the nested loop within the matrix multiplication algorithm is irregular.

Datasets: For *SSSP*, *PageRank* and *SpMV*, we use *CiteSeer*, a paper citation network from the *DIMACS* implementation challenges [7]. *CiteSeer* [119] has about 434 thousand nodes, 16 million edges and a node outdegree that varies from 1 to 1,188 across the graph (with an average of 73.9). For *BC* we use Wikipedia's who-votes-on-whom network (*Wiki-vote*) [120], a small-world network consisting of about 7 thousand nodes, 100 thousand edges and a node outdegree that varies from 0 to 893 across the graph (with an average of 14.6.9).

B. Experimental Results

In this section, we first discuss the selection of the kernel configurations used in our experiments; we then illustrate the rationale of our experiments by presenting some results on *SSSP*; and we finally complete our discussion by comparing the results reported on *PageRank*, *BC* and *SpMV*.

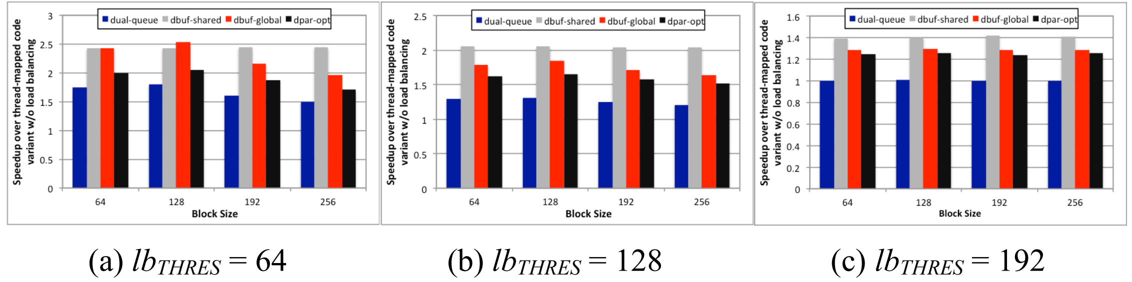
All the charts presented in this section report the speedup of the code variants derived from the use of the parallelization templates in Figure 42 over a baseline implementation that uses thread-mapping in the outer loop and no load balancing. The baseline GPU

implementations achieve the following speedups over serial CPU code: 8.2x (*SSSP*), 2.5x (*BC*), 15.8x (*PageRank*) and 2.4x (*SpMV*).

Kernel configurations – Recall that the parallelization templates in Figure 42 include two phases: one using thread-based mapping and one using block-based mapping. For example, the *dbuf-global* scheme first invokes a kernel where the outer loop is parallelized with thread-based mapping, and then it invokes a kernel that uses block-based mapping to process the delayed buffer. Here, we discuss the selection of the kernel configuration used in both cases.

For thread-based mapping, we leverage the CUDA occupancy calculator to determine the optimal thread-block size. Since the considered applications have a low register and shared memory utilization (and some of them do not use shared memory at all), the optimal block size results to be large in all cases. Specifically, we use 192 threads per block, equaling the number of cores per streaming multiprocessor on Kepler GPUs. We recall that in a thread-mapped implementation each thread is assigned one or more iterations of the outer loop in Figure 42(a). Hence, we configure the number of blocks to be run based on the block size, the total number of iterations to be run and the maximum grid configuration allowed on Kepler GPUs.

In case of block-based mapping, each iteration of the outer loop is assigned to a thread-block, and threads within a block execute iterations of the inner loop. A small thread-block configuration will tend to assign multiple iterations of the inner loop to each thread. Conversely, large blocks may lead to hardware underutilization, as some iterations of the outer loop may not present enough parallelism to fully exploit the GPU cores on a streaming multiprocessor. We recall that the value of the lb_{THRES} (load



(a) lb_{THRES} = 64 (b) lb_{THRES} = 128 (c) lb_{THRES} = 192

Figure 43: *SpMV*: Speedup of load balancing code variants over basic thread-mapped implementation under different settings

balancing threshold) parameter determines the iterations of the outer loop that are processed in the second, block-mapped phase of the code.

We performed a set of experiments to determine good block size configurations to be used in the block-mapped portions of the code under different lb_{THRES} settings. Figure 43 shows the results of experiments performed on *SpMV* using different block size configurations and various settings of parameter lb_{THRES}. All experiments are run on the *CiteSeer* network. We omit the *dpar-naïve* implementation because, as we will show later, it greatly underperforms the other code variants. According to the CUDA occupancy calculator, small blocks consisting of 32 threads lead to low hardware occupancy. Our experiments confirmed low performance with fewer than 32 threads per block; therefore, here we consider block sizes $\geqslant$ 64. As can be seen, the performance is insensitive to the block size but mainly affected by lb_{THRES}. Some templates perform better in the presence of smaller blocks, especially for small values of lb_{THRES}. We observed similar results on the other applications with different datasets. These results can be explained as follows. A block size larger than the value of lb_{THRES} may lead to hardware underutilization. To understand why, refer to the pseudo-code in Figure 42. All outer loop iterations presenting an inner loop size *f(i)* greater than lb_{THRES} are processed in a block-mapped

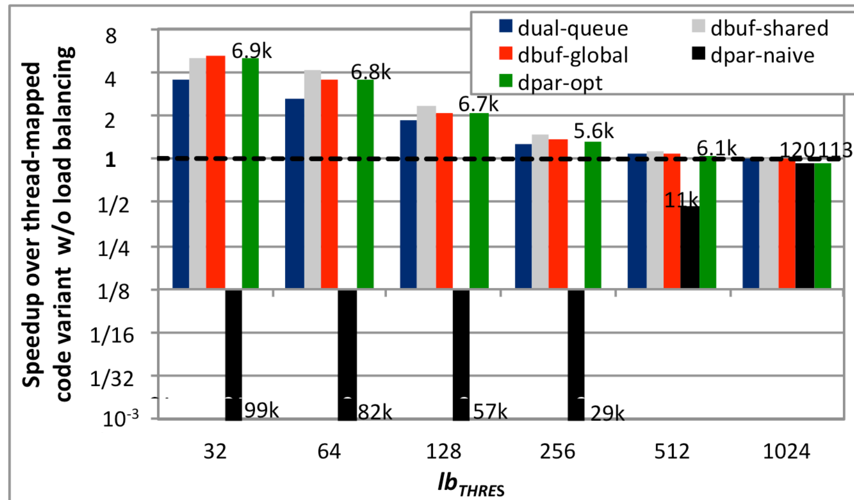


Figure 44: *SSSP*: Speedup of load balancing code variants over basic thread-mapped implementation

fashion. If *f(i)* is smaller than the block-size, the threads within the block may not be assigned any work, leading to GPU underutilization. Therefore, in the remaining experiments we use small blocks consisting of 64 threads for the block-mapped kernels.

Results on SSSP: We now compare the performance of the five parallelization templates on SSSP. We refer to the basic implementation described in [75], which encodes the graph data structure in *Compressed Sparse Row* (*CSR*) format (more information can be found in [75]). When a graph is represented in CSR format, its traversal assumes the form of the nested loop in Figure 41(a), where the outer loop iterates over the nodes in the graph, and the inner loop over the neighbors of each node i. In irregular graphs where the node outdegree (*f[i]*) varies significantly from node to node, this traversal loop is irregular. Here, we show experiments performed on *CiteSeer*. Figure 44 shows the speedup of all code variants over the baseline thread-mapped implementation; the number of nested kernel calls performed by the two dynamic parallelism-based solutions are reported on top of the bars. Due to the irregular nature of the *CiteSeer* graph, almost all code variants that include load balancing outperform the

basic thread-mapped implementation. The *dpar-naïve* code variant is the exception: due to the overhead of the large number of (small) nested kernel calls performed, this implementation leads consistently to (often significant) performance degradation. The delayed buffer-based and the optimized dynamic parallelism-based code variants yield the best results, and the performance improvement depends on the value of the load balancing threshold, which affects the amount of load balancing performed. The optimal value of this parameter corresponds to the warp size (no improvements were observed for $lb_{THRES}<32$).

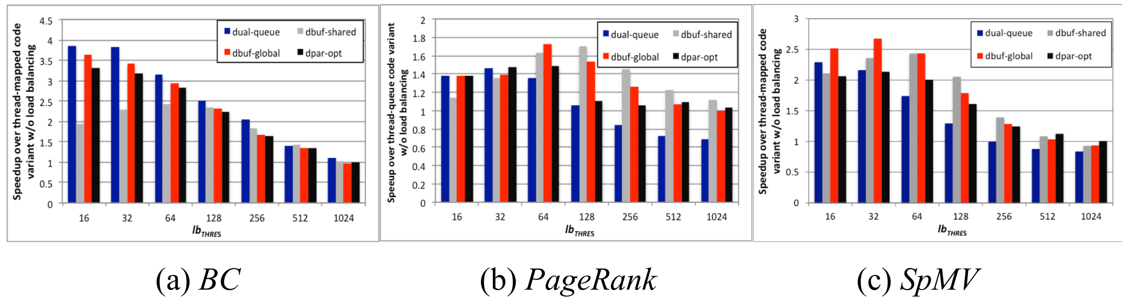
To better understand these results, we used the Nvidia Visual Profiler to collect three performance metrics: the warp execution efficiency (ratio of the average active threads per warp to the maximum number of threads per warp on a streaming multiprocessor), and the global memory load and store efficiency (ratio of the number of requested memory load and store transactions to the actual number of load and store transactions performed - the lack of memory coalescing can cause more memory transactions than requested to be triggered). Table 6 shows the profiling data gathered for $lb_{THRES} = 32$. As can be seen, all parallelization templates but *dpar-naïve* report an increased warp efficiency compared to the baseline code. This indicates a better utilization of the available GPU cores. In addition, by processing nodes with low outdegrees ($< lb_{THRES}$) and nodes with high outdegrees separately, these code templates improve the memory load and store efficiency. Finally, thanks to its use of shared memory, *dbuf-shared* improve the memory coalescing over *dbuf-global*, leading to better memory efficiency. To conclude, our proposed parallelization templates improve both GPU core utilization and memory access patterns.

Table 6: Profiling data collected on SSSP (lb_{THRES}=32)

Templates	Metrics		
	warp efficiency	*gld efficiency*	*gst efficiency*
baseline	35.6%	15.8%	3.2%
dual-queue	74.9%	79.1%	4.8%
dbuf-shared	75.7%	94.3%	50.4%
dbuf-global	72.3%	89.1%	8.5%
dpar-naïve	25.3%	45.5%	16.3%
dpar-opt	70.2%	63.2%	10.9%

Results on BC, PageRank and SpMV: Figure 45 shows the performance of *BC*, *PageRank* and *SpMV* using various lb_{THRES} settings. Again, we report the speedup achieved by our code variants over a thread-mapped implementation without load balancing. We make the following observations.

First, similarly to *SSSP*, the speedup decreases as lb_{THRES} increases. This behavior is not surprising: the load balancing threshold determines the number of iterations of the outer loop that are processed in a block-mapped fashion, thus reducing the warp divergence during the thread-mapped phase and the resulting core underutilization. In other words, the lower the value of lb_{THRES}, the more load balancing will take place through block-based mapping. Table 7, which shows how the warp execution efficiency of *dbuf-shared* varies with lb_{THRES}, supports this observation. As can be seen, the lower the value of lb_{THRES}, the higher the warp efficiency (and, consequently, the GPU



(a) *BC* (b) *PageRank* (c) *SpMV*

Figure 45: Speedup of load balancing code variants over basic thread-mapped implementation under different lb_{THRES} settings

Table 7: Warp execution efficiency (*dbuf-shared*)

Applications	lb_{THRES}				
	32	**64**	***256***	***1024***	***baseline***
SSSP	75.6%	71.9%	45.3%	37.2%	35.6%
BC	75.8%	56.7%	17.1%	10.8%	10.3%
PageRank	91.5%	87.0%	63.4%	50.9%	50.8%
SpMV	94.4%	82.3%	71.5%	51.5%	51.0%

utilization). Note that the use of this parallelization template always improves the warp efficiency over the baseline code. We observed similar trends with all other parallelization templates but *dpar-naïve*.

Second, *dual-queue* performs better than the other code variants only on *BC*. *Dual-queue* suffers from the overhead of the initial creation of the two queues. While this overhead is limited for small datasets (e.g. *Wiki-vote* used by *BC*), its negative effect on performance becomes more obvious on large datasets (e.g. *CiteSeer* used by *PageRank* and *SpMV*).

Third, on these applications *dbuf-shared* performs worse than *dbuf-global* for low values of lb_{THRES} and reports better or comparable performance for $lb_{THRES} \geqslant 128$. This trend can be explained as follows. Recall that both parallelization templates use a delayed buffer to identify the large iterations of the outer loop that must be processed in the second, block-mapped phase. Since *dbuf-global* stores this buffer in global memory, the load gets redistributed across the thread-blocks during the second phase of the code (that is, the content of the delayed buffer is partitioned fairly across thread-blocks). However, this load redistribution across thread-blocks does not take place in the case of *dbuf-shared*, which stores the buffer in shared memory and performs a single kernel call. The probability of load imbalance across thread-blocks is higher for low values of lb_{THRES},

since in this case more work is added to the delayed buffer. However, when lb_{THRES} increases, less work is added to the delayed buffer, thus decreasing the probability of work imbalance across thread-blocks in the second phase of the code. When the amount of load balancing is low, *dbuf-global* suffers from the overhead of launching the second kernel. We used profiling data to support this observation. Specifically, we analyzed the warp occupancy of these code variants (that is, the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor). For small values of lb_{THRES}, *dbuf-global* reports a higher warp occupancy compared to dbuf-shared (for example, for lb_{THRES} =32, the warp occupancy of *dbuf-shared* and *dbuf-global* is 18.3% and 26.9%, respectively), indicating a better hardware utilization.

Finally, we observe that *dpar-opt* performs similarly to (or slightly worse than) *dbuf-shared*. The nested kernel handling and launch overhead out shadows the benefit of dynamically remapping the work spawned within nested kernels to the GPU hardware.

4.5 Workload Consolidation

Since the Kepler architecture, Nvidia has introduced in its GPUs a new feature, called *dynamic parallelism* (DP), which makes it possible for GPU threads to dynamically spawn GPU kernels. Dynamic parallelism has also been recently added to the OpenCL 2.0 standard. By supporting nested kernel invocations, this feature enables nested parallelism on GPU, and can facilitate dynamic load balancing, data-dependent execution and the parallelization of recursive algorithms. However, the effective use of DP is not a trivial matter. Basic implementations of adaptive kernels that spawn child kernels on a per-thread basis whenever new work is locally generated tend to perform a large number

of small kernel launches. It has been shown that, due to the runtime overhead associated with nested kernel calls, these implementations often lead to significantly degraded performances, even worse than those of flat parallel variants of the same algorithms [91-93].

In this work, we address this problem and propose a workload consolidation approach to improve the performance of applications relying on DP [121, 122]. Specifically, we consolidate into a single nested kernel the workload belonging to kernels that would be spawned by multiple GPU threads. We consider performing kernel consolidation at three granularities: *warp*-, *block*- and *grid-level*, whereby the consolidation involves the kernels launched by all the threads within a warp, all the threads within a block or the entire grid, respectively. We evaluate our consolidation mechanisms on applications exhibiting two computational patterns: parallel irregular loops and parallel recursion.

4.5.1 Dynamic Parallelism

Traditionally, only CPU threads can launch GPU kernels. Dynamic Parallelism, a feature added to OpenCL 2.0 standard and supported by Nvidia GPUs with compute capability 3.5 and above, makes it possible for GPU threads to launch GPU kernels. Kernel launches can be nested and the deepest nesting level supported is currently 24. Kernels launched from different blocks or streams can execute concurrently, and up to 32 concurrent kernels are currently allowed on Nvidia GPUs. A parent kernel will return only after all its child kernels have completed; however, the order of their execution is unknown unless these kernels are explicitly synchronized by a *cudaDeviceSynchronize* call. For each nesting level up to an explicit synchronization, parent kernels may be

temporarily swapped out to free up GPU resources and allow the execution of their child kernels. Pending kernels due to either unresolved dependencies or lack of available hardware resources are fed to a temporary buffer. Global memory data are visible to both parent and child kernels, while shared and local memory variables are visible only within the kernel where they are declared.

4.5.2 Application Characterization

We consider two computational patterns that can benefit from DP: irregular loops and parallel recursion.

Irregular Loops - Irregular loops are characterized by an uneven work distribution across loop iterations. For example, nested loops where the number of iterations of inner loops varies across the iterations of outer loops are irregular. These loops can be found in many applications, such as sparse matrix operations and graph traversal algorithms that rely on the commonly used Compressed Sparse Row representation of matrix and graph data structures [75]. The degree of parallelism within irregular loops is typically data dependent [123-125] and known only at runtime. Due to their nature, the flat parallelization of these loops can cause work imbalance across threads, possibly leading to GPU underutilization and limited performance. For example, if loop iterations are distributed equally to threads, the unbalanced work distribution across iterations (and threads) will lead to warp divergence. Irregular loops can benefit from the use of DP for load balancing [126]. Specifically, overloaded threads can spawn child kernels and assign (part of) their work to them, thus limiting warp divergence.

Parallel recursion - Nested parallelism also arises in the presence of parallel recursion. While some recursive algorithms can be made iterative through various code transformation techniques (e.g. tail-recursion elimination, auto-ropes and other flattening techniques [127-129]) and subsequently undergo flat parallelization, recursion cannot be always eliminated. Before the introduction of DP on GPU, parallel recursion required both CPU and GPU intervention. Specifically, the CPU would control the flow of recursion and initiate the recursive calls, whose execution could then be offloaded to the GPU in the form of parallel kernels. This approach requires one kernel launch for every recursive call and incurs high CPU-GPU communication overhead. By enabling kernel launches from GPU threads, DP allows the recursive control flow to reside directly on the GPU. This, in turn, allows reducing the CPU-GPU communication and the kernel launch overhead. The most natural way to implement a parallel recursive function on GPU is by directly porting to this platform a CPU parallelization of that function and allowing each GPU thread to spawn a recursive kernel whenever the CPU code would perform a recursive call. As we will show, GPU implementations following this pattern often perform a large number of small kernel invocations, leading to substantial kernel launch overhead and hardware underutilization. Our proposed consolidation schemes target this problem.

4.5.3 Motivation

In this section, we first present the basic use cases of dynamic parallelism. Then we show how inefficient the basic implementation is and explain why using DP can lead to

performance degradation. These motivate us to propose our compiler-assisted workload consolidation solution.

A. Basic DP-Code Template

Figure 46(a) shows a basic code template for parallel kernels that use DP [93]. As in all GPU kernels, each thread (or thread-block) is assigned some data to process (or *work items*). Each thread (thread-block) initially performs some work (*prework*) on the data assigned to it. Then, depending on the outcome of a condition, the thread (thread-block) either spawns a child kernel to execute some more work, or performs that work on its own. Finally, the thread (thread-block) may optionally perform additional work (postwork). Note that both irregular loops and recursive algorithms fit in this code template. The only difference between these two computational patterns is the following: in irregular loops parent and child kernels are different, and the child kernel is generally used for load balancing purposes; in parallel recursion, parent and child kernels are identical. Figures 46(b) and (c) illustrate this basic code template on two algorithms: one with an irregular loop (*single source shortest path*), and the other with parallel recursion (*recursive tree traversal*).

```
__global__ void parent_kernel() {
   work_item= get_work_item(…)
   prework(work_item)
   if (condition)
      child_kernel<<<block_dim, thread_dim>>(…, work_item, …)
   else
      work(work_item)
   postwork(work_item)
}
```

(a) Basic code template for kernels using dynamic parallelism

```
__global__ void sssp() {
   int i = blockDim.x * blockIdx.x + threadIdx.x
   neighbors = get_neighbors(i)
   if (neighbors.size>THRESHOLD)
      process_neighbors<<<block_num, thread_num>>>(neighbors)
   else {
      work(neighbors)
   }
   postwork()
}
```

(b) Basic implementation of SSP with dynamic parallelism

```
__global__ void tree_traversal( node_t node) {
   int i = threadIdx.x
   node_t child = node.children[i]
   if (child.children.size>0)
      tree_traversal<<<1, child.children.size>>>(child)
   else {
     leafnode_work()
   }
   postwork()
}
```

(c) Basic code of tree traversal with dynamic parallelism

Figure 46: Basic-dp code template and sample codes

In the *SSSP* kernel, each GPU thread processes the neighbors of an assigned node. Because the number of neighbors may vary from node to node, the workload may be unevenly distributed across threads. To address this problem, each thread checks whether the amount of work assigned to it (*neighbors.size*) is larger than a given threshold. If this is the case, it spawns a child kernel and delegates the work to it; otherwise it performs the work on its own. Because the GPU hardware schedules different kernels independently, this mechanism allows redistributing the work to the GPU resources, potentially leading to better GPU utilization.

In the tree traversal kernel, each thread is assigned a *child* of a given node. Initially, the thread checks whether child has no children. In this (base) case, the thread performs *leafnode_work*; otherwise, it spawns a kernel recursively and delegates to it the processing of its assigned node.

The examples above illustrate “basic” implementations of irregular loops and parallel recursion that rely on DP. For irregular loops, DP is used to redistribute unbalanced work:

in this case, the thread (thread-block) executing an iteration of the loop invokes a nested kernel to offload the excess work to it. In case of parallel recursion, this basic code variant results from simply porting the CPU implementation of a parallel recursive function to GPU. Although more complex implementations are possible, these basic code variants require minimal effort from the programmer. However, as discussed below, this basic use of DP can incur significant overhead and lead to poor performance.

B. Limitations of Dynamic Parallelism

The effectiveness of DP is affected by different factors: sources of runtime overhead and GPU utilization. Below, we detail each of these aspects.

Kernel Launch Overhead - To launch a kernel, the CUDA driver and runtime need to parse the parameters list, buffer the values of these parameters, and dispatch the kernel. These steps have an associated overhead. This overhead is negligible when the number of nested kernels is small, but can accumulate and become significant in the presence of numerous kernel launches [92, 93].

Kernel Buffering Overhead - Kernels waiting to execute are inserted in a pending buffer. Since CUDA 6, this buffer consists of two pools: a fixed-size pool and a variable-size virtualized pool. The fixed-size pool incurs lower management overhead but by default can only accommodate a maximum of 2048 pending kernels. When it becomes full, pending kernels are fed to the virtual pool, which incurs extra management overhead. Applications spawning a large number of nested kernels can exhaust the fixed-size pool and experience performance degradation due to the virtual pool's overhead. It is possible to increase the capacity of the fixed-size pool through the CUDA function *cudaDeviceSetLimit*. However, this will result in a higher global memory reservation.

Synchronization Overhead - If there exists explicit synchronization between parent and child kernels, in order to free resources for the execution of child kernels, parent kernels will be swapped out into global memory. For each level up to the maximum level where they synchronize, up to 150 MB memory may be reserved for swapping. These extra memory transactions are source of additional overhead.

Effect of the kernel configuration – It is well known that the full use of the GPU hardware requires massive multithreading. CUDA currently allows up to 32 kernels to execute concurrently on a GPU. However, if configured to use small thread configurations, even 32 concurrent kernels may underutilize the GPU. Meanwhile, there is a limit on the maximum number of blocks that can be concurrently active. Thus, configuring nested kernels with a large number of blocks will limit the number of kernels executing in parallel. As a result, it is important for programmers to carefully select thread configurations that allow both good device utilization and desired level of concurrency.

The kernel launch, buffering and synchronization overheads can be reduced by limiting the number of kernel launches performed. In addition, to avoid GPU underutilization, it is important to avoid small kernel configurations that would lead to low occupancy even in the presence of kernel concurrency. In general, DP codes resulting in a large number of small kernel invocations tend to experience poor performance. In our previous work [92], we have shown that, due to the large number of small kernel calls they invoke, DP codes following the basic template in Figure 1 can underperform flat implementations of the same algorithms by up to a factor of 1000.

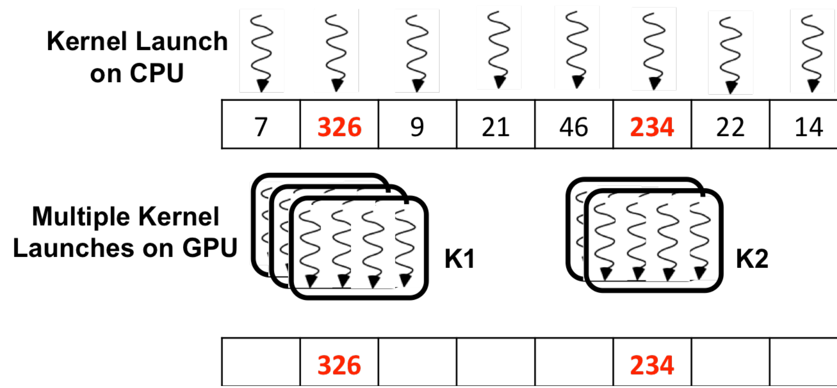
4.5.4 Methodology

In this section, we present our workload consolidation mechanism designed to avoid the performance degradation associated with the basic use of dynamic parallelism described in Chapter 4.5.3.

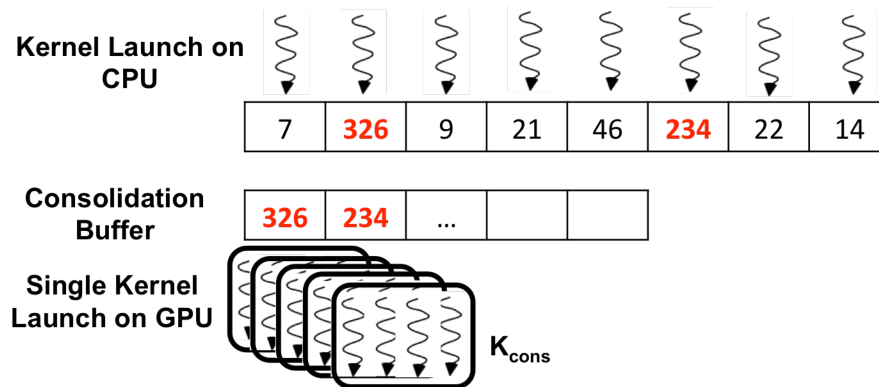
A. Workload Consolidation

The idea at the basis of workload consolidation is fairly simple: by aggregating kernels spawned by different threads into a single or few consolidated kernels, it is possible both to decrease the number of nested kernel invocations, thus limiting DP overhead, and to increase the degree of multithreading of the nested kernels invoked, thus increasing their GPU utilization. In order to perform workload consolidation, we buffer the work associated to the kernels to be consolidated, and we defer the handling of this aggregated work to the launch of one or more child kernels. Since in the SIMT model all threads execute the same instructions on different data, in order for a thread to buffer work, it will be sufficient for the thread to buffer the pointer(s) or index(es) to the data to be processed. This method requires barrier synchronization between the buffer insertions and the consolidated kernel launches.

This high level idea is illustrated in Figure 47. In the figure, the numbers in the array indicate the amount of work assigned to each thread, and the numbers in red indicate large workloads that need to be redistributed through the launch of nested kernels. In Figure 47(a), two child kernels (K_1 and K_2) are invoked: one to process 326 and the other to process 234 work items. Workload consolidation, illustrated in Figure 47(b), first inserts the work associated to K_1 and K_2 into a consolidation buffer. It then launches a



(a) Basic implementation of dynamic parallelism



(b) Dynamic parallelism with workload consolidation

Figure 47: Workload consolidation – illustration

single child kernel (*Kcons*) to process all the work in the buffer. *Kcons* will have a larger thread configuration than K_1 and K_2.

B. Consolidation Granularity

CUDA programming model has four levels of parallelism: thread, warp, block and grid. While threads, blocks and grids are exposed to programmers, warps are implicitly defined as groups of 32 threads that execute in lockstep and have contiguous identifiers. In the DP execution model, kernel launches are performed by threads. We consider three consolidation granularities: warp-, block- and grid-level.

Warp-level consolidation uses a buffer to aggregate work from the threads within a warp and launches one kernel per warp. This consolidation method can reduce the number of kernel invocations at most by a factor of 32. The benefit of warp-level consolidation is that the synchronization overhead is minimized because no additional synchronization is required beside the implicit one due to SIMD execution.

Block-level consolidation aggregates work associated to threads within a block and launches a kernel per block. This method can reduce the number of kernel invocations beyond what allowed by warp-level consolidation. However, this consolidation scheme requires a block-level synchronization (__syncthreads) after the buffer insertions, leading to higher synchronization overhead than the warp-level variant.

Grid-level consolidation aggregates work from all threads in a grid and then launches a single child kernel. Since CUDA does not provide global synchronization within a kernel, this consolidation method requires a customized barrier synchronization (we will discuss this aspect later). Because of this, grid-level consolidation suffers from the highest synchronization overhead.

C. Kernel Transformations

The overall kernel transformation flow is shown in Figure 48. The input is DP-based CUDA code annotated with the described pragma directive, and the output is the consolidated CUDA code. The kernel transformation process consists of two steps: (1) child kernel and (2) parent kernel transformation. For irregular loops, parent and child kernels are different, and the two code transformations are applied separately to each. For recursive computations, parent and child kernels are the same, and the two transformation steps are applied to the single input kernel sequentially.

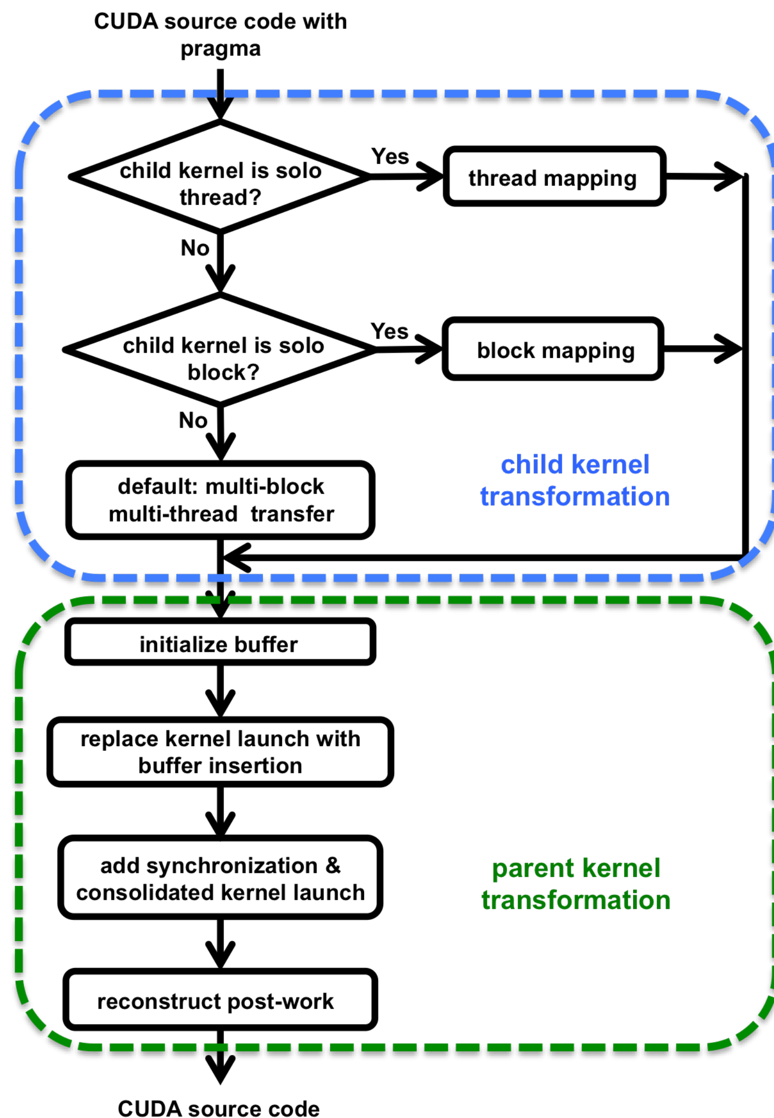


Figure 48: Kernel transformation flow

Child kernel transformation – This phase transforms the input child kernel into a consolidated child kernel. The new kernel fetches work from the consolidation buffer and processes that work according to the code in the input child kernel. Whenever possible, we generate moldable kernels [130] (that is, kernels with tunable kernel configuration). The way the original code is mapped to threads and blocks in the consolidated kernel

depends on the configuration of the input child kernel. Specifically, we identify the following cases:

Solo thread: The input child kernel consists of a single block and a single thread (e.g. quick sort in CUDA SDK). In the consolidated child kernel, each thread will fetch a work item (if available) from the consolidation buffer, and it will process that work item exactly as the original kernel does. To make the kernel moldable, we allow threads to fetch work from the buffer repeatedly until the buffer becomes empty.

Solo block: The input child kernel consists of a single block with multiple threads, and these threads operate cooperatively. In the consolidated child kernel, each block will fetch a work item (if available) from the consolidation buffer, and the threads within the block will cooperatively process that work item as in the original kernel. To make the kernel moldable, we allow blocks to fetch work from the buffer repeatedly until the buffer becomes empty. If the original child kernel is moldable, the number of threads per block in the generated child kernel will also be tunable; else, the two kernels will have the same block size.

Multi-block and *multi-thread*: When the original child kernel uses multiple blocks and threads per block, each work item is processed by all threads cooperatively. In this case, in the transformed kernel we use a for-loop to wrap the code of the original child kernel. The generated kernel will extract work from the buffer iteratively, and all threads will process cooperatively each work item. In this case, the transformed kernel is moldable only if the original kernel is such.

Parent kernel transformation – We divide the parent kernel into three sections: *prework*, *child kernel launch*, and *postwork* (Figure 46(a)). The prework and postwork

represent the processing done before and after the child kernel launch, respectively. Many kernels do not include any postwork. The code transformations required in the parent kernel are: (1) buffer allocation (before prework); (2) prework insertion; (3) replacement of the child kernel launch with buffer insertions; (4) insertion of the required barrier synchronization primitive; and (5) postwork transformation. As shown in Figure 47, steps (1)-(3) are fairly mechanic; however, steps (4) and (5) require some consideration.

If the original kernel includes barrier synchronization between the child kernel launch and the postwork, such synchronization must be preserved in the consolidated parent kernel. For warp-level consolidation, this problem is automatically solved by the implicit barrier synchronization due to the lockstep execution of the threads within a warp. For block-level consolidation, CUDA provides a block-level barrier synchronization primitive (*__syncthreads*). Grid-level consolidation requires more thought. First, the only global barrier synchronization provided by CUDA is the implicit one at the end of a kernel launch. However, using this mechanism would require splitting the parent kernel into two: a kernel handling the *prework* and the child kernel launch, and a kernel handling the postwork. In addition, CPU intervention would be required to invoke the postwork-kernel. This would be problematic in case of recursion, as it would require returning the control to the CPU after each child kernel launch, and to have calls to the postwork-kernel stacked on CPU. In other words, the CPU would acquire full recursion control, leading to the overheads discussed in Chapter 4.5.3. To address this problem, we implement a custom global synchronization mechanism that can be invoked from the GPU (see Chapter 4.5.5.E). Second, a global synchronization may cause a deadlock when active blocks on GPU are suspended at the global barrier while pending blocks are

waiting for active blocks to finish. To address this problem, we consolidate the postwork into a single kernel. The last block to complete its buffer insertions will then launch the consolidated child kernel, wait for its completion and then launch the consolidated postwork kernel. The other blocks will simply exit after completing their buffer insertions. Finally, the required barrier synchronization between the child kernel launch and the postwork is handled by inserting a *cudaDeviceSynchronize* call between the invocations of these two consolidated kernels. Dependencies between the prework and the postwork are handled by duplicating in the postwork the relevant portions of prework.

D. Compiler Directive Design

In order to direct the code transformations performed by the compiler, we provide a single directive that can be applied to generic DP-based code following the template in Figure 46. This directive allows identifying the child kernels to be consolidated and the work to be buffered. The grammar of the directive is: *'#pragma dp [clause+]'*. Table 8 lists the clauses available, which specify the consolidation granularity, the type and size of the consolidation buffer, the indexes or pointers to the work items to be buffered and

Table 8: Clauses of our workload consolidation compiler directive

Clause	Argument	Description	Optional
consldt	granularity: *warp*, *block*, *grid*	Workload consolidation granularity	No
buffer	type: *default*, *halloc*, *custom*	Buffer allocation mechanism	Yes
	perBufferSize: integer or variable name	Buffer size	
	totalSize: integer	Total size of all buffers	
work	varlist: list of indexes or pointers to work	List of variables to be stored in buffer	No
threads	thread number: integer	Number of thread/block for consolidated kernel	Yes
blocks	block number: integer	Number of blocks for consolidated kernel	Yes

the configuration of the consolidated kernel. Some of these clauses are optional and programmers can use them for further optimization and performance tuning. For instance, developers can optionally specify the configuration of the consolidation buffers and the one of the child kernels. We provide more details on these options in Chapter 4.5.4.E.

Figure 49(a) illustrates the use of our proposed compiler directive to annotate the original CUDA code. In this case, block-level consolidation is selected, the buffer can have at most 256 elements and is instantiated with the default CUDA allocator, and variable curr is buffered. The generated code is shown in Figure 49(b) (in this particular case, the synchronization primitive in use is __synchthread).

E. Implementation Details

This section describes the implementation details of our source-to-source compiler, which converts annotated CUDA code into consolidated GPU kernels. Our compiler is implemented using the ROSE compiler infrastructure [131].

Rose compiler infrastructure – ROSE (version 0.96.a) incorporates Edison Design Group (EDG) Frontend 4.0 that supports the parsing of CUDA, C and C++ code. We use its parser building APIs to implement the parser for the pragma directive. The pragma information is linked to the abstract syntax tree (AST). Based on the directive information and the AST, we customize the traversal and transformation functions to generate a new AST, which is then fed to and unparsed by the backend of ROSE to generate the consolidated parent and child kernels.

Consolidation Buffers – The design of the consolidation buffers involves several considerations, some of them leading to the need for the directive clauses listed in Table I.

```
__global__ void parent_kernel() {
   work_item = get_work_item(…)
   prework(work_item)
   if (condition) {
   #pragma dp consldt(block) buffer(default, 256) work(work_item)
     child_kernel<<<block_dim, thread_dim>>(…, work_item, …)
   } else  work(work_item)
   postwork(work_item)
}
```

(a) Annotated CUDA code (parent kernel)

```
__global__ void parent_kernel() {
   work_item = get_work_item(…);
   prework(work_item)
   if (condition)  insert_buffer(curr)
   else  work(work_item)
   synchronize
   if (thread_id==selected)
     child_kernel_consolidate<<<b_dim_con, t_dim_con>>>()
   synchronize
   postwork(work_item)
}
```

(b) Generated CUDA code (parent kernel)

Figure 49: Example of use of our workload consolidation compiler directive

Memory selection: The consolidation buffer can be either implemented in global or in shared memory. Global memory provides slower access but is visible to both parent and child kernels. Conversely, shared memory is faster but private to each block (and thus not visible within nested kernels). While parent kernels could fill temporary buffers in shared memory and then copy them into global memory to allow access by child kernels, the limited size of shared memory makes this solution not scalable. As a result, we store the consolidation buffers solely in global memory.

Dynamic allocation method: For the allocation of the consolidation buffers, we allow three alternatives: (1) the default allocator provided by CUDA; (2) the open-source halloc memory allocator for GPU [132]; and (3) a customized allocator that leverages a pre-allocated memory pool.

Buffer size for customized allocator: Due to the irregular nature of nested parallelism, the buffers required by different consolidated kernels may vary in size. When using the pre-allocated memory pool, the programmer needs to set both its size and the size of the portion of the memory pool allocated to each buffer (recall that in warp/block-level consolidation every warp/block uses a consolidation buffer). The size of the pre-allocated

memory pool (500MB by default) can be specified using the *totalSize* argument in the pragma directive. The per-buffer size (*perBufferSize*) is predicted as:

$$totalThread * totalBuffVar * const$$

where *totalThread* is the total number of threads from which we consolidate the child kernels, *totalBuffVar* is the number of buffered variables (indexes or pointers) per work item; and *const* is a constant (default value: 4) that estimates the number of work items assigned to a single thread. We have observed that, in most cases, the *perBufferSize* can be determined from a runtime variable that indicates a property of the current work item. For instance, for the tree applications in our benchmarks, the buffer size can be derived from the variable that indicates the number of children of a given node. If the user cannot provide such variable, a constant may also be specified to its best estimation. Our customized allocator can utilize the information from the *#pragma* to allocate properly sized buffers from the pre-allocated global memory pool for different consolidation granularities. Notice that for grid level consolidation, only one buffer is required for each grid: in this case, the grid can directly utilize the whole memory pool and the *perBufferSize* is ignored.

Global Barrier Synchronization on GPU – The global barrier synchronization is implemented using a counter whose value is initialized to the number of blocks executed. When a block reaches the barrier, the first thread in the block decrements the counter by one atomically. A counter decrement to zero indicates that the last block has reached the barrier.

Kernel Configuration Handling – When launching kernel calls, it is common wisdom to select a configuration that achieves high device occupancy, which is defined

as the ratio of the number of active warps to the number of maximum active warps that the device can host. Although higher occupancy does not always guarantee higher performance, it usually produces a good enough result that can be further tuned. The use of the *CUDA Occupancy Calculator* allows finding a kernel configuration (*B*, *T*) that maximizes the occupancy for a single kernel. However, concurrent kernels must share the GPU resources, and thus such configuration will not be optimal for concurrent kernels initiated with DP [130]. To allow multiple kernels to be concurrently active on GPU, programmers need to downgrade the configuration they obtain by using the Occupancy Calculator. We refer to Kernel Concurrency (*KC*) as the maximum number of concurrently active kernels. The highest *KC* supported by CUDA as of compute capability 3.5 is 32. Due to the hardware resource limitations, a concurrency of X may be achieved by downgrading the configuration (*B*, *T*) to ([*B*/*X*], *T*). We name such configuration *KC_X*.

For grid-level consolidation, at any given time there is only one active consolidated kernel that processes all the work from all threads in the parent kernel. Hence, we expect the best configuration to be the one that maximizes the device occupancy for a single kernel. Thus, we use *KC_1* as the default kernel configuration.

For block-level consolidation, each block in the parent kernel will spawn a consolidated kernel that will handle a smaller amount of work collected from a block in the parent kernel. We decide to downgrade the configuration by a factor of 16 and use *KC_16* as the default configuration.

For warp-level consolidation, each warp in the parent kernel will spawn a consolidated kernel that handles an even smaller amount of work collected from a warp,

and a maximum concurrency of 32 can be easily achieved. Thus, we use *KC_32* as the default kernel configuration.

Because users may need to fine-tune the configuration for their applications to achieve the best performance, we also provide *#pragma* clauses that allow for user-specified kernel configurations.

4.6.4 Experimental Evaluation

In this section, we evaluate our proposed workload consolidation methods on various applications and datasets. Specifically, we compare the generated consolidated kernels with the original, basic DP kernels (*basic-dp* code template in Figure 46) and with flat GPU implementations of the considered algorithms (denoted by *np-dp*). In all the figures, *warp-level*, *block-level* and *grid-level* refer to the consolidation granularity considered.

We first evaluate the performance of using different memory allocators to implement the consolidation buffers. We then evaluate the effectiveness of our method to select the kernel configuration of nested kernels by comparing the resulting performance with the best performance achieved using the optimal kernel configuration found by exhaustive search. We then study the overall performance of the different consolidated kernels using the optimal allocator and kernel configuration. Finally, we use profiling to study the effect of our consolidation schemes on hardware utilization.

Hardware and Software Setup: We run all experiments on a workstation powered by two 6-core Xeon E5-2620 CPUs and a NVIDIA K20c GPU. We use CUDA runtime and compiler version 7.0. We use Nvidia Visual Profiler to collect the profiling data. We average the results of the experiments over multiple runs.

Benchmark implementations: The benchmarks used in our evaluations are *Single Source Shortest Path* (SSSP), *Sparse Matrix Vector Multiplication* (SpMV), *PageRank* (PR), *Graph Coloring* (GC), *Recursive Breadth-first Search* (BFS-Rec), *Tree Heights* (TH) and *Tree Descendants* (TD). Specifically, we consider the *basic-dp* and *no-dp*/flat implementations from [75, 92, 117, 118]. We use *basic-dp* implementations as baseline, and report the performance of flat kernels (*no-dp*) and consolidated kernels with different consolidation granularities. Since flat kernels achieve better performance than CPU implementations, we do not report results of CPU implementations.

Datasets: For applications based on graphs and sparse matrices, the datasets used are *CiteSeer* (used in SSSP, SPMV, PG) and *Kron_log16* (used in GC, BFS-Rec), both from the DIMACS challenges [133]. *CiteSeer* is a paper citation network with about 434 thousand nodes, 16 million edges and a node outdegree that varies from 1 to 1,199 across the graph (with an average value of 73.9). *Kron_log16* has 65 thousand nodes and 5 million edges, with a node outdegree that varies from 8 to 36,114. For the trees, we use datasets from [92]: dataset1 is a depth-5 tree whose nodes have a number of children varying from 128 to 256 and only half of the non-leaf nodes have children; dataset2 is a depth-5 tree whose nodes have a number of children varying from 32 to 128 and all non-leaf nodes have children.

A. Implementation of the Consolidation Buffers

As explained in Chapter 4.5.4.E, the consolidation buffers can be implemented using three allocators: the default CUDA *malloc* allocator, the open-source high performance *Halloc* allocator [132] and our customized allocator. Figure 50 shows the performance results of workload consolidation on SSSP using these three allocators. In the figure,

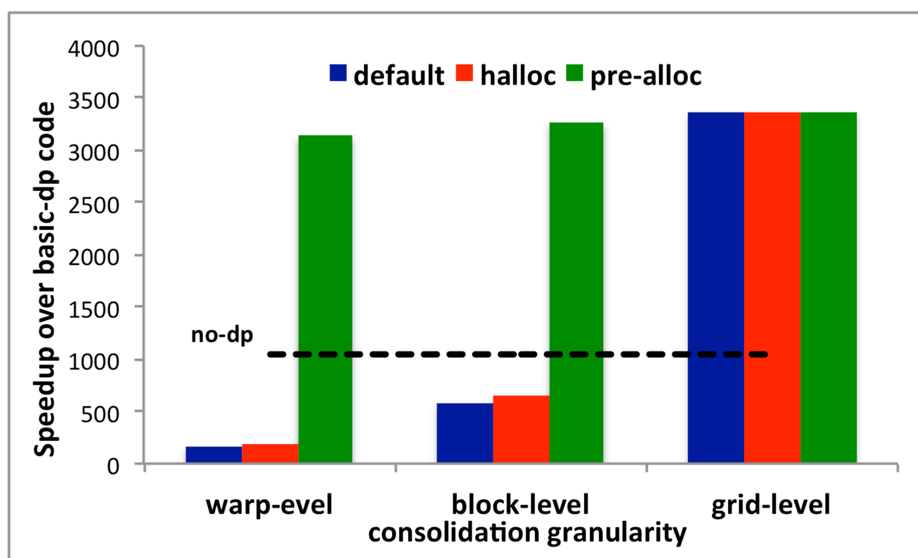


Figure 50: Performance of different buffer implementations (SSSP)

default refers to the CUDA *malloc/free* primitives, halloc to the *Halloc* allocator, and *pre-alloc* to our customized allocator. We can see that the *default* and *halloc* allocators achieve similar results in all cases. For block-level consolidation, the performance of both default and halloc are worse than that of the flat GPU code (*no-dp*), while *pre-alloc* achieves roughly 3x speedup over *no-dp*. This is due to the higher overhead introduced by *default* and *halloc* on every allocation operation. This overhead also contributes to a 5.7x performance gap between *pre-alloc* and *default/halloc* in case of block-level consolidation. The performance degradation of default and *halloc* is even worse (20x slowdown) for warp-level consolidated code, which requires more frequent buffer allocation operations. Since grid-level consolidation only requires a single buffer, in this case there is no obvious performance difference among these three allocators. In the remaining experiments, we only show the results reported using the better performing *pre-alloc* allocator.

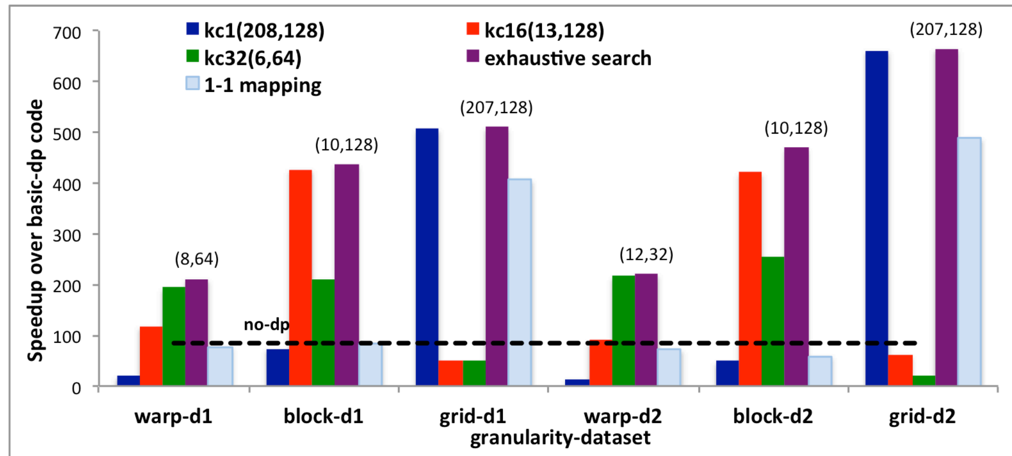


Figure 51: Performance of different kernel configurations (TD)

B. Selection of the Kernel Configuration

In Chapter 4.5.4.E we discuss three configurations for consolidated kernels: *KC_1*, *KC_16* and *KC_32*. These configurations allow the consolidated kernels to achieve concurrency levels (maximum number of concurrently active kernels) of 1, 16 and 32, respectively. We compare these configurations with two additional configurations schemes, *1-1 mapping* and *exhaustive search*. The *1-1 mapping* configuration indicates that the kernel is configured to have as many blocks (or threads, in the case of thread-mapped child kernels) as the number of items in the buffer. The *exhaustive search* configuration is the best configuration we find from exhaustively searching the configuration space [134]. In Figure 6 we report the results on tree descendants for all considered consolidation granularities over two datasets.

We first compare the three proposed configurations. *KC_1* works best for grid-level consolidation; *KC_16* works best for block-level consolidation; and *KC_32* works best for warp-level consolidation. This meets our expectations and is coherent with the analysis presented in Chapter 4.5.4.E. We then compare *KC_1* for grid-level, *KC_16* for

block-level and *KC_32* for warp-level consolidation with *1-1 mapping*. As can be seen, our solution performs much better, especially for block- and warp-level consolidation. This is because the varying block size of *1-1 mapping* lowers the Kernel Concurrency and increases the size of the pending queue, leading to higher overhead and degraded performance. At last, we compare our scheme with the best configuration found by *exhaustive search*. We observe that the configurations selected by our method (*KC_1* for grid-level, *KC_16* for block-level and *KC_32* for warp-level consolidation) achieve on average 97% of the performance of the optimal configuration found by exhaustive search.

The same experiments conducted on the other benchmarks using various datasets report similar results. In conclusion, our configuration selection method for nested kernels is effective and leads to nearly optimal performance. In all the remaining experiments, we use *KC_1*, *KC_16* and *KC_32* configurations for kernels consolidated at the grid-level, block-level and warp-level granularities, respectively.

C. Overall Performance

Figure 52 presents the overall speedup of kernels consolidated at different granularities over *basic-dp*. The chart also reports the speedup achieved by flat parallel code (*no-dp*) over the baseline. As can be seen, the *basic-dp* implementation suffers from severe performance degradation due to the significant kernel management overhead and the limited GPU utilization. Even compared with the flat GPU kernels, *basic-dp* reports slowdown factors from 80 to 1100. Warp-level consolidation improves the performance of *basic-dp* on average by a factor of 1000x but in some cases is not significantly better than the flat GPU kernel (*no-dp*). Block-level consolidation outperforms warp-level consolidation, and grid-level consolidation achieves the best performance across all

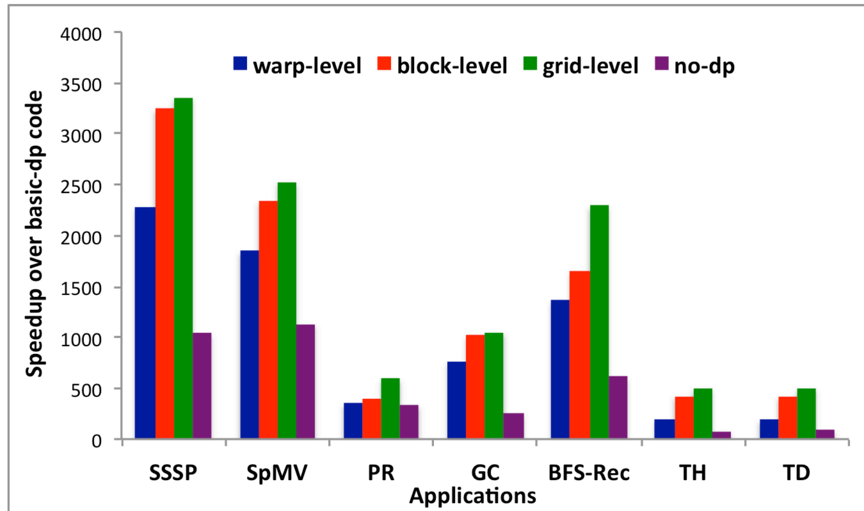


Figure 52: Overall speedup over basic dynamic parallelism

benchmarks. Even if warp-level consolidation has the benefit of very low synchronization overhead, when compared to block- and grid-level code, it suffers from the more significant overhead introduced by the additional child kernel launches. Grid-level consolidation reduces the number of child kernel launches dramatically, and thus achieves the best performance despite its extra synchronization overhead. On average, warp-level, block-level and grid-level consolidation outperform *basic-dp* by a factor of 999, 1357 and 1459, respectively, and *no-dp* by a factor of 2.18, 3.26 and 3.78, respectively.

D. Profiling Results

In this section, we analyze the improvements in hardware utilization achieved by workload consolidations.

Figure 53 shows the overall *warp execution efficiency*, which is defined in the CUDA documentation [135] as "the ratio of the average active threads per warp to the maximum number threads per warp". For each application, we show the results reported by the basic-dp implementation and the three considered workload consolidation schemes. On

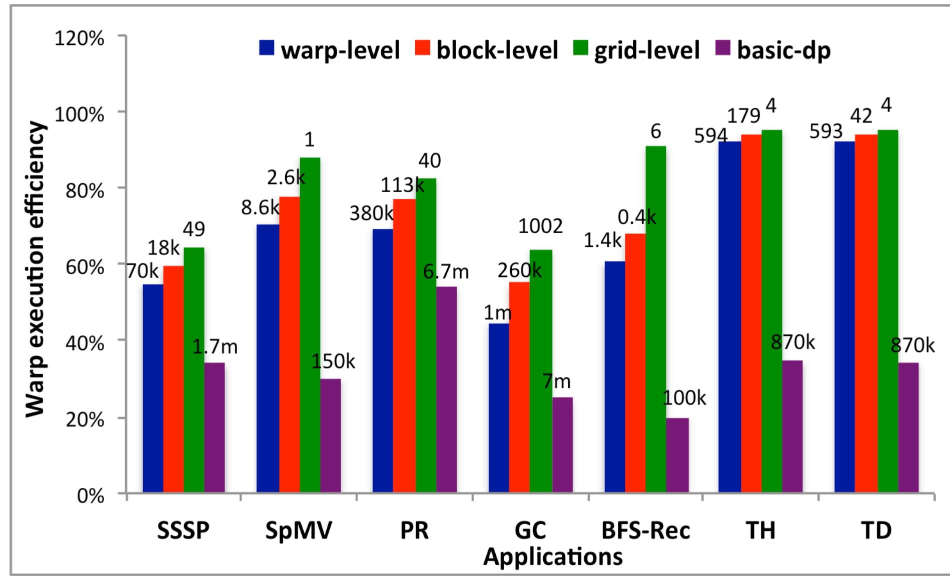


Figure 53: Warp execution efficiency across benchmarks

top of the bars we report the number of child kernel invocations performed in each case. First, we can observe that the proposed consolidation methods reduce the number of kernel invocations to 0.07%-14.48% of the ones performed by the corresponding basic-dp code. For instance, in PageRank, consolidation reduces the number of kernel invocations from 6.7 million (*basic-dp*) to 380 thousand (*warp-level*), 113 thousand (*block-level*) and 40 (*grid-level*). Second, average warp execution efficiencies are improved from 33.2% (*basic-dp*) to 69.3% (*warp-level*), 75% (*block-level*) and 83.1% (*grid-level*). The warp execution efficiency measured by Nvidia profiler includes not only parent and child kernels execution, but also child kernel launch overhead. Child kernel launches will take more clock cycles than buffer insertion operations, decreasing the warp efficiency. Consolidation replaces kernel launches with buffer insertions; as a result, it reduces the overhead of warp divergence and leads to improve overall warp efficiency.

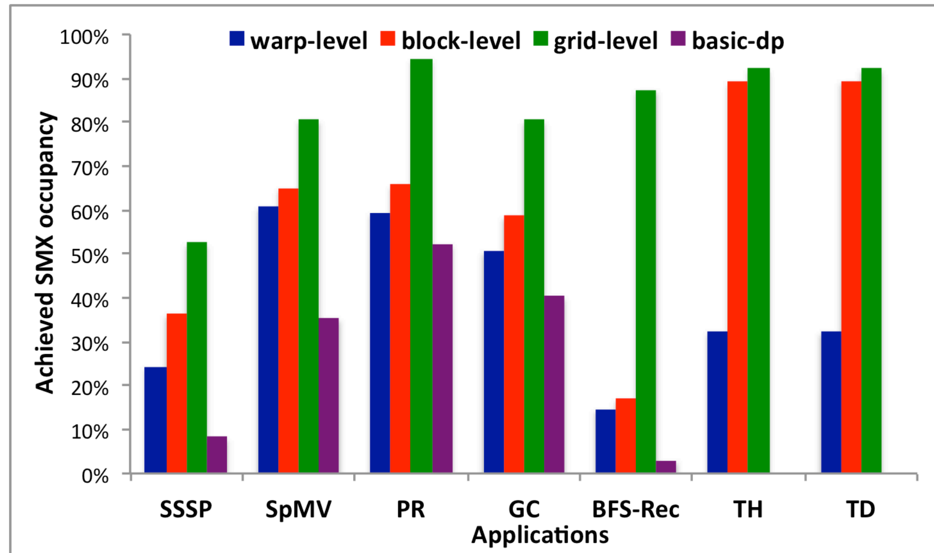


Figure 54: SMX occupancy (achieved hardware utilization)

Figure 54 shows the achieved streaming multiprocessor occupancy, which measures the ratio of average active warps over maximum warps supported per streaming multiprocessor [135]. On average, workload consolidation improves this metric from 27.9% (*basic-dp*) to 39.3% (*warp-level*), 60.3% (*block-level*), and 82.9% (*grid-level*). Recall that in *basic-dp* each thread launches a "small" kernel. Hence, the GPU device is filled with many such "small" concurrent kernels. As mentioned in Chapter 3.5.3.B, there is a hardware limitation on the maximum number of concurrent kernels that a GPU can accommodate. On K20c, this limit is 32. Therefore, in the *basic-dp* case, thirty-two "small" concurrent kernels will underutilize the hardware. Workload consolidation, on the other hand, increases the average child kernel size and improves resource utilization.

To measure the efficiency of memory accesses, we monitor the numbers of DRAM transactions (read+write) performed by each kernel. Figure 55 shows the ratio between the number of DRAM transactions performed by each consolidated kernel and those performed by the *basic-dp* code. In all cases, the total DRAM transactions are reduced. Specifically, *warp-level*, *block-level* and *grid-level* consolidation lead to 60%, 34% and 36% of the original transactions, respectively. This reduction in memory transactions can be motivated as follows: first, the consolidation increases the average child kernel size, thus leading to better caching behavior and memory bandwidth utilization; second, a decrease in the number of nested kernels will lower the chance of swapping parent kernels out, therefore reducing the memory transaction overhead associated to kernel swapping; third, the decreased number of nested kernels reduces the use of the virtualized pool within the pending queue, lowering the overhead of virtual pool management. It can also be noticed that, for some benchmarks (e.g. *SpMV*), block-level achieves better memory utilization than grid-level consolidation. This is due to the overhead associated to our global synchronization mechanism.

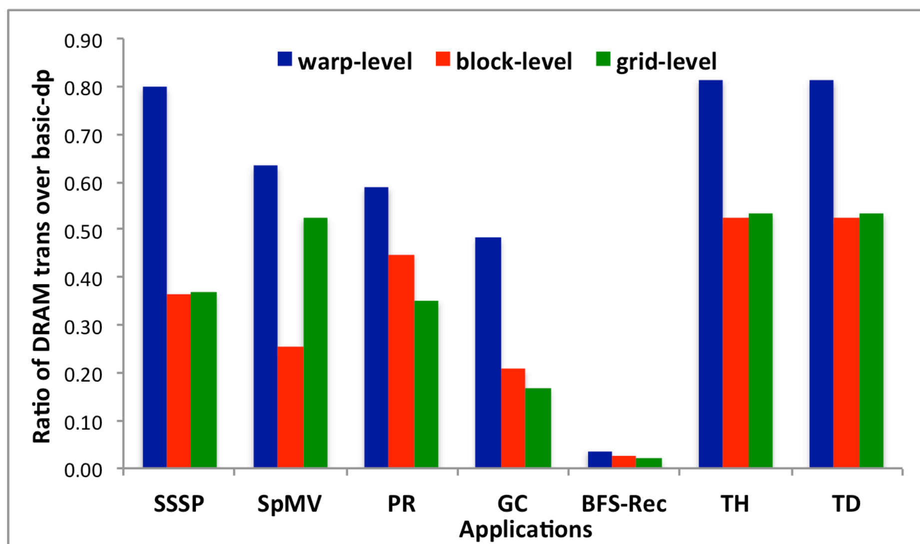


Figure 55: DRAM transactions ratio over basic dynamic parallelism

Chapter 5 Deep Neural Network on CPU-GPU

Deep learning is a branch of machine learning that uses a layered architecture of data processing stages for pattern recognition. Due to its effectiveness in many applications, deep learning has gained popularity in both academia and industry. Convolutional neural networks (CNNs) are the most successful models for deep learning, and they have been used in various domains, including computer vision [136] and speech recognition [137].

5.1 Related Work

With the emergence of powerful GPUs and the availability of large data sets for training, we have witnessed a significant improvement of deep CNNs in terms of training time and accuracy. The Visual Geometry Group (VGG) at University of Oxford has designed a 16- and a 19-layer model with a 7.4% and a 7.3% top-5 error rate, respectively [138]. Microsoft has recently proposed a NN model with 152 layers – 8x deeper than the VGG nets – reporting a 3.57 % error rate [139]. This fast development has led to the proliferation of applications based on NNs. Examples of emerging applications based on NNs include: auto tagging [140], the estimation of a person's pose [141], and the generation of a descriptive sentence from an image [142]. While previous work has focused on performance, our paper focuses on evaluating the energy efficiency of deep neural networks on CPUs and GPUs.

Although the mainstream approach for training deep convolutional neural networks is using CPUs and GPUs, researchers have recently started to explore the use of other architectures and devices, including FPGA [143, 144], RRAM [145], neuromorphic

processors [146] and Tetra-Parallel architecture [147]. These hardware implementations are specifically tailored to convolutional neural networks and yield impressive results in terms of performance and energy efficiency. However, these hardware innovations are still at an early stage and it is urgent to understand the power and energy behavior of commonly used neural network frameworks on CPU and GPU – the main goal of our paper.

In spite of the advancement in the development of deeper and more complicated neural network structures, research on investigating the energy behavior of different neural networks and software frameworks is still in its infancy. In its white paper [148], Nvidia provides limited power and energy consumption results for CNN inference on two frameworks. Our paper distinguishes itself by providing a comprehensive study on the energy-efficiency of deep neural network training that covers different frameworks, different platforms and different hardware settings.

5.2 Energy Efficiency

5.2.1 Motivation

In recent years convolutional neural networks (CNNs) have been successfully applied to various applications that are appropriate for deep learning, from image and video processing to speech recognition. The advancements in both hardware (e.g. more powerful GPUs) and software (e.g. deep learning models, open-source frameworks and supporting libraries) have significantly improved the accuracy and training time of CNNs. It is now possible to train large and complex neural networks in reasonable time on

relatively inexpensive hardware. This has led to the rapid growth of neural network-based deep learning algorithms.

However, the race for speed and accuracy comes at the cost of energy consumption, an aspect that has been overlooked in previous work. While the classification accuracy has traditionally been considered as the primary metric of success for image and video recognition applications, the community has recently recognized the need for deep neural network implementations that are both accurate and energy efficient. As a result, in 2015 IEEE Rebooting Computing has launched the "Low-Power Image Recognition Challenge" (*LPIRC*), an initiative aimed to promote the design of energy efficient image classification methods. Currently, research in designing energy efficient CNNs is still in its infancy. The knowledge on the power consumption behaviors of different CNNs and training frameworks is very limited. With the size of data sets grows exponentially, the energy demand for training such data sets increases rapidly. It is highly desirable to design deep learning frameworks and algorithms that are both accurate and energy efficient.

Modern GPUs comprise hundreds to thousands of compute cores and can provide high computational power and throughput. As a result, GPU computing has become the de-facto approach for CNNs training [149]. Meanwhile, Intel has recently released the Intel Deep Learning Framework (IDLF) [150], which provides high performance CNNs training on CPU platforms. While Nvidia claims that its GPU framework reports a 11x-14x speedup over the CPU version of Caffe on an Intel IvyBridge processor [151], S. Hadjis et al. [152] point out that Nvidia's comparison is based on an unoptimized CPU baseline, and they introduce CPU optimizations reducing the GPU-CPU performance gap

to only 1.86x. None of these analyses, however, consider the energy efficiency of training the neural networks.

In this work, we conduct a comprehensive study on the power behavior and energy efficiency of CNNs on both CPUs and GPUs [153]. We evaluate popular deep learning frameworks using energy-related metrics and, for each of these frameworks, we provide accurate power measurements using a set of carefully selected networks and layers. We conduct our evaluation on different processor architectures (i.e., Intel Xeon CPUs, Nvidia Kepler and Maxwell GPUs) and explore the effect of a variety of hardware settings to facilitate the design of energy efficient deep learning solutions.

5.2.2 Methodology

A. Introduction to CNN Frameworks

In order to be practically applicable, CNNs require software frameworks that allow high performance training of large-scale networks including millions of parameters. Popular open-source CNN frameworks include: *Caffe* [154], *Torch* [155], *TensorFlow* [156], *MXNet* [157], *Nervana* [158] and *CaffeConTroll* [152]. All these frameworks except *CaffeConTroll* offer both CPU- and GPU-support. In all cases, GPU support is based on the Nvidia cuDNN library [159]. In addition to the cuDNN-based implementation, Caffe and Torch include custom GPU implementations of convolutional layers, pooling layers and activation functions.

B. Experimental Setup

Benchmark Suite: In our experimental evaluation we use Convnet [160], an open-source benchmark that includes most publicly accessible implementations of CNNs. This

benchmark is designed to measure the execution time of the forward and backward propagation of different layers/networks. To obtain accurate power measurements, we apply minor modifications (as described in [148]) to the Convnet benchmark (e.g., we remove unnecessary timing and logging code). We test the training phase (forward and backward propagation) and configure each run to have multiple (>100) iterations, which simulates the real use of these frameworks. Also, by default, the batch size is set to 128, which is a commonly used value.

Neural networks: In our evaluation we use four ImageNet winner neural network models: *AlexNet v2* [161], *OverFeat* [162], *VGG_A* [138] and *GoogleNet* [163]. These networks have been proven successful models and are included in the Convnet benchmark.

Hardware: We perform all experiments on a single machine including a 16-core Intel Xeon E5 - 2650 v2 @ 2.6GHz with Hyper-Threading enabled, an Nvidia Tesla K20m GPU with 5GB memory and an Nvidia Titan X GPU with about 12GB memory. The machine has 32GB DDR3 main memory and a 128 GB SSD hard drive, and runs CentOS v7.

Software: The drivers, libraries and frameworks used in our experiments are: CUDA 7.0, cuDNN v3, OpenBLAS 0.2.16, Caffe (commit ID be163be), Torch 7 (commit ID eb8d7f2), and TensorFlow (commit ID fd464ca), MXNet (commit ID d25053) and CaffeConTroll (commit ID 8191f6c). The CPU and DRAM power data are collected via Intel's Running Average Power Limit (RAPL) interface [164] and the GPU power is obtained via the Nvidia System Management Interface [165].

5.2.3 Overall Results on CPU & GPU

A. Native GPU implementations versus cuDNN

Because GPUs can provide more computational power than general-purpose CPUs, most deep learning frameworks rely on GPUs to provide fast training and inference. To facilitate GPU-accelerated deep learning, Nvidia released the cuDNN library, which includes highly optimized implementations of common operations found in neural networks (e.g. convolution, pooling). Although some early developed frameworks like Caffe and Torch have their own GPU implementations, newer frameworks (e.g. TensorFlow) rely on cuDNN for neural network related operations. In this section we compare native GPU implementations of neural network operations with those found in the cuDNN library.

Figure 56 presents the results of this analysis on the Caffe framework, which can be configured to either use a native GPU implementation or rely on the cuDNN library. We tested 500 iterations of forward and backward propagation with a batch size of 128. In the figure, the bars represent the energy consumption per image processed (left y-axis).

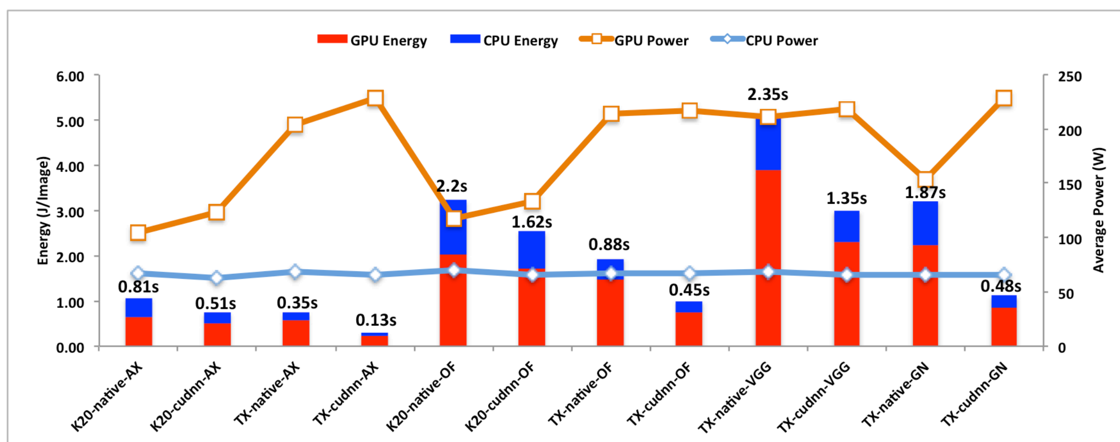


Figure 56: Comparison between native GPU implementation and cuDNN v3 library in Caffe

Specifically, the bottom (red) and top (blue) part of each bar indicate the energy consumption of GPU and CPU, respectively. The two lines show the average power consumption (in Watts) of GPU and CPU (idle); the power consumption scale is on the right y-axis. On top of each bar we report the execution time of a single iteration. In the experiments we use two Nvidia GPUs: a K20m and a Titan X (shown as K20 and TX along the x-axis, respectively) and four networks (AlexNet, OverFeat, VGG_A and GoogleNet). Due to its limited memory capacity, we could not run the VGG_A and GoogleNet networks on the K20m GPU.

As can be observed, cuDNN yields higher GPU power and lower energy consumption than native GPU code on both K20m and Titan X. The increase in power consumption from native GPU kernel to cuDNN varies from 2% (OverFeat on Titan X) to 48% (GoogleNet on Titan X). On average, cuDNN increases power consumption by 16% and reduces energy consumption by 42%. This can be explained as follows. Due to its higher GPU utilization, cuDNN leads to higher power consumption than native GPU code. However, by significantly reducing the training time, cuDNN diminishes the total energy consumption.

If we compare the power and energy consumption of the CNN code on different GPU platforms, we can conclude that, although Titan X consumes more power than K20m, it is more energy-efficient. Taking AlexNet as an example, when moving from K20m to Titan X, the energy consumption is reduced by 15% (for native GPU code) and by 54% (for cuDNN). Compared to K20m, Titan X can deliver more computational power, leading to higher power consumption but also to lower execution time. The reduction in

execution time is significant enough to yield better energy efficiency despite the higher power consumption.

Figure 56 shows another interesting fact: the CPU consumes a significant amount of energy although it mostly is in the idle state, which should not be ignored. As can be seen, the CPU idle power is about 67w in all cases, while the GPU power varies from 104w to 134w on K20m and from 204w to 228w on Titan X. In general, the CPU accounts for 22% to 40% of the total energy consumption. This indicates that, to be energy-efficient, a CNN framework should utilize both CPU and GPU during the training phase.

Since cuDNN leads to better performance and is more energy-efficient than native GPU implementations of neural network related kernels, we use cuDNN as the default GPU library in all remaining experiments.

B. GPU frameworks

There are many frameworks that use GPUs to accelerate CNN training. We selected five popular frameworks: *Caffe* [154], *Torch* [155], *TensorFlow* [156], *MXNet* [157] and *Nervana* [158]. The first four support both K20m and Titan X, while Nervana supports only Titan X. Since Nervana supports 16- and 32-bit floating-point arithmetic, for this framework we have two settings (TX-fp16 and TX-fp32, respectively). Figure 3 reports the results obtained by running forward and backward propagation on AlexNet using a batch size of 128. As in Figure 2, the bars represent the energy consumption per image (y-axis on the left), the lines show the CPU/GPU power in Watts (y-axis on the right), and the numbers on the bars represent the execution time of each iteration.

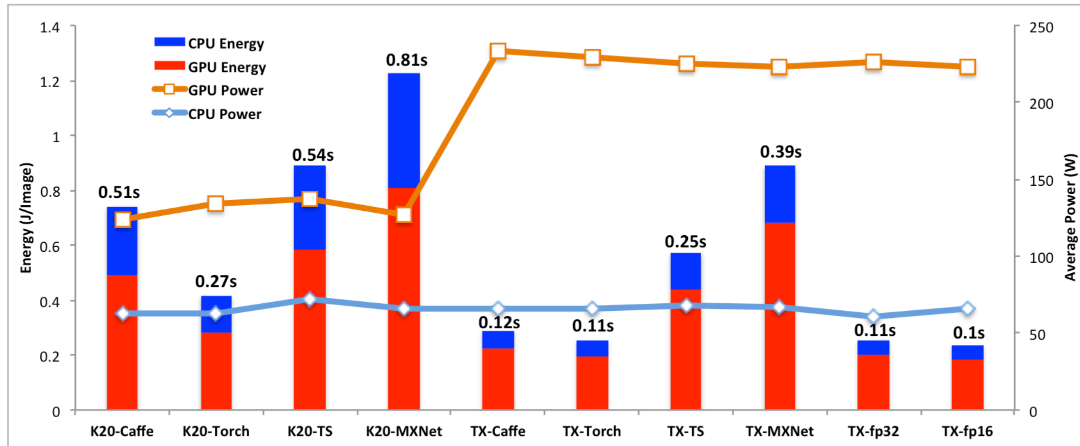


Figure 57: Comparison among different frameworks on K20m and TitanX GPUs

As can be seen, in these experiments Titan X consumes an average power of 227w, 74% more than K20 (130w). Torch outperforms other frameworks in terms of energy efficiency on K20m, and performs similarly to Caffe and Nervana on Titan X. Using 16-bit floating-point arithmetic allows a 10% energy reduction over single precision arithmetic. We can again observe that CPU idle power cannot be ignored, and is especially significant in the case of K20 (since this GPU leads to longer execution times).

C. CPU frameworks

In this section we study the energy and power behavior of CNN frameworks when using only CPUs. Because of its wide use and flexibility, in this set of experiments we focus on Caffe and its derivatives. Caffe supports three CPU libraries (Atlas, OpenBLAS and MKL) that can be statically configured. In our evaluation, we also consider Caffe-OpenMP (an optimized CPU version of Caffe) [166] and CaffeConTroll (a Caffe's derivative that uses an optimization called "*lowering*" [152]). The results of this comparison are shown in Figure 58.

The performance of these frameworks and libraries on CPU is significantly affected by the degree of multithreading. Our machine has 16 physical cores and all the CPU versions are configured to spawn at most 16 threads. To accurately measure power and energy consumption of CPU-based frameworks, we also measure and report the DRAM power data.

As can be seen from Figure 58, the average power consumption of different CPU-based frameworks varies from 103w to 188w. Compared to CPU, DRAM consumes a relatively small portion of energy (11%). Among these CPU versions, Caffe-OpenMP is the most energy-efficient and consumes 2.9 Joules per image processed. It is worth noting that while Caffe-OpenMP is more energy efficient than other CPU implementations, it consumes over twice the amount of energy than all considered GPU frameworks. We can conclude that CPUs are generally less energy-efficient than the GPUs for training CNNs.

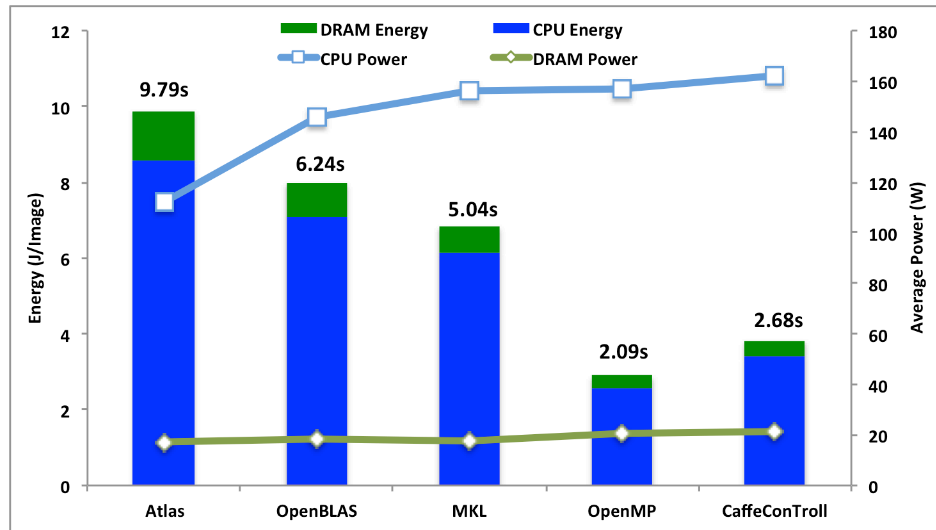


Figure 58: Comparison among CNN frameworks on CPUs

5.2.4 Effect of NN and Batch Size Configuration

A. NN Structure

In this section, we focus on the impact of the network's structure on energy efficiency. To this end, we disassemble AlexNet and OverFeat into four types of layers (convolutional, pooling, fully connected, and ReLU) and measure the distribution of the energy consumption across these layers. In order to ensure that our approach is accurate, we verified that the cumulative energy consumption results from all layers are coherent with the measurements on the integrated network. In Figure 59 we show the percentage breakdown of the energy consumption across layers on GPU and CPU using Caffe. As can be seen, convolutional layers are predominant and consume 87% of the total energy consumption. The second most power-hungry layers are fully connected layers, which account for 10% of the total energy consumption. Pooling layers and ReLU layers (which apply activation functions) account for less than 5% of the energy consumption. This

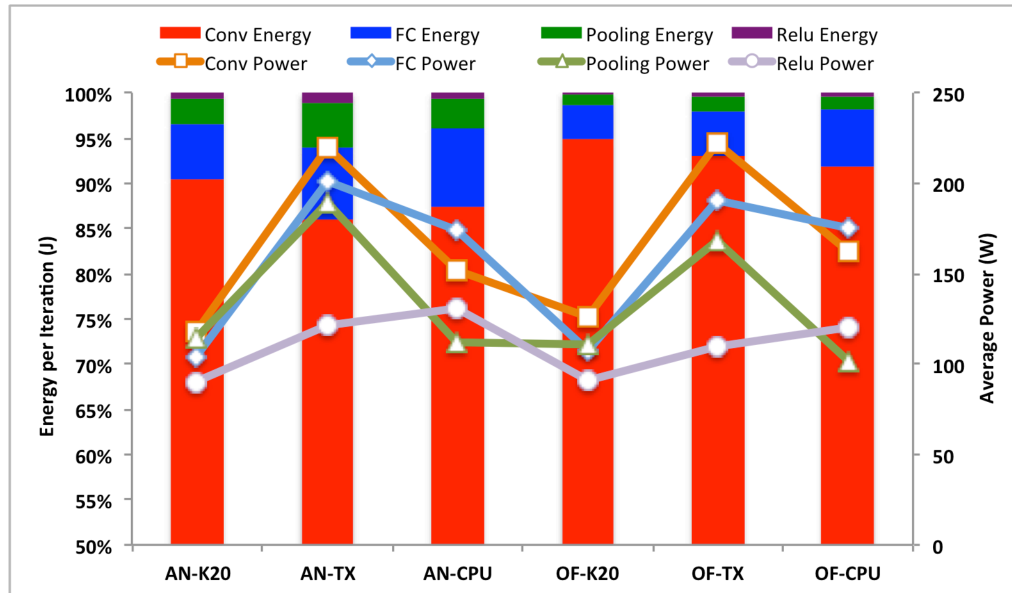


Figure 59: Breakdown of energy consumption of AlexNet and OverFeat on K20 and CPU using Caffe

trend is even more noticeable for OverFeat. In OverFeat, the convolutional layers have more filters (leading to a larger number of neurons) than in AlexNet. Although OverFeat has the same number of layers as AlexNet, its convolutional layers consume a larger portion of the energy, as high as 95%, 93% and 92% on K20m, Titan X and CPU, respectively. From this analysis we can conclude that, in order to optimize energy consumption of deep CNNs, the priority should be on improving the energy efficiency of the convolutional layers.

B. Batch Size

The batch size is an important setting when training a neural network. Larger batch sizes lead to more images being packed into a batch and sent to the network for training in a single iteration. Intuitively, larger batches allow more data-level parallelism. However, loading a larger batch requires more memory to store the data, possibly exhausting the limited GPU memory.

Figure 60 reports the power and energy consumption of AlexNet on K20, Titan X and CPU when the batch size varies from 8 to 256. On all three hardware platforms, increasing the batch size leads to linear growth in power. Small batch sizes (e.g., 8 or 16) cause CPU and GPU under-utilization. An increase in the batch size will yield higher hardware utilization, and, consequently, an increase in power consumption. However, as can be seen, increasing the batch size can reduce the energy consumption per image. For instance, when the batch size is increased from 8 to 16, the energy consumption per image is reduced by 13%, 18% and 31% on K20, Titan X and CPU, respectively. When the hardware utilization saturates, however, a further increase in the batch size does not further improve energy consumption. For example, when increasing the batch size from

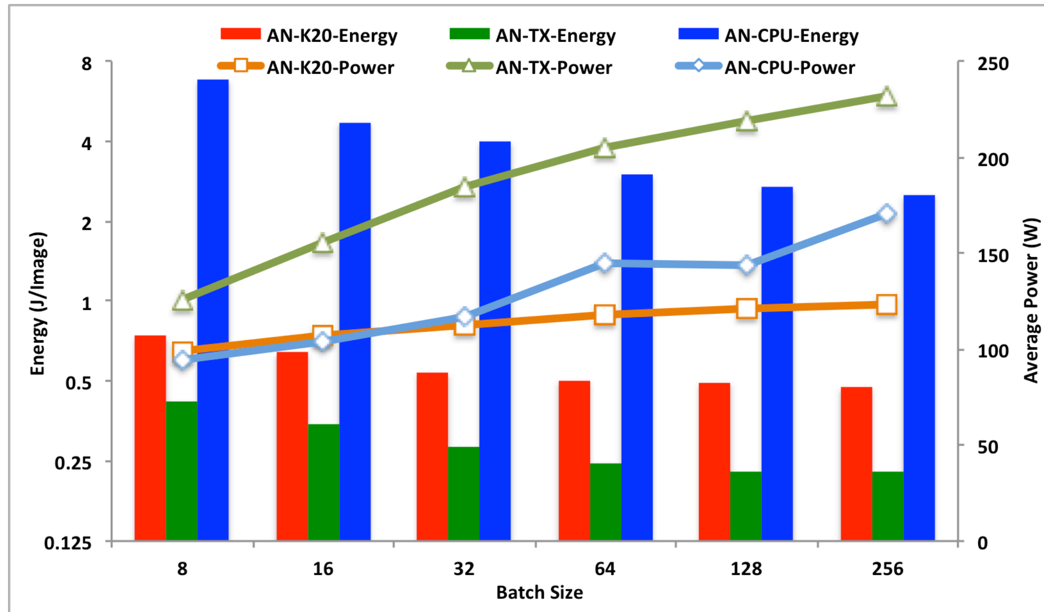


Figure 60: Power and energy consumption on K20m, Titan X and CPU with different batch sizes using Caffe

128 to 256, the energy consumption per image reduces only by 4% on K20 and by 7% on CPU, while Titan X does not benefit from large batch sizes.

5.2.5 Effect of Hardware Settings

In this section, we analyze the impact of different hardware settings on power and energy consumption. Specifically, we investigate how the use of Hyper-Threading on CPU and the use of ECC and DVFS on GPU affect the performance, power and energy consumption of CNNs.

A. Hyper-Threading

Hyper-Threading (HT) is a technology introduced by Intel in order to improve the performance obtainable through parallelization. With HT enabled, the operating system treats each physical processor as two logical cores. HT can improve the performance of memory/IO intensive applications by hiding their latencies, but it often degrades the

performance of compute intensive workloads. To evaluate the impact of HT on deep learning frameworks, we conducted experiments with HT enabled/disabled, in each case configuring the number of threads to equal the number of logical cores. Because our machine has sixteen physical cores, when HT is enabled we set the application to spawn thirty-two threads.

Figure 61 shows the results reported on Caffe and its derivatives. CPU and DRAM energies are stacked into one bar named *HT-E-XXX* or *HT-D-XXX*. In the former case, HT is enabled; in the latter case, HT is disabled. The y-axis on the left side shows the energy consumption per image. The lines represent the power curves of CPU and DRAM and the power scale is on the y-axis on the right side. The values on the bars indicate the total execution time of each setting. Although different libraries and implementations experience different power consumptions, the power consumption is not significantly affected by HT. However, HT affects the execution time and, consequently, the energy

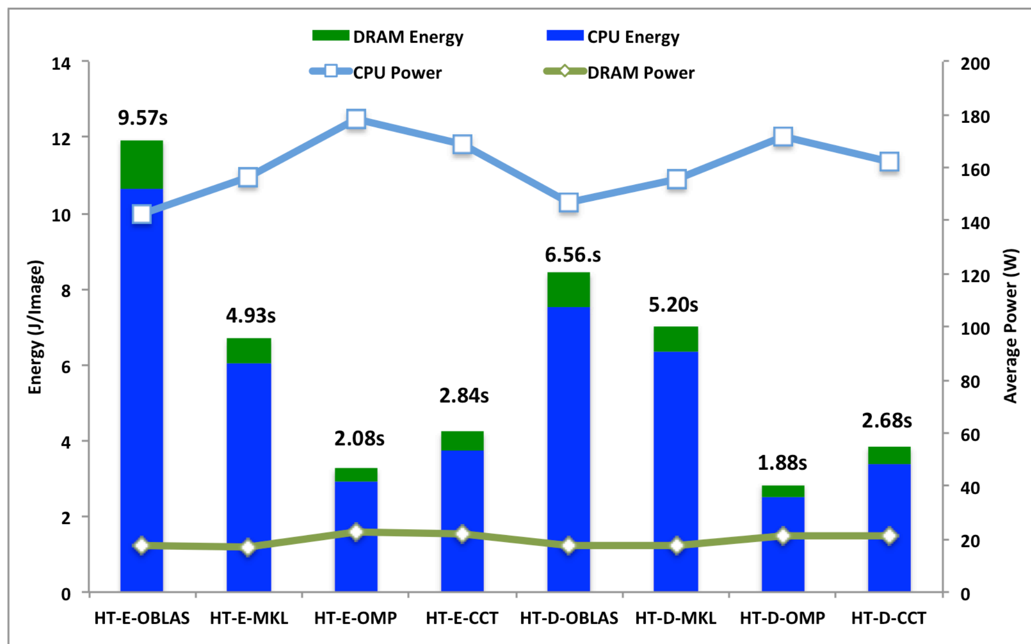


Figure 61: Experimental results with different Hyper-Threading settings

consumption. As can be seen, while MKL reports a slight benefit from enabling HT (5% speedup and 4.8% energy saving), OpenBLAS, OpenMP, CaffeConTroll all suffer various degrees of performance degradation and consume more energy when HT is enabled. In the worst case (Caffe with OpenBLAS), enabling HT leads to an increase in execution time and energy consumption by 45% and 42%, respectively. This experiment shows that performance and energy efficiency of neural network training are generally negatively affected by HT, since this application saturates the hardware with computation and does not experience benefits from memory latency hiding.

B. ECC

Nowadays Error Correction Code RAM is the standard configuration for high performance computing clusters. With ECC enabled, RAM can detect and correct single bit errors. This feature is also available on high-end GPUs, like the ones used in this study.

Although enabling ECC can reduce errors, this feature does not come for free. When ECC is enabled, some bits are reserved, thus reducing the available memory footprint and bandwidth. In addition, enabling ECC causes applications to suffer from more expensive synchronization and uncoalesced memory accesses [167].

We have tested the use of ECC on K20m using Caffe along with the cuDNN library. Our experiments show no significant changes in performance and energy efficiency when ECC is enabled. This is because CNN training does not require synchronization operations and present relatively regular and coalesced memory access patterns. However, we have observed that enabling ECC leads to a 6.2% increase in memory utilization. Since deep CNN applications typically do not require bit-level accuracy and can tolerate

random errors, for these applications disabling ECC can be beneficial in that it allows a better utilization of the limited memory capacity of the GPU. We note that modern GPUs have from 1 to 12 GB of device memory, while CPUs are typically equipped with 16GB up to 1 TB RAM.

C. DVFS

Dynamic Voltage and Frequency Scaling (DVFS) is an advanced power-saving technology that aims to lower a component's power state while still meeting the performance requirement of the workload [168]. Both Titan X and K20m GPUs support DVFS with various clock frequencies. On these devices, DVFS can control two clock frequencies: memory frequency and core frequency. Nvidia provides the nvidia-smi utility and the Nvidia Management Library (NVML) to control these frequencies.

Table 9: Memory and core frequencies supported on K20m GPU

Memory Frequency (MHz)	GPU Core Frequency (MHz)
2600	758
	705
	666
	640
	614
324	324

Table 9 shows the clock frequencies supported on K20m for GPU cores and memory. When the memory frequency is set to 324MHz, the only core frequency is 324Hz. Titan X support wider range of core and memory frequencies. Due to space limits, in this paper we only report results K20m.

Figure 62 shows the power, energy and performance results obtained when applying different memory and core frequencies to CNN training. In the experiments, the batch size is varied from 16 to 128. As expected, power consumption increases with the operation frequency. In the experiments, the power consumption varies from 44w at

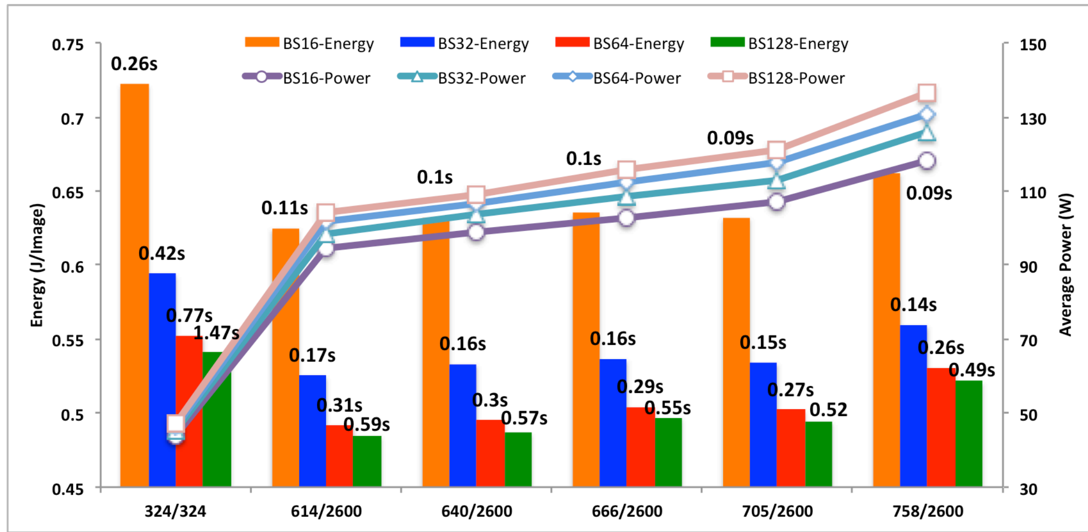


Figure 62: Power and energy consumption using different memory and core frequencies

324/324 MHz frequencies to 118w at 758/2600 MHz frequencies. This wide range of power consumption degrees shows that frequency scaling is an effective method that can be leveraged to meet power cap requirements.

For energy-consumption, our experiments show a "valley" trend: the energy-consumption is relatively high at both the lowest and highest frequencies, and is relatively low at intermediate frequencies. This is because low frequency leads to low power consumption at the cost of longer execution time, negatively affecting the total energy consumption. Conversely, increasing frequency beyond a certain level increases the power consumption while providing only a limited return on performance, also leading to energy inefficiency. This "energy-valley" trend indicates that training deep neural networks in an energy-aware fashion requires operating at frequencies that allow a good trade-off between execution time and power consumption.

5.3 CNN on CPU-GPU heterogeneous architecture

5.3.1 Motivation

GPUs have achieved great success in accelerating various applications. Compared to CPUs, GPUs are comprised of hundreds of cores that can provide more computation power in throughput oriented computing. However, researchers from Intel have proven that the performance gap between CPUs and GPUs is exaggerated, and the comparison between an optimized GPU implementation and a single-threaded CPU baseline is often unfair [43]. Currently, using GPUs for training of CNNs has become the de facto approach. On Nvidia's website, using *Caffe* with GPUs can achieve 11x~14x speedup over CPU version for training CNNs. However, S. Hadjis et al. [44] point out that the CPU baseline has not been optimized, and their proposed optimizations on CPUs can narrow the performance gap to 1.86x, which is comparable to the performance on GPUs.

We observe the current research community's obsession with how to build a larger CNN to improve accuracies in different tasks and how to speedup the parameter learning of CNNs. However, the race for speed is at the sacrifice of energy cost, which is a key metric for any computing system. Hence, power consumption efficiency of CNNs has not been addressed to the extent needed. In traditional competitions like ILSVRC, classification accuracy is the only metric to decide final ranks. More recently, energy efficiency has been taken into consideration and a new competition named "*Low-Power Image Recognition Challenge*" (*LPIRC*) has been developed [45]. This competition is a part of the IEEE Reboot Computing Initiatives and successfully held its 1st competition in early 2015. As CNNs are increasingly used in real world applications, there is a growing interest in energy related research of CNNs.

Although GPUs can deliver a significant amount of computation power, deploying large-scale neural networks on GPUs is limited by the constrained size (1GB - 12GB) of GPU global memory. This prevents exploiting large-scale neural networks to improve classification accuracies for various computer vision and machine learning tasks. On the other hand, CPU has fewer cores than GPU, and the larger memory attached to CPU (several hundred of GB to TB level) makes it feasible to deploy large CNNs with billions of parameters on a single node computer. Regardless of the fact that CPUs and GPUs have complementary advantages and disadvantages regarding computation power and memory space, current frameworks either use CPUs or GPUs and cooperative CPU-GPU methods have not been fully explored. We believe that fully utilizing the heterogeneous system (combining CPUs and GPUs) cannot only extend the ability to train large networks, but also improve the overall system utilization, thereby increasing throughput as well as energy efficiency.

Based on previous analysis, we propose two different CNN parallelization methods on heterogeneous CPU-GPU platforms: Heterogeneous Net (*HetNet*) and Hybrid Net (*HybNet*) [169]. For *HetNet*, the CNN model is mirrored on both CPUs and GPUs. In each forward or backward propagation, the batched data are partitioned into a CPU batch and a GPU batch and then fed to corresponding devices accordingly. For HybNet, a CNN is partitioned into non-sharing CPU layers and GPU layers. In each forward or backward propagation, the batched data need to move between CPU layers and GPU layers.

5.3.2 HetNet

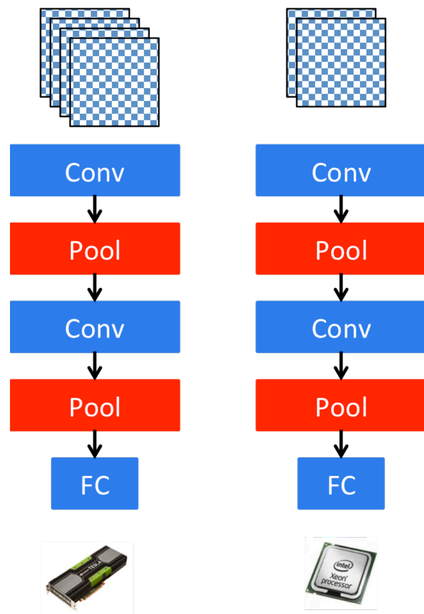


Figure 63: An illustration of HetNet

Figure 63 is an illustration of *HetNet*. In this approach, we keep two mirrored parts of the same model: one on the CPU (*CPUNet*) and one on the GPU (*GPUNet*). When the input data are fetched from the hard drive, we need two batches: one for CPU (*CPU batch*) and the other for GPU (*GPU batch*). In this way, we can perform training or classification on *CPUNet* and *GPUNet* in parallel with its own mini batch. Using CNNs for classification only requires a forward pass without updating the parameters, so the data will be fed into *CPUNet* or *GPUNet* without introducing any extra overhead. However, for learning parameters of CNNs from training data, parameter updating is needed after a full forward and backward propagation.

One problem we need to address is how to update the parameters when training a neural network. Training CNNs requires a forward propagation and a backward propagation to get the gradients. In *HetNet*, two mirrored networks are present thus two different copies of gradients exist. One is from the *CPUNet* and the other is from *GPUNet*. A straightforward but reasonable approach is to merge the two copies of gradients using asynchronous data parallel training [156].

HetNet can fully utilize both CPUs and GPUs in the heterogeneous system and thereby increase the total hardware utilization. The only possible overhead is merging two copies of gradients. However, this operation can be easily and efficiently parallelized

efficiently by vectorization. With improved hardware utilization, we expect that the total training time can be reduced. From an energy efficiency perspective, since static power consumption has contributed to a large portion of the total energy cost, idle CPU can still consume a significant amount of energy. So *HetNet* also improves the energy efficiency by including CPUs for computation tasks.

5.3.3 HybNet

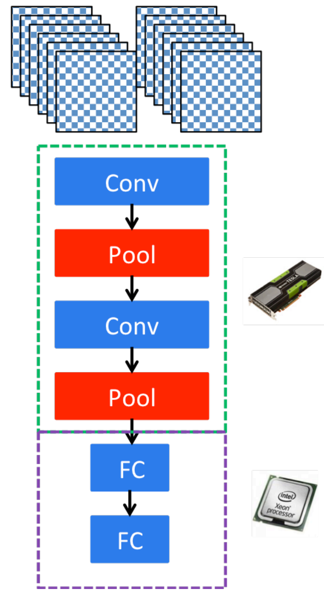


Figure 64: An illustration of HybNet

Figure 64 shows another approach (*HybNet*) that we use to deploy CNN models on a CPU-GPU heterogeneous system. This method is motivated by the fact that FC layers have more parameters than convolutional and pooling layers but are less computationally intensive (multiplication vs. convolution). Also, FC layers are often placed in the last few layers of a CNN, which make the partition possible. We place all the FC layers on CPUs while placing all convolutional and pooling layers on GPU. So in forward propagation, the data go through convolutional layers and pooling layers on GPU first and then the output of last pooling layer is copied to CPU. Then, the FC layers are ready to perform the remaining forward path. For the backward propagation, the data are processed through the FC layers on CPU. After being processed by the last fully

connected layer, the propagated gradients are copied from CPU to GPU. Then, the GPU will complete the remaining backward propagation to get the gradients.

The proposed *HybNet* approach partitions the network into GPU layers (beginning convolutional and pooling layers) and CPU layers (ending FC layers). So it moves parts of the neural network to the CPU so that limited GPU global memory can hold a larger model. However, this will incur data transfer between CPU and GPU for every mini batch as well as fine-grained synchronizations. Also, even though FC layers perform vector-vector multiplications, which can be efficiently implemented on CPUs using high performance libraries (e.g., OpenBLAS [46] and MKL [47]), we still cannot ignore the performance gap between CPU and GPU. Nonetheless, the benefit of fitting larger networks into CPU-GPU heterogeneous system may lead to higher accuracy in machine learning tasks.

5.3.4 Experimental Evaluation

A. Performance Evaluation of HetNet

To evaluate the performance of *HetNet*, we modified *Caffe* to support *CPUNet* and *GPUNet*. We used the same test platforms and software configurations as in Chapter 5.2.2.B. Figure 65 shows the processing time of *HetNet*. We used three software settings: GPU-only (labeled as "K20" and "Titan X"), HetNet-XX-MKL and HetNet-XX-OMP. The suffix "-MKL" indicates implementations based on MKL and suffix "-OMP" indicates the OpenMP implementation of optimized *Caffe*. The numbers on the bars indicate the speedups when the CPU-MKL code serves as the baseline. We can see that including *CPUNet* as part of processing engine in *HetNet* does improve the performance.

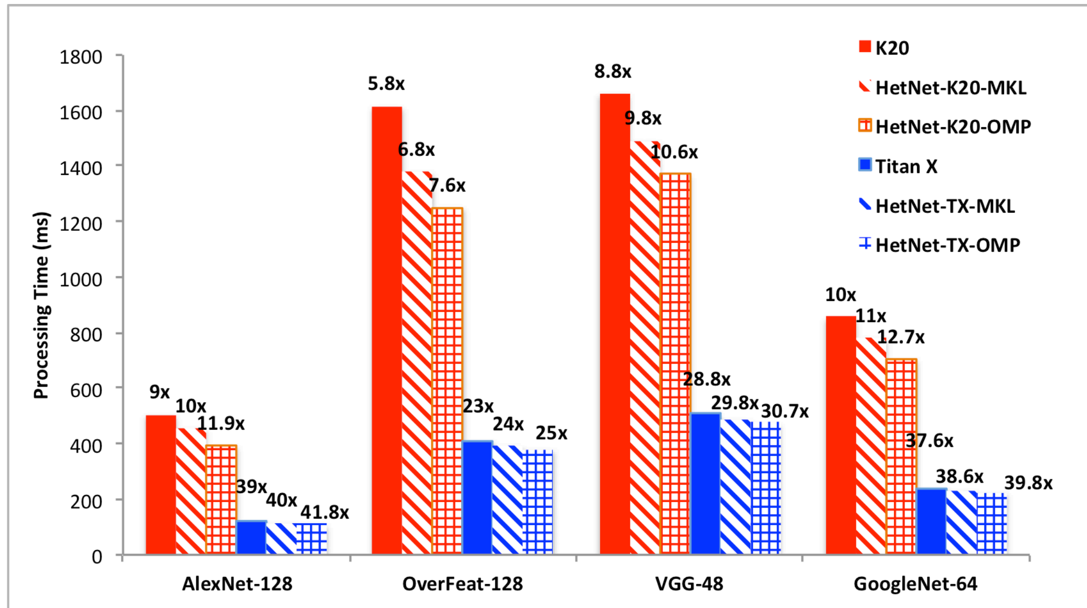


Figure 65: Performance of HetNet

However, the entity of this improvement depends on the performance gap between CPU and GPU. For instance, the K20 GPU has lower single FLOPS than Titan X. As a consequence, utilizing Titan X can achieve a speedup as high as 39x over CPU, compared to a maximum 10x speedup on K20 over CPU. Another observation is that the achieved speedup of *HetNet* over GPU-only version depends on the performance of the *CPUNet* and the *GPUNet*. We can see that using the same *CPUNet* on the same CPU, *HetNet* achieves 20% to 32% speedup on K20 but less than 9% on Titan X. Also, since the OpenMP optimized *Caffe* can provide better performance than MKL enabled *Caffe*, *HetNet* using OpenMP version of *CPUNet* narrows the performance gap between CPU and GPU, thus achieving higher speedups of *HetNet* over GPU-only code.

B. Performance Evaluation of HybNet

As we described in Chapter 5.3.3, *HybNet* partitions the network into GPU layers (convolutional and pooling layers) and CPU layers (ending FC layers). This causes data movements between CPU and GPU at the boundary between the layers mapped onto these devices, thus introducing some overhead. In addition, the same layers are slower on CPU than on GPU. Thus having CPU layers in *HybNet* incurs some performance degradation. Figure 66 shows the performance of *HybNet* on K20 and Titan X on different networks. The numbers following the network names are the batch sizes. We can see that there is no significant difference between our *HybNet* and the GPU-only version on *GoogleNet*. This is because *GoogleNet* has only a single small FC layer. The other three networks (i.e. *AlexNet*, *OverFeat* and *VGG*) have three large FC layers each. Offloading these FC layers to CPU, which is slower than GPU, leads to longer processing time. From the figure, we can see that on K20, *HybNet* leads to a drop in processing speed ranging from 7% to 14% over a GPU-only implementation. On Titan X, due to the

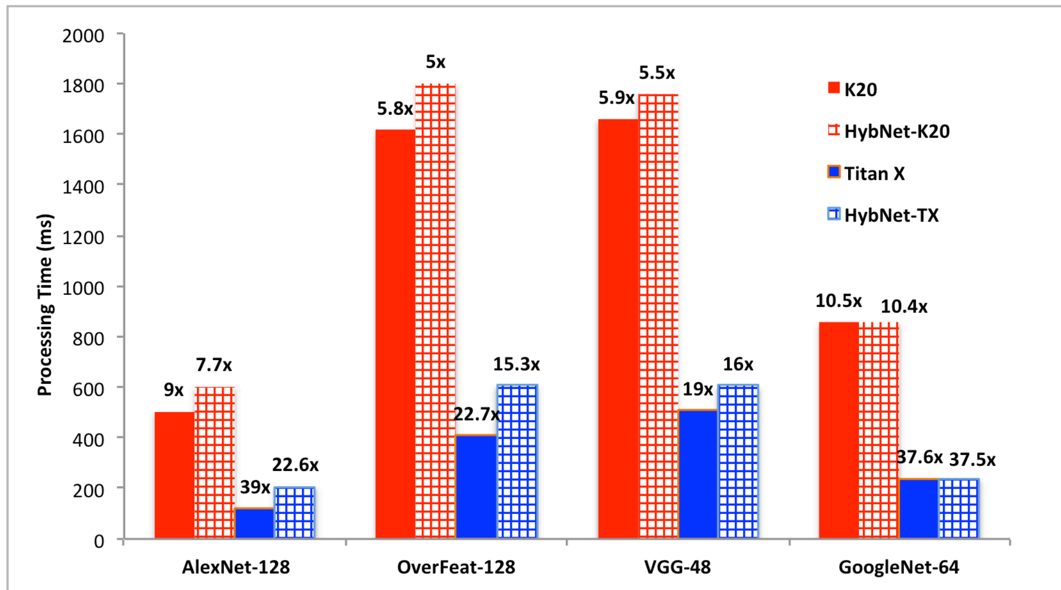


Figure 66: Performance of HybNet with different neural networks

enlarged performance gap between CPU and Titan X (compared to K20), the performance degradation is more obvious, ranging from 16% to 42%.

In terms of GPU memory usage, *HybNet* benefits from moving all FC layers from GPU to CPU. Figure 67 shows the GPU memory footprint for all the test cases in Figure 66. Since *GoogleNet* has only a single small FC layer, in this case the GPU memory footprint of *HybNet* is almost the same as that of the GPU-only version. For the other three networks, on average *HybNet* reduces the GPU memory usage by 30% compared to the GPU-only solution, although the exact numbers depend on the network and batch size.

The data residing on GPU store two pieces of information: learnable parameters of the network and intermediate data flowing through the network. Once the network is designed, the total number of learnable parameters is fixed. However, the size of the intermediate data depends on the batch size: increasing the batch size will increase the amount of intermediate data. Figure 68 and Figure 69 show the performance and GPU memory consumption of *HybNet* on *AlexNet* using different batch sizes (from 16 to 128),

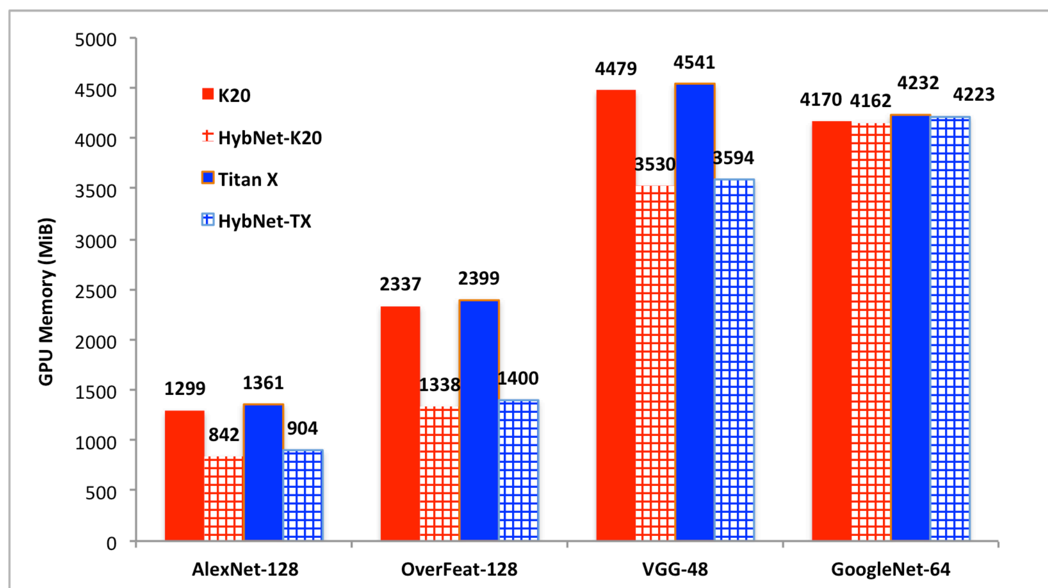


Figure 67: GPU memory footprint of HybNet with different neural networks

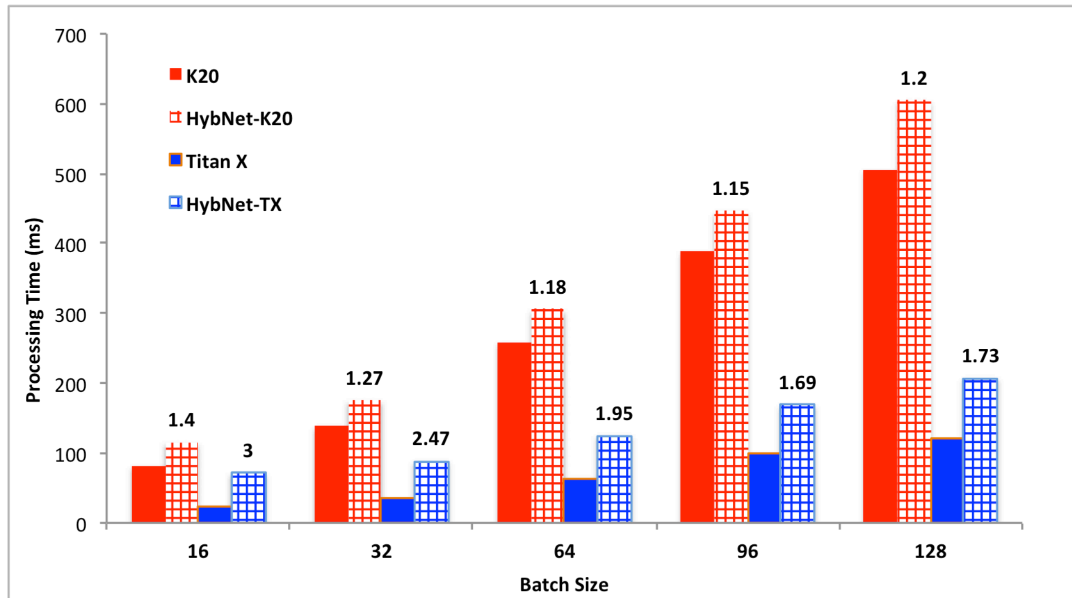


Figure 68: Performance of HybNet with AlexNet under different batch sizes

respectively. In Figure 68, the numbers on the *HybNet* bars are the ratios of *HybNet*'s processing time over GPU-only processing. We can observe that increasing the batch size amortizes the performance degradation. This is because, for large batches, the convolutional layers and pooling layers consume much more processing time than the FC layers. Therefore, as the batch size increases, the slowdown due to moving the FC layers to CPU has a lesser effect on performance.

Figure 69 shows the GPU memory footprint of *HybNet*. As can be seen, with the increase of batch size, the total GPU memory footprint increases but the total amount of reduced memory seems to remain the same. This can be explained that the FC layers moved to the CPU have a lot of learnable parameters but a few data, thus increasing data batch size has limited impact on increasing the memory footprint of the FC layers. This indicates that our *HybNet* can efficiently reduce the memory footprint when the batch size is small.

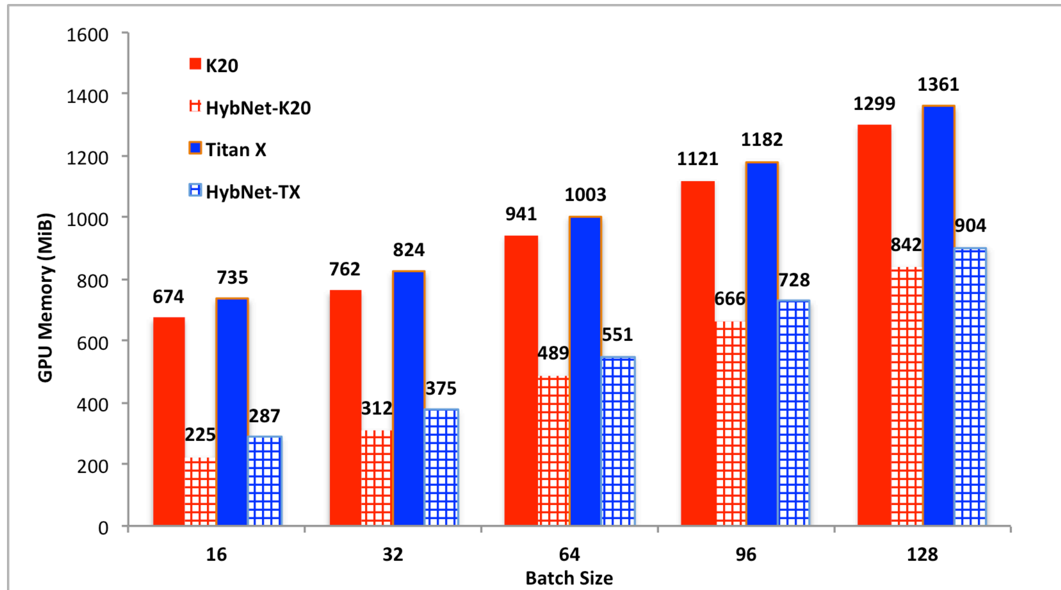


Figure 69: GPU memory footprint of HybNet with AlexNet under different batch sizes

5.4 Virtual Memory for CNN on GPU

5.4.1 Motivation

In 2012 *AlexNet* was the first CNN-based model to win the ImageNet Large Scale Visual Recognition Competition (*ILSVRC*), reducing the error rate dramatically from 26% to 16%. In subsequent years, other CNN-based models dominated ILSVRC, and each year researchers have proposed new models with more layers and convolutional kernels, leading to deeper network architectures, and pushing the state-of-the-art of image recognition to human levels. By including more learnable parameters, larger neural networks have the potential for achieving higher accuracy. However, larger and deeper neural networks require more powerful processors to be trained in reasonable time. GPUs can provide more than 10x speedups over CPUs and have become the de-factor platform for training neural networks. However, GPUs have limited device memory (4 GB to 12 GB). We have observed that current CNN frameworks for GPU, like Caffe, allocate all

memory required by a CNN on the GPU at the beginning of computation and free it only at the end. Large-scale neural networks often have large memory footprints exceeding the GPU memory, making it infeasible for a GPU to accommodate and train these models.

On the other hand, virtual memory has become a standard component in operating systems for GPUs and existing work [170] has explored extending the concept of virtual memory to GPUs. This motivates us to design and implement *Virtual Deep Neural Network* (*vDNN*), a memory manager with virtual memory support for CNNs on GPU. We have prototyped vDNN on top of *Caffe*. Our experiments show that vDNN allows training large-scale neural networks (e.g. 30 GB memory footprint) on K20 (4.5 GB device memory) and Titan X (12 GB device memory) at the cost of some runtime overhead.

5.4.2 Design

The main component of vDNN is a memory manager that supports virtual memory. The basic idea to keep a persistent copy of all the data of a CNN on the host (CPU) memory, and to move the data required in each step of the GPU computation from host to device on demand. Once the device copy is allocated, it will reside on GPU unless it is swapped out to the persistent copy in the host memory due to a shortage of device memory. This *swap* operation can happen when a data copy is requested from the device but serving this request would cause the device capacity to be exceeded.

The memory manager includes the following components: (1) *mapping table*, (2) *swap area*, (3) *swap candidate pool* and (4) *working set*.

Mapping table: This table stores the mapping between device copies and host copies of the data. It is similar to an OS *page table*, which stores the address translation between physical and virtual memory. The main difference is that in vDNN the data are organized in chunks with different sizes (instead of fixed "page sizes").

Swap area: The swap area resides in the host memory and stores the persistent copies of the data.

Swap candidate pool: When the requested device memory allocation exceeds the device capacity, one or multiple swaps will happen. The swap candidate pool maintains a record of the data copies residing on GPU that can potentially be moved to the swap area in the host memory so as to free GPU memory.

Working set: The working set maintains a record of data copies that are actively in use on the GPU. Unlike the swap candidate pool, the data copies in the working set should not be swapped out because they are currently used.

5.4.3 Experimental Evaluation

To evaluate the performance of our virtual memory system, we conduct two experiments. The first one is a meant to evaluate the overhead of vDNN's virtual memory system. In this experiment, we use small CNN models that fit the device memory, but limit the amount of GPU memory used by vDNN so as to force some swap operations. We then compare the performance of vDNN with that of the unmodified *Caffe*. We recall that *Caffe* allocates and de-allocates all the CNN data on GPU at the beginning and at the end of the computation, respectively. In the second experiment, we let the memory manager

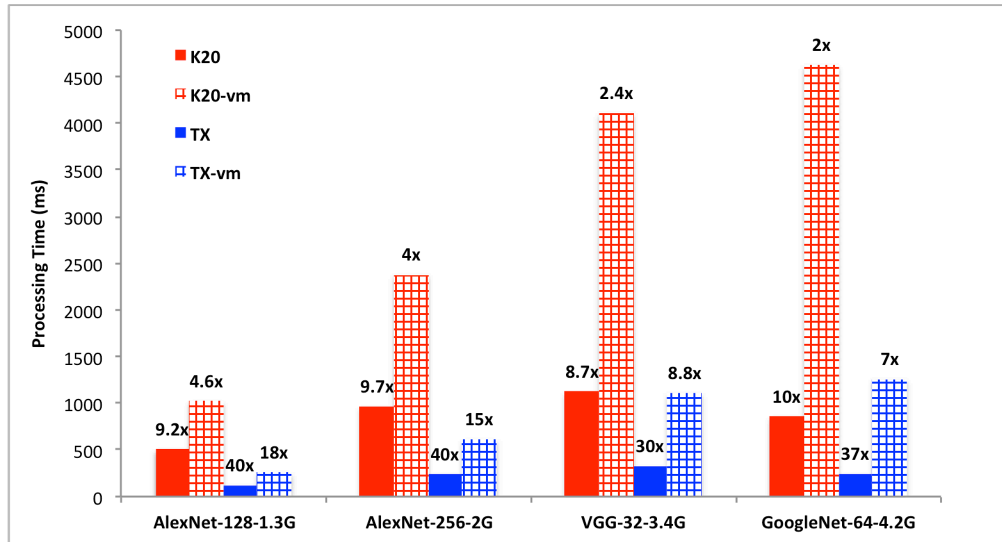


Figure 70: Performance of vDNN with 1G GPU memory

use all the GPU memory and train some very large networks that would require more device memory than available on current GPUs.

In Figure 70 we show performance data collected when configuring vDNN to use 1 GB physical GPU memory. We trained four models with different batch sizes. On the x-axis, each test is named in the format of "network name" - "batch size" - "required memory size". Since vDNN is configured to use 1GB device memory and the memory footprint of the considered neural networks ranges from 1.3 GB to 4.2 GB, swap operations will happen in all cases. The numbers on the bars indicate the speedup of each implementation over Caffe-MKL. For each neural network, we test *Caffe* on GPU with and without vDNN. As can be seen, vDNN experiences performance degradation due to the swap overhead. For the smallest neural network, AlexNet-128-1.3G, vDNN achieves roughly half of the performance of original *Caffe* on GPU. This performance slowdown increases with the size of the CNN model. For instance, on K20 "AlexNet-128-1.3G", vDNN achieves 50% performance of original *Caffe* but this number drops to 40% on

“AlexNet-256-2G”, 27.5% on “VGG-32-3.4G” and 20% on “GoogleNet-64-4.2G”. This is because larger models lead to more swap operations and data movements, hurting performance.

Figure 71 shows results of the second experiment, in which vDNN manages the full available GPU memory and large models, which cannot fit into GPU memory, are trained. In this experiment, we train *GoogleNet* using different batch sizes. The naming convention on the x-axis is the same as in Figure 70 and “GN” stands for “GoogleNet”. The y-axis shows the processing time in exponential scale. Recall that K20 and Titan X have 4.5 GB and 12 GB device memory, respectively. Hence, on “GN-64-4G” no swap is necessary and the vDNN achieves almost the same performance as the original *Caffe* on GPU (overhead less than 2%). With the increase in batch size, the total memory required by the model also increases. As a result, swap operations become more frequent and the speedup of vDNN over CPU-MKL becomes smaller. From the Figure 71, we can see that

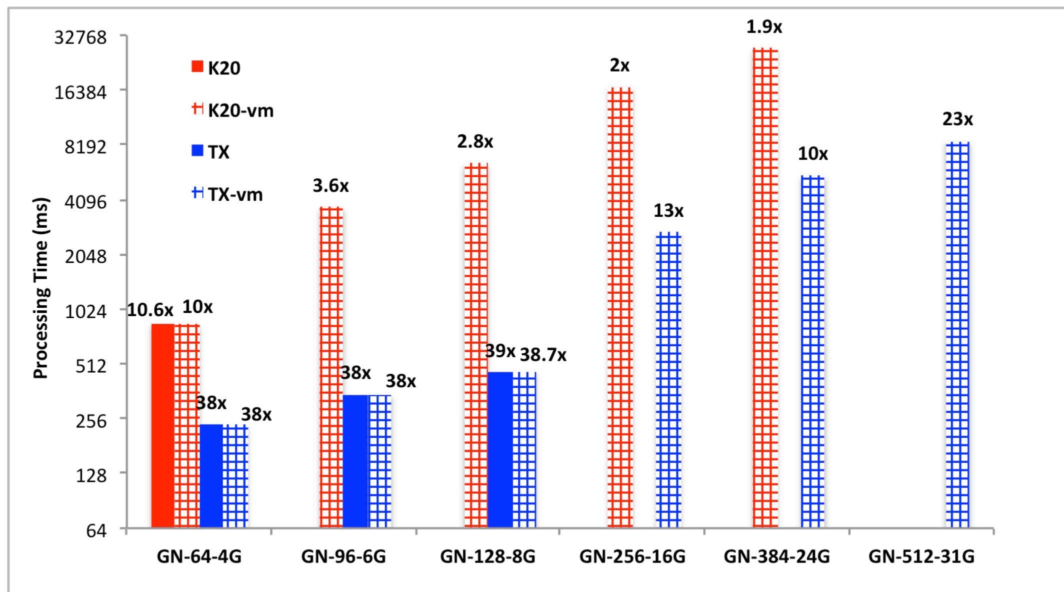


Figure 71: Performance of vDNN with full GPU global memory

vDNN can facilitate training 24 GB model on K20 with 4.5 GB GPU memory and 31 GB model on Titan X with 12 GB memory. These models are larger than any available GPU in the market. vDNN makes training these very large neural networks possible on existing GPU hardware.

Chapter 6 Conclusion

In this dissertation, we have studied the acceleration and deployment of three important categories of emerging applications on many-core processors. The considered applications range from bioinformatics, to graph processing and other applications with irregular computation and memory access patterns, to deep neural networks, which represent some of the most challenging and computational intensive problems faced in our research community. We have addressed several research questions related to the design of software systems for many-core platforms. Our contributions include:

1) We have explored different implementations of the Needleman-Wunsch (NW) algorithm on GPU. The methods considered differ in their computational patterns, their use of the available hardware parallelism, and their handling of the data dependencies intrinsic in NW. Our analysis gives insights into the architectural benefits and costs of using GPUs for bioinformatics, and our proposed techniques are also applicable to other domains (e.g. computer vision algorithms).
2) We have designed source-to-source transformation techniques to automatically generate parallel code for different many-core platforms (GPUs and the Intel Xeon Phi) starting from a platform-agnostic graph programming API. We have proposed a programming framework including programming and runtime support for dynamic memory allocation, and we have studied the effect of synchronization on the performance of our runtime library.
3) We have explored an implementation space for graph algorithms on GPU. Our analysis shows that there is no optimal solution across graph problems and

datasets. We have proposed an adaptive runtime that dynamically selects the most suitable implementation among the ones resulting from the aforementioned exploration space and we have designed data structures that lead to minimal overhead when switching between implementations at runtime. Further, we have devised heuristics that guide the decisions of our adaptive runtime.

4) We have proposed and studied several parallelization templates to allow the effective deployment of irregular applications with uneven work distribution on GPU. We have observed that a new GPU hardware feature - named *dynamic parallelism* (DP) – can help embedding work balancing mechanisms in applications, but its naïve use can suffer from significant overhead. To avoid this overhead, we have proposed a software-based workload consolidation mechanism for kernels relying on DP, and we have integrated these schemes in a directive-based compiler. By automating our code transformations, we allow programmers to write simple code focusing on functionality rather than on performance. We have observed that static methods to configure the degree of multithreading of GPU kernels are ineffective in the presence of DP and we have proposed a systematic way to configure dynamic kernel launches

5) We have conducted a comprehensive study on the power behavior and energy efficiency of numerous well-known CNNs and training frameworks on CPUs and GPUs, and we have provided a detailed workload characterization to facilitate the design of energy efficient deep learning solutions. We have extended existing CPU-only or GPU-only CNNs learning methods to CPU-GPU cooperative

operation. Further, we have proposed and implemented vDNN on top of Caffe to facilitate training large CNN models with limited GPU memory.

Our work leaves some open research directions. These include:

1) We have explored the acceleration of a specific bioinformatics application. In fact, many algorithms from bioinformatics and other application areas share similar computational patterns. Therefore, it would be interesting to generalize our study and propose more generic frameworks allowing the effective deployment of classes of applications with similar patterns.
2) For irregular applications, we have limited our study to the single node/single GPU setting. However, with the ever-increasing dataset sizes in modern applications, parallelization on distributed systems becomes paramount. This will introduce more challenges related to graph partitioning, communication across partitions, and algorithm refactoring.
3) Embedded systems including CPUs, GPUs and FPGAs are widely used in various areas such as consumer electronic products, drones, autonomous driving. It would be interesting to extend the workload characterization that we have proposed for training deep neural networks to these platforms.

References

[1] L. Song, M. Feng, N. Ravi, Y. Yang, and S. Chakradhar., "COMP: Compiler Optimizations for Manycore Processors," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 659-671.

[2] S. Henikoff, and J. G. Henikoff, "Amino-acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences, U.S.A.,* vol. 89, no. #22, pp. 10915-10919, 1992.

[3] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM,* vol. 18, no. 6, pp. 341-343, 1975.

[4] S. B. Needleman, and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology,* vol. 48, pp. 443-453, 1970.

[5] J. R. Cole, Q. Wang, E. Cardenas, J. Fish, B. Chai, R. J. Farris, A. S. Kulam-Syed-Mohideen, D. M. McGarrell, T. Marsh, G. M. Garrity, and J. M. Tiedje, "The Ribosomal Database Project: improved alignments and new tools for rRNA analysis," *Nucleic Acids Research,* vol. 37, no. Database issue, pp. D141-5, Jan, 2009.

[6] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological review,* vol. 386, 1958.

[7] TheanoTeam. "Convolutional Neural Networks (LeNet)," http://deeplearning.net/tutorial/lenet.html.

[8] T. F. Smith, and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology,* vol. 147, no. #1, pp. 195-197, Mar 25, 1981.

[9] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice.," *Nucleic Acids Research,* vol. 22, pp. 4673-4680, 1994.

[10] D. M. Hillis, C. Moritz, and B. K. Mable, *Molecular Systematics: Second Edition*, Sunderland, MA: Sinauer Associates, 1996.

[11] M. Nei, and T. Gojobori, "Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions," *Molecular Biology and Evolution,* vol. 3, no. 5, pp. 418-426, 1986.

[12] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank," *Nucleic Acids Research,* vol. 41, no. Database issue, pp. D36-42, Jan, 2013.

[13] W. R. Pearson, and D. J. Lipman, “Improved tools for biological sequence comparison,” *Proc Natl Acad Sci U S A,* vol. 85, no. 8, pp. 2444-8, Apr, 1988.

[14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic Local Alignment Search Tool,” *Journal of Molecular Biology,* vol. 215, no. #3, pp. 403-410, 1990.

[15] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. H. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped Blast and Psi-Blast : A new-generation of protein database search programs,” *Nucleic Acids Research,* vol. 25, no. #17, pp. 3389-3402, 1997.

[16] H. Li, J. Ruan, and R. Durbin, “Mapping short DNA sequencing reads and calling variants using mapping quality scores,” *Genome Research,* vol. 18, no. 11, pp. 1851-8, Nov, 2008.

[17] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome,” *Genome Biology,* vol. 10, no. 3, pp. R25, 2009.

[18] K. Meusemann, B. M. von Reumont, S. Simon, F. Roeding, S. Strauss, P. Kuck, I. Ebersberger, M. Walzl, G. Pass, S. Breuers, V. Achter, A. von Haeseler, T. Burmester, H. Hadrys, J. W. Wagele, and B. Misof, “A phylogenomic approach to resolve the arthropod tree of life,” *Molecular Biology and Evolution,* vol. 27, no. 11, pp. 2451-64, Nov, 2010.

[19] N. R. Pace, “Mapping the tree of life: progress and prospects,” *Microbiol Mol Biol Rev,* vol. 73, no. 4, pp. 565-76, Dec, 2009.

[20] L. W. Parfrey, J. Grant, Y. I. Tekle, E. Lasek-Nesselquist, H. G. Morrison, M. L. Sogin, D. J. Patterson, and L. A. Katz, “Broadly sampled multigene analyses yield a well-resolved eukaryotic tree of life,” *Syst Biol,* vol. 59, no. 5, pp. 518-33, Oct, 2010.

[21] O. Beja, M. T. Suzuki, J. F. Heidelberg, W. C. Nelson, C. M. Preston, T. Hamada, J. A. Eisen, C. M. Fraser, and E. F. DeLong, “Unsuspected diversity among marine aerobic anoxygenic phototrophs,” *Nature,* vol. 415, no. 6872, pp. 630-3, Feb 7, 2002.

[22] M. Kim, M. Morrison, and Z. Yu, “Status of the phylogenetic diversity census of ruminal microbiomes,” *FEMS Microbiol Ecol,* vol. 76, no. 1, pp. 49-63, Apr, 2011.

[23] S. G. Tringe, and E. M. Rubin, “Metagenomics: DNA sequencing of environmental samples,” *Nature Reviews Genetics,* vol. 6, no. 11, pp. 805-814, Nov, 2005.

[24] J. C. Venter, K. Remington, J. F. Heidelberg, A. L. Halpern, D. Rusch, J. A. Eisen, D. Wu, I. Paulsen, K. E. Nelson, W. Nelson, D. E. Fouts, S. Levy, A. H. Knap, M. W. Lomas, K. Nealson, O. White, J. Peterson, J. Hoffman, R. Parsons, H. Baden-Tillson, C. Pfannkoch, Y. H. Rogers, and H. O. Smith, "Environmental genome shotgun sequencing of the Sargasso Sea," *Science,* vol. 304, no. 5667, pp. 66-74, Apr 2, 2004.

[25] M. F. Whitford, R. J. Forster, C. E. Beard, J. Gong, and R. M. Teather, "Phylogenetic analysis of rumen bacteria by comparative sequence analysis of cloned 16S rRNA genes," *Anaerobe,* vol. 4, no. 3, pp. 153-63, Jun, 1998.

[26] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program GPUs for general-purpose uses," *SIGARCH Comput. Archit. News,* vol. 34, no. 5, pp. 325-335, 2006.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in Proc. of IISWC, 2009, pp. 44-54.

[28] "Nvidia Applications Catalog," http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf.

[29] P. D. Vouzis, and N. V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics,* vol. 27, no. 2, pp. 182-8, Jan 15, 2010.

[30] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using Graphics Processing Units," *BMC Bioinformatics*, 2007.

[31] J. P. Walters, X. Meng, V. Chaudhary, T. Oliver, L. Y. Yeow, B. Schmidt, D. Nathan, and J. Landman, "MPI-HMMER-Boost: Distributed FPGA Acceleration," *The Journal of VLSI Signla Processing Systems for Signal, Image, and Video Technology,* vol. 48, no. 3, pp. 6, 2007.

[32] B. Zhang, T. K. Saha, and M. A. Hasan, "Name disambiguation from link data in a collaboration graph," in Advances in Social Networks Analysis and Mining (ASONAM), 2014 IEEE/ACM International Conference on, 2014, pp. 81-84.

[33] B. Pang, N. Zhao, M. Becchi, D. Korkin, and C. R. Shyu, "Accelerating large-scale protein structure alignments with graphics processing units," *BMC Res Notes,* vol. 5, pp. 116, 2012.

[34] W. Liu, B. Schmidt, and W. Muller-Wittig, "CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware," *IEEE Transactions on Computational Biology and Bioinformatics,* vol. 8, no. 6, pp. 1678-1684, 2011.

[35] Y. Liu, B. Schmidt, and D. L. Maskell, "MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA," 2009, pp. 121-128.

[36] Y. Liu, B. Schmidt, and D. L. Maskell, “DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI,” *BMC Bioinformatics*, 2011.

[37] Y. Liu, B. Schmidt, and D. L. Maskell, “CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform,” *Bioinformatics,* vol. 28, no. 14, pp. 1830-1837, 2011.

[38] Z. Zheng, T.-D. Nguyen, and B. Schmidt, "CRiSPy-CUDA: Computing Species Richness in 16S rRNA Pyrosequencing Datasets with CUDA," *Pattern Rcognition in Bioinformatics*, pp. 37-49, 2011.

[39] S. A. Manavski, and G. Valle, “CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment,” *BMC Bioinformatics,* vol. 9 Suppl 2, pp. S10, 2008.

[40] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, “Streaming algorithms for biological sequence alignment on GPUs,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 19, pp. 1270-1281, 2007.

[41] Y. Gao, and J. D. Bakos, “GPU Acceleration of Pyrosequencing Noise Removal,” in Proc. of SAAHPC, Argonne, IL USA, 2012, pp. 94-101.

[42] Y. Liu, D. L. Maskell, and B. Schmidt, “CUDASW++: Optimizing Smith-Waterman Sequence Database Searches for CUDA-enabled Graphics Processing Units,” *BMC Research Notes,* vol. 2, no. 73, 2009.

[43] A. Wirawan, C. K. Kwoh, N. T. Hieu, and B. Schmidt, “CBESW: Sequence Alignment on the Playstation 3,” *BMC Bioinformatics,* vol. 9, 2008.

[44] A. Szalkowski, C. Ledergerber, P. Krahenbuhl, and C. Dessimoz, “SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2,” *BMC Res Notes,* vol. 1, 2008.

[45] J. Li, S. Ranka, and S. Sahni, “Pairwise sequence alignment for very long sequences on GPUs,” in Proc. of ICCABS, 2012, pp. 1-6.

[46] K.-B. Li, “ClustalW-MPI: ClustalW analysis using distributed and parallel computing,” *Bioinformatics,* vol. 19, no. 12, pp. 2, 2003.

[47] A. Biegert, C. Mayer, M. Remmert, J. Soding, and A. N. Lupas, “The MPI Bioinformatics Toolkit for protein sequence analysis,” *Nucleic Acids Research,* vol. 34, pp. 5, 2006.

[48] D. Li, and M. Becchi, “Multiple Pairwise Sequence Alignments with the Needleman-Wunsch Algorithm on GPU,” in High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, 2012, pp. 1471-1472.

[49] D. Li, K. Sajjapongse, H. Truong, G. Conant, and M. Becchi, “A Distributed CPU-GPU Framework for Pairwise Alignments on Large-Scale Sequence Datasets,” in Proceedings of the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Ashburn, VA, 2013.

[50] H. Truong, D. Li, K. Sajjapongse, G. Conant, and M. Becchi, “Large-scale pairwise alignments on GPU clusters: exploring the implementation space,” *Journal of Signal Processing Systems,* vol. 77, pp. 131-149, 2014.

[51] J. Sanders, and E. Jabdrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*: Addison-Wesley Professional, 2010.

[52] D. A. Bader, and G. Cong, “Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs,” *J. Parallel Distrib. Comput.,* vol. 66, no. 11, pp. 1366-1378, 2006.

[53] C. E. Leiserson, and T. B. Schardl, “A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers),” in Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, Thira, Santorini, Greece, 2010, pp. 303-314.

[54] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, “Scalable Graph Exploration on Multicore Processors,” in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.

[55] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, “A Parallelization of Dijkstra's Shortest Path Algorithm,” in Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science, 1998, pp. 722-731.

[56] G. Vaira, and O. Kurasova, “Parallel Bidirectional Dijkstra's Shortest Path Algorithm,” in Proceedings of the 2011 conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, 2011, pp. 422-435.

[57] D. B. Johnson, and P. Metaxas, “A parallel algorithm for computing minimum spanning trees,” in Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, San Diego, California, USA, 1992, pp. 363-372.

[58] F. Dehne, S. G, #246, and tz, “Practical Parallel Algorithms for Minimum Spanning Trees,” in Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems, 1998, pp. 366.

[59] V. Koubek, and J. Kršňáková, "Parallel algorithms for connected components in a graph," *Fundamentals of Computation Theory*, pp. 208-217: Springer Berlin Heidelberg, 1985.

[60] I. William Mclendon, B. Hendrickson, S. Plimpton, and L. Rauchwerger, "Finding strongly connected components in parallel in particle transport sweeps," in Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, Crete Island, Greece, 2001, pp. 328-329.

[61] W. Schudy, "Finding strongly connected components in parallel using O(log2n) reachability queries," in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, Munich, Germany, 2008, pp. 146-151.

[62] T. K. Saha, B. Zhang, and M. A. Hasan, "Name Disambiguation from link data in a collaboration graph using temporal and topological features," in Journal of Social Network Analysis and Mining (SNAM 2015), 2015.

[63] P.-Y. Chen, B. Zhang, M. A. Hasan, and A. Hero, "Incremental Method for Spectral Clustering of Increasing Orders," in KDD Workshop on Mining and Learning with Graphs (MLG 2016), 2016.

[64] D. Gregor, and A. Lumsdaine, "Lifting sequential graph algorithms for distributed-memory parallel computation," in Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, USA, 2005, pp. 423-437.

[65] F. Hielscher, and P. Gottschling. "ParGraph," http://pargraph.sourceforge.net.

[66] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: an adaptive, generic parallel C++ library," in Proceedings of the 14th international conference on Languages and compilers for parallel computing, Cumberland Falls, KY, USA, 2003, pp. 193-208.

[67] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment,* vol. 5, no. 8, pp. 716-727, 2012.

[68] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in Processings of the 10th USENIX conference on Operating Systems Design and Implementation, Berkeley, CA, USA, 2012, pp. 31-46.

[69] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: a DSL for easy and efficient graph analysis," in Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, England, UK, 2012, pp. 349-362.

[70] M. Dundar, Q. Kou, B. Zhang, Y. He, and B. Rajwa, "Simplicity of Kmeans versus Deepness of Deep Learning: A Case of Unsupervised Feature Learning with Limited Data," in IEEE International Conference on Machine Learning Applications, 2015.

[71] S. Choudhury, K. Agarwal, S. Purohit, B. Zhang, M. Pirrung, W. Smith, and M. Thomas, *NOUS: Construction and Querying of Dynamic Knowledge Graphs*, arXiv preprint, 2016.

[72] B. Zhang, S. Choudhury, M. A. Hasan, X. Ning, K. Agarwal, S. Purohit, and P. P. Cabrera, "Trust from the past: Bayesian Personalized Ranking based Link Prediction in Knowledge Graphs," in SDM Workshop on Mining Networks and Graphs (MNG 2016), 2016.

[73] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, San Diego, California, USA, 2007, pp. 211-222.

[74] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, San Jose, California, USA, 2011, pp. 12-25.

[75] P. Harish, and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in Proceedings of the 14th international conference on High performance computing, Goa, India, 2007, pp. 197-208.

[76] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in Proceedings of the 47th Design Automation Conference, 2010, pp. 52-55.

[77] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, Louisiana, USA, 2012, pp. 117-128.

[78] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU implementation of inclusion-based points-to analysis," in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, Louisiana, USA, 2012, pp. 107-116.

[79] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing Strongly Connected Components in Parallel on CUDA," in Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2011, pp. 544-555.

[80] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, San Antonio, TX, USA, 2011, pp. 267-276.

[81] S. Hong, T. Oguntebi, and K. Olukotun, “Efficient Parallel Graph Exploration on Multi-Core CPU and GPU,” in Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 78-88.

[82] B. Zhang, V. Dave, and M. A. Hasan, *Feature Selection for Classification under Anonymity Constraint*, arXiv preprint, 2015.

[83] T. Garfinkel, and M. Rosenblum, “When virtual is harder than real: security challenges in virtual machine based computing environments,” in Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10, Santa Fe, NM, 2005, pp. 20-20.

[84] M. Price, “The Paradox of Security in Virtual Environments,” *Computer,* vol. 41, no. 11, pp. 22-28, 2008.

[85] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, “Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems,” in Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria, 2010, pp. 353-364.

[86] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, “On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest,” in Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, 2013.

[87] J. Zhong, and B. He, “Medusa: Simplified Graph Processing on GPUs,” *IEEE Trans on Parallel and Distributed Systems*, 2013.

[88] J. DiMarco, and M. Taufer, “Performance Impact of Dynamic Parallelism on Different Clustering Algorithms and the New GPU Architecture,” in Proceedings of the DSS11 SPIE Defense, Security, and Sensing Symposium - Modeling and Simulation for Defense Systems and Applications VI, Baltimore, MD, 2013.

[89] A. Adinetz. "Adaptive Parallel Computation with CUDA Dynamic Parallelism," http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/.

[90] A. Adinetz. "A CUDA Dynamic Parallelism Case Study: PANDA," http://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda/.

[91] Y. Yang, and H. Zhou, “CUDA-NP: realizing nested thread-level parallelism in GPGPU applications,” in Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, Orlando, Florida, USA, 2014, pp. 93-106.

[92] D. Li, H. Wu, and M. Becchi, “Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations,” in

Proceedings of the 44th International Conference on Parallel Processing Beijing, China, 2015.

[93] J. Wang, and S. Yalamanchili, “Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications,” in Proceedings of the 2014 IEEE International Symposium on Workload Characterization, Raleigh, NC, 2014.

[94] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, “Dynamic Thread Block Launch: a Lightweight Execution Mechanism to Support Irregular Applications on GPUs,” in Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015.

[95] D. Li, S. Chakradhar, and M. Becchi, "GRapid: a Compilation and Runtime Framework for Rapid Prototyping of Graph Applications on Many-core Processors."

[96] S. Chakradhar, M. Becchi, and D. Li, *Source-to-source transformations for graph processing on many-core platforms* US,to NEC Corporation, USPTO, 2016.

[97] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in Proceedings of the IEEE International Symposium on Workload Characterization, La Jolla, CA, 2012, pp. 141-151.

[98] R. Nasre, M. Burtscher, and K. Pingali, “Data-driven versus Topology-driven Irregular Computations on GPUs,” in Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, 2013.

[99] R. Nasre, M. Burtscher, and K. Pingali, “Atomic-free irregular computations on GPUs,” in Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, Houston, Texas, 2013, pp. 96-107.

[100] R. Nasre, M. Burtscher, and K. Pingali, “Morph algorithms on GPUs,” in Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, Shenzhen, China, 2013, pp. 147-156.

[101] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM,* vol. 33, no. 8, pp. 103-111, 1990.

[102] D. Li, and M. Becchi, “Deploying Graph Algorithms on GPUs: an Adaptive Solution,” in Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, 2013.

[103] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.: Addison Wesley, 2006.

[104] M. Becchi, and P. Crowley, “A-DFA: a Low-complexity Compression Algorithm for Efficiant Regular Expression Evaluation,” *ACM Trans. Archit. Code Optim.,* vol. 10, no. 1, 2013.

[105] A. Ramamurthy, "Towards scalar synchronization in SIMT architectures," M.S. Thesis, Electrical and Computer Engineering, The University of British Columbia, 2011.

[106] S. Lee, and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.

[107] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*: Addison-Wesley Professional, 2013.

[108] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 3-12.

[109] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, Philadelphia, Pennsylvania, USA, 2005, pp. 181-192.

[110] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv, "NetShield: massive semantics-based vulnerability signature matching for high-speed networks," in Proceedings of the ACM SIGCOMM 2010 conference, New Delhi, India, 2010, pp. 279-290.

[111] G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting evasion attacks at high speeds without reassembly," in Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications, Pisa, Italy, 2006, pp. 327-338.

[112] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue,* vol. 6, no. 2, pp. 40-53, 2008.

[113] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, San Antonio, TX, USA, 2011, pp. 3-12.

[114] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*: MIT press, 2001.

[115] J. Balfour. "CUDA Threads and Atomics."

[116] D. Li, and M. Becchi, "Designing Code Variants for Applications with Nested Parallelism on GPUs," in GPU Technology Conference, 2015.

[117] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen, "Parallel PageRank computation using GPUs," in Proceedings of the Third Symposium on Information and Communication Technology, 2012.

[118] J. L. Greathouse, and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format." pp. 769-780.

[119] "DIMACS Implementation Challenges," http://dimacs.rutgers.edu/Challenges/.

[120] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data.

[121] D. Li, H. Wu, and M. Becchi, "Exploiting Dynamic Parallelism to Efficiently Support Irregular Nested Loops on GPUs," in International Workshop on Code Optimisation for Multi and Many Cores, 2015, pp. 5.

[122] H. Wu, D. Li, and M. Becchi, "Compiler-assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU," in IEEE 30th International Symposium on Parallel & Distributed Processing (IPDPS), 2016.

[123] J. Liu, Y. Chen, and Y. Zhuang, "Hierarchical I/O Scheduling for Collective I/O," in Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, 2013, pp. 211-218.

[124] J. Liu, Y. Lu, and Y. Chen, "In-advance data analytics for reducing time to discovery," in Big Data (Big Data), 2014 IEEE International Conference on, 2014, pp. 329-334.

[125] J. Liu, Y. Che, and S. Byna, "Collective Computing for Scientific Big Data Analysis," in The Eighth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2015.

[126] S. Jones. "Introduction to Dynamic Parallelism," http://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf.

[127] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, 2013, pp. 1-12.

[128] G. E. Blelloch, and G. W. Sabot, "Compiling collection-oriented languages onto massively parallel computers," *J. Parallel Distrib. Comput.,* vol. 8, no. 2, pp. 119-134, 1990.

[129] G. E. Blelloch, *NESL: A Nested Data-Parallel Language*, Carnegie Mellon University, 1992.

[130] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in Proceedings of the 20th international symposium on High performance distributed computing, San Jose, California, USA, 2011, pp. 217-228.

[131] D. J. Quinlan, C. Liao, J. Too, R. P. Matzke, and M. Schordan. "ROSE Compiler Infrastructure," 2015; http://www.rosecompiler.org.

[132] A. V. Adinetz, and D. Pleiter. "Halloc: A fast and highly scalable GPU dynamic memory allocator," https://github.com/canonizer/halloc.

[133] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov, "Spamming botnets: signatures and characteristics," in Proceedings of the ACM SIGCOMM 2008 conference on Data communication, Seattle, WA, USA, 2008, pp. 171-182.

[134] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A Framework for Adaptive Code Variant Tuning," in Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 501-512.

[135] "Profiler User's Guide," http://docs.nvidia.com/cuda/profiler-users-guide/ - axzz3nGyZAhq7.

[136] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in Proceedings of 2010 IEEE International Symposium on Circuits and Systems, 2010.

[137] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional Neural Networks for Speech Recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing,* vol. 22, pp. 10, 2014.

[138] K. Simonyan, and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition."

[139] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification."

[140] Y. Taigman, M. Yang, M. A. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 1701-1708.

[141] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler, "Joint training of a convolutional network and a graphical model for human pose estimation," in Advances in neural information processing systems, 2014, pp. 1799-1807.

[142] R. Lebret, P. O. Pinheiro, and R. Collobert, "Phrase-based image captioning," *arXiv*, no. 1502.03671, 2015.

[143] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in IEEE International Solid-State Circuits Conference (ISSCC), 2016, pp. 262-263.

[144] T. Song, D. Li, and Y. Yao, "Multi-source data oriented flexible real-time information fusion platform on FPGA," in 2011 International Conference on Electronics, Communications and Control (ICECC), 2011, pp. 4401-4404.

[145] T. Tang, R. Luo, B. Li, H. Li, Y. Wang, and H. Yang, "Energy efficient spiking neural network design with RRAM devices," in International Symposium on Integrated Circuits (ISIC), 2014, pp. 268-271.

[146] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in Advances in Neural Information Processing Systems, 2015, pp. 1117-1125.

[147] S. Park, J. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H. Yoo, "An Energy-Efficient and Scalable Deep Learning/Inference Processor With Tetra-Parallel MIMD Architecture for Big Data Applications," *IEEE transactions on biomedical circuits and systems*, 2016.

[148] Nvidia, *GPU-Based Deep Learning Inference: A Performance and Power Analysis*, 2015.

[149] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and A. Ng, "Deep learning with COTS HPC systems," in Proceedings of the 30th international conference on machine learning, 2013, pp. 1337-1345.

[150] Intel. "Intel Deep Learning Framework," https://01.org/intel-deep-learning-framework.

[151] Nvidia. "Accelerate Machine Learning with the cuDNN Deep Neural Network Library," https://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/.

[152] S. Hadjis, F. Abuzaid, C. Zhang, and C. Re, "Caffe con Troll: Shallow Ideas to Speed Up Deep Learning," in Proceedings of the Fourth Workshop on Data analytics in the Cloud, 2015.

[153] D. Li, X. Chen, M. Becchi, and Z. Zong, "Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs," in IEEE International Conference on Smart City/SocialCom/SustainCom (SustainCom), 2016.

[154] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv*, 2014.

[155] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A MATLAB-like environment for machine learning."

[156] Google, *TensorFlow: Large-Scale Machine Learning on heterogeneous Distributed Systems*.

[157] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems,” in Workshop on Machine Learning Systems, 2016.

[158] N. Systems. "Preview release of convolutional kernels," https://github.com/NervanaSystems/nervana-lib-gpu-performance-preview.

[159] Nvidia. "NVIDIA CUDNN GPU Accelerated Deep Learning," https://developer.nvidia.com/cudnn.

[160] S. Chintala. "convnet-benchmarks," https://github.com/soumith/convnet-benchmarks.

[161] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” *arXiv*, 2014.

[162] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks."

[163] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” 2014.

[164] Intel. https://01.org/rapl-power-meter.

[165] Nvidia. "NVIDIA System Management Interface," https://developer.nvidia.com/nvidia-system-management-interface.

[166] B. Ginsburg. "Caffe-OpenMP," https://github.com/borisgin/caffe/tree/openmp.

[167] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*: Pearson Education, 2013.

[168] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong, “Effects of dynamic voltage and frequency scaling on a k20 gpu,” in International Conference on Parallel Processing (ICPP), , 2013, pp. 826-833.

[169] M.-C. Liu, J. Xu, and D. Li, *Learning Convolution Neural Networks on Heterogenous CPU-GPU Platform*, US,to SONY, USPTO, 2015.

[170] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar, “A virtual memory based runtime to support multi-tenancy in clusters with GPUs,” in Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, Delft, The Netherlands, 2012, pp. 97-108.

VITA

Da Li obtained his Bachelor of Engineering from the School of Automation, Beijing Institute of Technology in 2011. After graduation, he attended University of Missouri – Columbia to pursue his Doctoral of Philosophy in Electrical and Computer Engineering under Dr. Michela Becchi's guidance. He interned at Alibaba Group, NEC Laboratories America, AT&T Labs Research and Sony Electronics U.S. Research Center during his study and graduated in July 2016.