

Accelerating Data Parallel Applications via Hardware and Software Techniques

by

Ramin Bashizade

Department of Computer Science
Duke University

Date: _____

Approved:

Alvin R. Lebeck, Advisor

Sayan Mukherjee

Daniel J. Sorin

Hai Li

Lisa Wu Wills

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2020

ABSTRACT

Accelerating Data Parallel Applications via Hardware and
Software Techniques

by

Ramin Bashizade

Department of Computer Science
Duke University

Date: _____

Approved:

Alvin R. Lebeck, Advisor

Sayan Mukherjee

Daniel J. Sorin

Hai Li

Lisa Wu Wills

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2020

Copyright © 2020 by Ramin Bashizade
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

The unprecedented amount of data available today opens the door to many new applications in areas such as finance, scientific simulation, machine learning, etc. Many such applications perform the same computations on different data, which are called data-parallel. However, processing this enormous amount of data is challenging, especially in the post-Moore's law era. Specialized accelerators are a promising solution to meet the performance requirements of data-parallel applications. Among these are graphics processing units (GPUs), as well as more application-specific solutions.

One of the areas with high performance requirements is statistical machine learning, which has widespread applications in various domains. These methods include probabilistic algorithms, such as Markov Chain Monte-Carlo (MCMC), which rely on generating random numbers from probability distributions. These algorithms are computationally expensive on conventional processors, yet their statistical properties, namely, interpretability and uncertainty quantification compared to deep learning, make them an attractive alternative approach. Therefore, hardware specialization can be adopted to address the shortcomings of conventional processors in running these applications.

In addition to hardware techniques, probabilistic algorithms can benefit from algorithmic optimizations that aim to avoid performing unnecessary work. To be more specific, we can skip a random variable (RV) whose probability distribution function (PDF) is concentrated on only one value, i.e., there is only one value to

choose, and the values of its neighboring RVs have not changed. In other words, if a RV has a concentrated PDF, its PDF will remain concentrated until at least one of its neighbors changes. Due to their high throughput and centralized scheduling mechanism, GPUs are a suitable target for this optimization.

Other than probabilistic algorithms, GPUs can be utilized to accelerate a variety of applications. GPUs with their Single-Instruction Multiple-Thread (SIMT) execution model offer massive parallelism that is combined with a relative ease of programming. The large amount and diversity of resources on the GPU is intended to ensure applications with different characteristics can achieve high performance, but at the same time it means that some of these resources will remain under-utilized, which is inefficient in a multi-tenant environment.

In this dissertation, we propose and evaluate solutions to the challenges mentioned above, namely i) accelerating probabilistic algorithms with uncertainty quantification, ii) optimizing probabilistic algorithms on GPUs to avoid unnecessary work, and iii) increasing resource utilization of GPUs in multi-tenant environments.

To my family, especially my best friend and wife, Mohaddeseh, and to the memory of my mom.

Contents

Abstract	iv
List of Tables	xi
List of Figures	xii
Acknowledgements	xv
1 Introduction	1
1.1 Accelerating Markov Random Field Inference with Uncertainty Quantification	3
1.2 Optimizing Markov Random Field Inference via Event-driven Gibbs Sampling on GPUs	4
1.3 Adaptive Simultaneous Multi-tenancy for GPUs	4
1.4 Dissertation Outline	5
2 Background	6
2.1 Markov Random Field	6
2.2 Probabilistic Algorithms	7
2.3 GPU Execution Model	8
3 Accelerating Markov Random Field Inference with Uncertainty Quantification	10
3.1 Motivation	12
3.1.1 Example Application: Motion Estimation	12
3.1.2 Uncertainty Quantification	14

3.2	Design Overview and Challenges	15
3.3	Stochastic Processing Accelerator	18
3.3.1	Overview	18
3.3.2	Stochastic Processing Unit	19
3.3.3	Stochastic Processing Element	21
3.3.4	Accelerator Topology	30
3.3.5	Runtime	34
3.3.6	Limitations and Future Work	34
3.4	Methodology	35
3.4.1	Applications and Metrics	35
3.4.2	HLS Implementation	36
3.4.3	GPU Implementation	37
3.5	Evaluation	37
3.5.1	ASIC Analysis	40
3.6	Related Work	42
3.7	Summary	44
4	Optimizing Markov Random Field Inference via Event-driven Gibbs Sampling on GPUs	46
4.1	Motivation	47
4.1.1	Approximation in Gibbs Sampling	47
4.1.2	Stable Random Variables	48
4.2	Event-Driven Gibbs Sampling	49
4.3	EDGS Implementation for GPUs	52
4.4	Evaluation	54
4.4.1	Methodology	54
4.4.2	Results	58

4.4.3	Takeaways	66
4.5	Related Work	67
4.6	Summary	68
5	Adaptive Simultaneous Multi-tenancy for GPUs	70
5.1	Motivation	75
5.1.1	Resource Requirements	75
5.1.2	Issue Slot Utilization	76
5.1.3	Non-overlapping Execution	76
5.2	Adaptive Simultaneous Multi-tenancy	77
5.2.1	Overview	77
5.2.2	Host-side Service	79
5.2.3	Application Side	80
5.2.4	Kernel Code Transformation	82
5.2.5	Profiling and Pruning the Parameter Space	85
5.2.6	Sharing Policy	86
5.2.7	Example Scenario	87
5.2.8	Limitations and Future Work	90
5.3	Evaluation	91
5.3.1	Platform	91
5.3.2	Benchmark Kernels	91
5.3.3	Single Kernel Performance	93
5.3.4	Multi-Kernel Performance	95
5.4	Related Work	98
5.5	Summary	100
6	Conclusions	102

Bibliography	106
Biography	116

List of Tables

3.1	Values used to calculate Equation 3.5 for Figure 3.8.	29
3.2	Application parameters used in evaluations.	36
3.3	FPGA prototypes results.	38
3.4	Qualitative comparison of proposed accelerator with related work. . .	43
4.1	Parameters for design space exploration.	54
4.2	Application parameters used in the evaluations.	57
4.3	Best performing configurations for the baseline and EDGS.	63
5.1	Application API.	81
5.2	NVIDIA Tesla K40c specifications.	91
5.3	Benchmark kernels characteristics.	92

List of Figures

2.1	Markov Chain Monte Carlo algorithm for Markov Random Field.	6
2.2	Streaming Multiprocessor (SM) components.	9
2.3	GPU components.	9
2.4	Kernel structure.	9
3.1	Data access patterns for performing first-order MRF inference.	13
3.2	Cumulative distribution of pixels per number of unique labels.	14
3.3	Overview of the proposed accelerator’s architecture.	17
3.4	SPU microarchitecture, reproduced from [105].	20
3.5	Architecture of an SPE with two SPUs.	21
3.6	Example singleton 2 access pattern and proposed baking scheme.	25
3.7	Memory banking scheme for LMem.	26
3.8	Maximum per iteration bandwidth usage.	28
3.9	LMem entry and off-chip message bit structure.	30
3.10	Estimated resources for links and crossbars in SPE topology.	31
3.11	Replicating singleton 2 data to eliminate long communication path.	33
3.12	Comparison of output quality results.	38
3.13	Memory space to store data for generating labels histogram.	39
3.14	Accelerator area at different design points.	40
3.15	SPE area breakdown with one and two SPU(s).	41
3.16	SPE power breakdown with one and two SPU(s).	42

3.17	Speedup of a 1024×2 SPE ASIC design compared to GPU.	42
4.1	Normalized number of label changes with respect to total updates. . .	48
4.2	Comparison of update procedure in baseline and EDGS.	50
4.3	An example scenario of updating RVs in a first-order MRF using EDGS.	51
4.4	Normalized slowdown of baseline design points for stereo vision. . . .	55
4.5	Normalized slowdown of EDGS-2 design points for stereo vision. . . .	55
4.6	Normalized slowdown of EDGS-4 design points for stereo vision. . . .	56
4.7	Normalized slowdown of EDGS-8 design points for stereo vision. . . .	56
4.8	Normalized slowdown of EDGS-16 design points for stereo vision. . .	57
4.9	Speedup, end-point error, and skipped update percentages of EDGS for stereo vision.	59
4.10	Normalized slowdown of baseline design points for motion estimation.	60
4.11	Normalized slowdown of EDGS-2 design points for motion estimation.	60
4.12	Normalized slowdown of EDGS-4 design points for motion estimation.	61
4.13	Normalized slowdown of EDGS-8 design points for motion estimation.	61
4.14	Normalized slowdown of EDGS-16 design points for motion estimation.	62
4.15	Speedup, end-point error, and skipped update percentages of EDGS for motion estimation with a 15×15 window.	62
4.16	Speedup, end-point error, and skipped update percentages of EDGS for motion estimation with a 7×7 window.	63
4.17	Speedup, end-point error, and skipped update percentages of EDGS for image segmentation.	65
5.1	Spatial utilization of different resource types in SMs for the benchmark kernels.	75
5.2	Issue slot utilization for the benchmark kernels.	75
5.3	Overview of the proposed adaptive simultaneous multi-tenant system.	78
5.4	The kernel transformation for persistent threads and preemption. . .	83

5.5	An example scenario of two applications running kernels on the GPU simultaneously.	88
5.6	Utilization of various resource types in benchmark kernels.	93
5.7	Performance and register usage of benchmark kernels.	94
5.8	Performance of benchmark kernels with different numbers of CTAs. . .	95
5.9	Normalized STP for combinations of kernels with at least one low-ISU kernel.	96
5.10	Normalized STP for combinations of kernels with high-ISU kernels. . .	96

Acknowledgements

I would like to express my sincere gratitude to all the people without whom I would not have been able to complete my Ph.D. studies. My advisor, Professor Alvin R. Lebeck, supported and patiently guided me even when I was disappointed in myself. I am grateful to Professor Sayan Mukherjee who helped me in my research in probabilistic computing. I also thank my other committee members, Professors Daniel J. Sorin, Hai Li, and Lisa Wu Wills; and other current and former faculty members at Duke Computer Science Department from whom I learned a lot, Professors Benjamin C. Lee, Andrew Hilton, Jeffrey Chase, Landon Cox, and many others. I also appreciate the financial support from Duke University, National Science Foundation, and Intel Corporation.

Past and present staff members at the Duke Computer Science Department were so kind they made everything seem so simple. Pam Spencer, Melanie Eberhart, Alison Hriciga, and most importantly, Marilyn Butler, who always went out of her way to help me in all matters, academic and non-academic alike. Joe Shamblin, and all IT Lab staff in general, helped me a lot with servers and equipment.

I am also thankful to my friends and collaborators at Duke University. Especially Xiangyu Zhang, who was very kind and always willing to help, and Pulkit Misra, a great office-mate and a fellow soccer fan, and others in the Computer Science Department.

Finally, I would like to deeply thank my best friend and wife, Mohaddeseh, who

always supported me through the difficult times. I hope I can return her favor in the many years we will have together.

Introduction

Today there is more data available than ever before. This unprecedented amount of data opens the door to many new applications in areas such as finance [26], scientific simulation [28], machine learning [57], etc. Many such applications perform the same computations on different data, which are called data-parallel. However, processing this enormous amount of data is challenging, especially in the post-Moore's law era [92]. As a result, to address the shortcomings of general-purpose processors in meeting the performance requirements of data-parallel applications, architects have turned to specialized accelerators. Among these are graphics processing units (GPUs), as well as more application-specific solutions.

One of the areas with high performance requirements is statistical machine learning, which has widespread applications in various domains [5, 12, 27, 37, 38, 54, 63, 64, 88]. These methods include probabilistic algorithms, such as Markov Chain Monte-Carlo (MCMC), which rely on generating random numbers from probability distributions. These algorithms are computationally expensive on conventional processors, yet their statistical properties, namely, interpretability and uncertainty quantification compared to deep learning, make them an attractive alternative approach.

Therefore, hardware specialization can be adopted to address the shortcomings of conventional processors in running these applications.

In addition to hardware techniques, probabilistic algorithms can benefit from algorithmic optimizations that aim to avoid performing unnecessary work. These algorithms operate in two modes, sampling and optimization, which provide either the distribution of the result (sampling) or more quickly converge to the final result (optimization). In the optimization mode, most Random Variables (RVs) tend to not change labels very often, and the adoption of approximation techniques makes this more likely to happen. Therefore, we can detect this scenario and further speedup the application by skipping the computations for those RVs. To be more specific, we can skip a random variable (RV) whose probability distribution function (PDF) is concentrated on only one value, i.e., there is only one value to choose, and the values of its neighboring RVs have not changed. In other words, if a RV has a concentrated PDF, its PDF will remain concentrated until at least one of its neighbors changes. Due to their high throughput and centralized scheduling mechanism, GPUs are a suitable target for this optimization.

Other than probabilistic algorithms, GPUs can be utilized to accelerate a variety of applications. GPUs with their Single-Instruction Multiple-Thread (SIMT) execution model offer massive parallelism that is combined with relative ease of programming. They provide thousands of simple cores, register files, and a memory system that is designed to hide latency with high throughput achieved by executing many threads concurrently. The large amount and diversity of resources on the GPU is intended to ensure applications with different characteristics can achieve high performance, but at the same time it means that some of these resources will remain under-utilized. Moreover, the trend in moving toward cloud computing and multi-tenant environments where infrastructure is shared among multiple applications amplifies the inefficiency of leaving resources under-utilized, and further highlights the

importance of better resource utilization.

In the remainder of this chapter, we briefly introduce our proposed solutions to the challenges mentioned above, namely i) accelerating probabilistic algorithms with uncertainty quantification, ii) optimizing probabilistic algorithms on GPUs to avoid unnecessary work, and iii) increasing resource utilization of GPUs in multi-tenant environments.

1.1 Accelerating Markov Random Field Inference with Uncertainty Quantification

A Markov Random Field (MRF) is a powerful graphical model for representing a wide range of applications in statistical machine learning. A MRF encodes the conditional dependence among random variables (RVs). One approach to solving problems represented by a MRF is using probabilistic algorithms such as Gibbs sampling. These methods go through all RVs in the MRF and update them iteratively.

We propose a high-throughput accelerator for MRF inference using Gibbs sampling. We design a tiled architecture that takes advantage of near-memory computing and memory banking and communication schemes tailored to the semantics of MRF. Additionally, we propose a hybrid on-chip/off-chip memory system to efficiently support uncertainty quantification. We implement an FPGA prototype of our proposed architecture using high-level synthesis tools and achieve 146MHz frequency for an accelerator with 32 function units on an Intel Arria 10 FPGA. Compared to prior work on FPGA, our accelerator achieves $26\times$ speedup. ASIC analysis in 15nm technology node shows that our design with 2048 function units running at 3GHz outperforms GPU implementations of motion estimation and stereo vision run on Nvidia RTX 2080 Ti by $135\times$ and $158\times$, respectively, while occupying only 7.7% of the area.

1.2 Optimizing Markov Random Field Inference via Event-driven Gibbs Sampling on GPUs

Probabilistic algorithms such as Gibbs sampling operate in two modes of sampling and optimization, which provide either the distribution of the result (sampling) or more quickly converge to the final result (optimization). We build on three observations in the optimization mode to skip updating RVs that cannot change their label during the current iteration, hence avoiding unnecessary work: i) after the warm-up period, most RVs tend to not change labels very often, ii) an RV can only change its label if either it has a non-concentrated probability distribution function (PDF), or at least one of the RVs on which it is conditionally dependent has changed its label, and iii) approximation techniques make it increasingly likely that RVs have concentrated PDFs. Therefore, we introduce Event-Driven Gibbs Sampling (EDGS), which only updates RVs when necessary. Our analysis shows that 26.3%-30.3% speedup can be gained for two applications, motion estimation and stereo vision on a GPU compared to a baseline that does not take advantage of EDGS. However, for image segmentation, the overheads of our approach are higher than its benefits. In addition, our observations show that in the case of an application with a large number of labels, the approximation technique used actually increases the output quality.

1.3 Adaptive Simultaneous Multi-tenancy for GPUs

GPUs are energy-efficient, massively parallel accelerators that are increasingly deployed in multi-tenant environments such as datacenters for general-purpose computing as well as graphics applications. Using GPUs in multi-tenant setups requires an efficient and low-overhead method for sharing the device among multiple users that improves system throughput while adapting to the changes in workload. This requires mechanisms to control the resources allocated to each kernel, and an efficient

policy to make decisions about this allocation.

We propose adaptive simultaneous multi-tenancy to address these issues. Adaptive simultaneous multi-tenancy allows for sharing the GPU among multiple kernels, as opposed to single kernel multi-tenancy that only runs one kernel on the GPU at any given time and static simultaneous multi-tenancy that does not adapt to events in the system. Our proposed system dynamically adjusts the kernels' parameters at run-time when a new kernel arrives or a running kernel ends. Evaluations using our prototype implementation show that, compared to sequentially executing kernels, the system throughput is improved by an average of 9.8% (and up to 22.4%) for combinations of kernels that include at least one low-utilization kernel.

1.4 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 presents the necessary background on probabilistic algorithms and GPUs. Chapter 3 describes the design of our proposed accelerator for MRF inference with Gibbs sampling, which supports uncertainty quantification, along with the evaluation results on an FPGA prototype and ASIC analysis. The proposed optimization for Gibbs sampling which detects stable RVs and skips updating them is explained in Chapter 4. We implement and evaluate this optimization on a GPU, but we also discuss the limitations of the GPU execution model and provide results that demonstrate the full potential of our proposed approach. In Chapter 5, we present and evaluate our multi-tenant system for GPUs, in addition to the analysis of the effectiveness of the system under various workload scenarios. Finally, the conclusion and future directions are discussed in Chapter 6.

2

Background

In this chapter, we first briefly explain MRF and probabilistic algorithms (Sections 2.1 and 2.2, respectively), and then present the background on GPU execution model (Section 2.3).

2.1 Markov Random Field

A Markov Random Field (MRF) is an undirected graphical model for representing the dependencies among a set of RVs, which satisfies the Markov property, i.e., the future state of the process only depends on the current state and not any previous

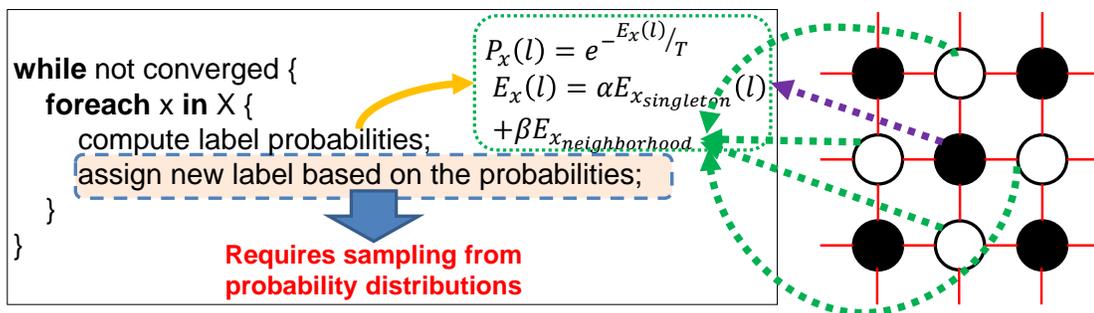


FIGURE 2.1: Markov Chain Monte Carlo algorithm (left) for Markov Random Field (right) inference. Note that sampling is performed in the inner loop.

states. A MRF can be used to represent a wide range of applications in statistical machine learning. In this dissertation, we address problems that are represented by a first-order MRF, but we expect most of the techniques to be applicable to other types of MRF too, albeit after undergoing some modifications.

Figure 2.1 (right) illustrates an example first-order MRF model and its connection with the Gibbs sampling algorithm. In the model, each RV depends on its four immediate neighbors. Due to this structure, the first-order MRF model can be divided into two regions so that all RVs in each region are conditionally independent. This enables us to generate a chromatic schedule to update all RVs in each region in parallel [29, 31, 46, 91].

2.2 Probabilistic Algorithms

Bayesian inference combines new evidence and prior beliefs to update the probability estimate for a hypothesis. Consider D as the observed data and X as the latent random variable. The prior distribution of X is $p(X)$ and $p(D | X)$ is the probability of observing D given a certain value of X . In Bayesian inference, the goal is to retrieve the posterior distribution $p(X | D)$ of the random variable X when D is observed. As the dimensions of D and X increase, it often becomes difficult or intractable to numerically derive the exact posterior distribution $p(X | D)$.

One approach to solving these inference problems is to use probabilistic Markov chain Monte-Carlo (MCMC) methods, e.g., Gibbs sampling [29], that converge to an exact solution by iteratively generating samples for RVs. Figure 2.1 shows this process. For each label l of RV x , we calculate an energy value using the singleton data and neighbor labels ($E_x(l)$). Singleton and neighborhood energies are weighted by parameters α and β , which are application-specific. Having $E_x(l)$, a probability $P_x(l)$ is computed for each label using $\exp(-E_s(l)/T)$, in which T is a per iteration parameter. Once all label probabilities are calculated, we sample from the cumulative

distribution function (CDF) by generating a uniform random number and seeing where it falls in the range of the CDF. The result is the new label for RV x in the current iteration.

Gibbs sampling may be used in one of two modes: i) pure sampling, or ii) optimization. The main difference between these two modes is that in pure sampling, the parameter T is the same for all Gibbs sampling iterations, whereas in optimization (simulated annealing), T gradually decreases to help faster convergence to the final solution. A more detailed explanation of Gibbs sampling is provided elsewhere [29].

In practice, MCMC becomes inefficient for many problems that have high dimensionality (i.e., many RVs) and complex structure. It can require many iterations before convergence, and the inner loop in Figure 2.1 includes generating samples from probability distributions, which is computationally expensive for conventional processors [95]. Therefore, massively parallel platforms such as GPUs can be utilized to address these shortcomings.

2.3 GPU Execution Model

GPUs are massively parallel accelerators that are composed of thousands of simple cores. A large number of cores together with cache, shared memory,¹ register files, and some other components form streaming multi-processors (SMs). All SMs share a last level cache. Figures 2.2 and 2.3 show the architecture of an SM and the GPU.

Figure 2.4 illustrates the structure of a kernel. GPU kernels comprise a large number of threads that execute the same instructions on different data, hence the name Single-Instruction-Multiple-Thread (SIMT). These threads are grouped together to form thread blocks, and a set of thread blocks is called a grid. All thread blocks in a grid have the same dimensions, and all threads of the same thread block can only run on a single SM. A thread block is a logical notion that helps the programmers

¹ Scratchpad memory in NVIDIA terminology is called shared memory.

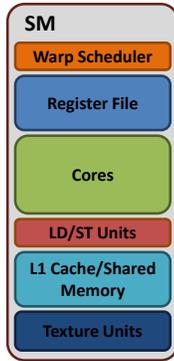


FIGURE 2.2: Streaming Multiprocessor (SM) components.

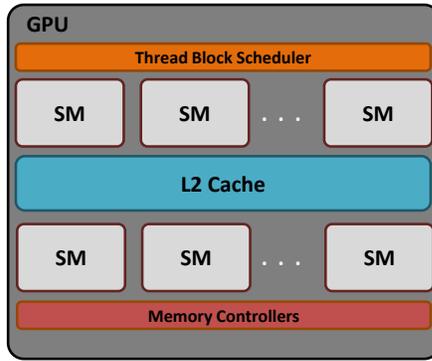


FIGURE 2.3: GPU components.

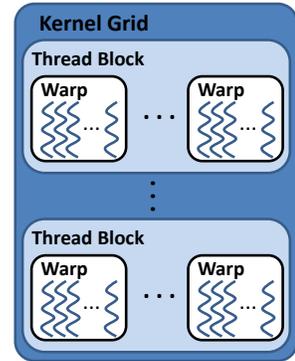


FIGURE 2.4: Kernel structure.

reason about their code. However, a limited number of thread blocks can fit on the device at the same time. We separate these concepts and refer to the physical thread blocks actually running on the GPU as concurrent thread arrays (CTAs). Note that in some other works thread blocks and CTAs are used interchangeably.

When there are enough resources on an SM to host a waiting thread block, the block scheduler dispatches that thread block to the available SM for execution. If there are more than one SM with enough resources, the mapping happens in a round-robin fashion. In each SM then, the warp scheduler dispatches ready warps to execute instructions. A warp is the smallest group of threads that execute in lockstep to reduce the overhead of instruction fetch and control flow. The programmer has no control over the size of a warp. Once a thread block is mapped to an SM, it continues execution until it finishes. In other words, there is no mechanism for preemption or yielding resources (the Pascal [68] and Volta [70] architectures perform context switching, but the programmer does not have control over the operation).

Accelerating Markov Random Field Inference with Uncertainty Quantification

Compared to Deep Neural Networks, probabilistic algorithms make it easier to gain insight into why a result is obtained, and to what degree we can be certain about the results. This can be achieved by quantifying the uncertainty of the result, which is a valuable property of probabilistic algorithms, such as MCMC, and is of utmost importance for some applications, such as many image segmentation applications where quantifying the uncertainty in the segmentation boundaries is crucial (e.g., a surgeon’s decision to resect what sections of a tumor will be impacted by the segmentation generated by an algorithm [16, 60]).

However, the benefits of MCMC come at a price. Since MCMC requires iteratively sampling from probability distributions, it is often computationally intensive. This is due to the significant overhead of sampling in conventional processors [95]. Furthermore, MCMC at first appears to be a sequential algorithm because updating each random variable (RV) depends on the latest value of all other RVs, which means that it may take a long time to finish. Deploying pseudo-random number

generation can help reduce the sampling inefficiency [105]. To avoid the overhead of serial execution, though, one can take advantage of the conditional independence of RVs, i.e., develop a schedule which allows multiple independent RVs to be updated in parallel [29, 31, 46, 91].

In this chapter, we propose an accelerator which builds on these ideas and fuses them with architectural contributions that allow fast and efficient execution of MCMC and minimize the overhead of uncertainty quantification. To be more specific, we propose a tiled architecture to exploit near-memory computing, and the parallelism exposed by taking advantage of the conditional independence of RVs in the first-order Markov Random Field model. We develop memory banking and on-chip communication schemes tailored to the semantics of the model to facilitate a stall-free pipeline.

In addition to the tiled architecture for the accelerator, we propose a hybrid on-chip/off-chip memory system to support uncertainty quantification by maintaining a log of the values that RVs take on, called their labels, throughout the MCMC execution. By carefully analyzing the behavior of two image analysis applications [9, 53], we observe that most RVs take on a limited number of unique labels during the execution, and thus, we design this memory system such that it strikes a balance between on-chip memory capacity and off-chip communication bandwidth.

We implemented an FPGA prototype of our proposed design using Intel High-Level Synthesis (HLS) compiler [40], and developed the necessary runtime to verify and evaluate our implementation using real-world applications and input datasets. The results show that our design achieves a clock rate of 146MHz and a throughput of 4.672B labels/sec on an Arria 10 FPGA. This is a $26\times$ speedup over the previous work [51]. We also perform ASIC analysis on our HLS implementation using Mentor Graphics HLS Compiler [33] and show that an accelerator with 2048 function units running at 3GHz in 15nm technology node [65] outperforms GPU implementations

of motion estimation and stereo vision on an RTX 2080 Ti by $135\times$ and $158\times$, respectively.

3.1 Motivation

One approach to solving inference problems is to use probabilistic Markov chain Monte-Carlo (MCMC) methods that converge to an exact solution by iteratively generating samples for RVs (Figure 2.1). In practice, MCMC becomes inefficient for many problems that have high dimensionality and complex structure. It can require many iterations before convergence, and the inner loop in Figure 2.1 includes generating samples from probability distributions, which is computationally expensive for conventional processors [95] and thus, a specialized accelerator is needed to address these shortcomings.

3.1.1 Example Application: Motion Estimation

To shed more light on the details of the first-order MRF inference using MCMC algorithm, we explain how first-order MRF can be utilized to represent the motion estimation problem and how MCMC with Gibbs sampling can solve it [53]. The goal is to estimate the 2-D motion vectors between two time-varying images, such as two consecutive frames of a video. Figure 3.1 (top) shows a sample input for this problem.

In this example, the target is to compute the motion vector of the pixel in the center of the blue box in Figure 3.1 (bottom-left). To do so, the inner loop in Figure 2.1 must be executed, which includes computing the probability values according to the equations in the figure. In the equations, $P_x(l)$ is the probability that RV x takes on label l , $E_x(l)$ is the energy of label l which depends on the singleton and neighborhood values, and α , β , and T are application parameters. $E_{x_{singleton}}$ depends on two types of singleton data: i) singleton 1, which is the gray-scale value of the pixel

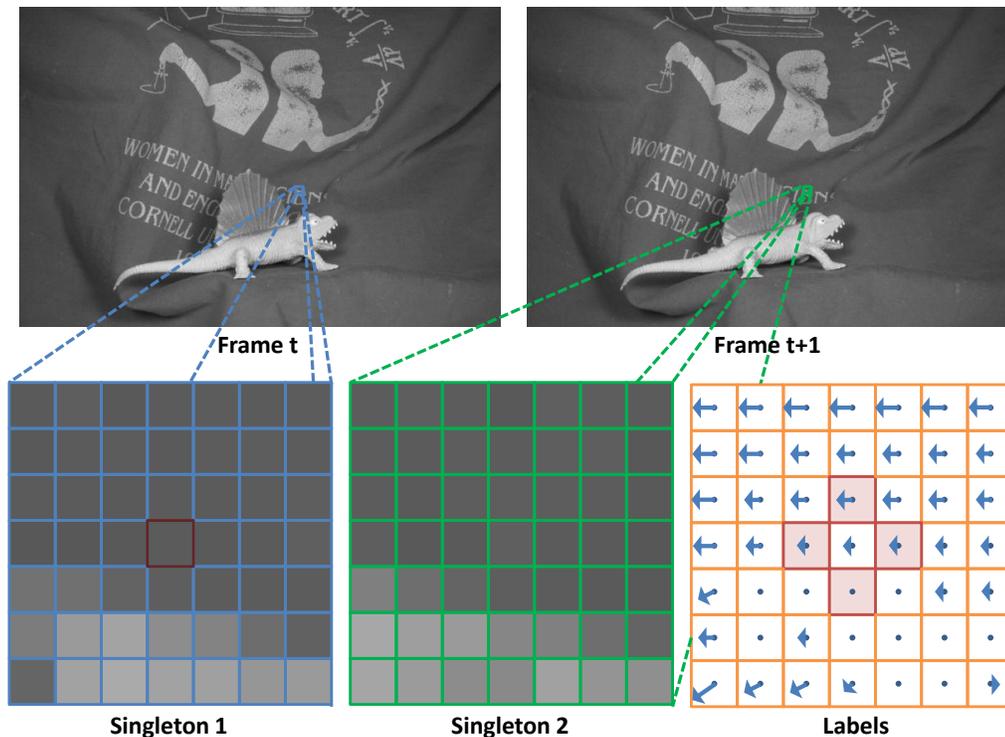


FIGURE 3.1: Data access patterns for performing first-order MRF inference using MCMC to solve motion estimation (*dimetrodon* from Middlebury database [6]).

itself, and ii) singleton 2, which is the gray-scale value of each of the pixels inside the green box in Figure 3.1 (bottom-middle), which form a 7×7 window surrounding the aforementioned pixel corresponding to possible labels (in this application, each motion vector) and come from frame $t + 1$. The pattern of singleton 2 is application specific. Generally, for each RV, $E_{x_{neighborhood}}$ may depend on the latest labels of all other RVs. However, for the first-order MRF model, $E_{x_{neighborhood}}$ is calculated using the current labels of the top, down, left, and right neighbors of the pixel, the shaded boxes shown in Figure 3.1 (bottom-right), each of which are motion vectors themselves. This neighborhood pattern is fixed for the first-order MRF model. Once the probabilities for all possible labels are calculated, they are used to create a probability distribution function (PDF), which in turn is used for sampling and determining the new label for the pixel.

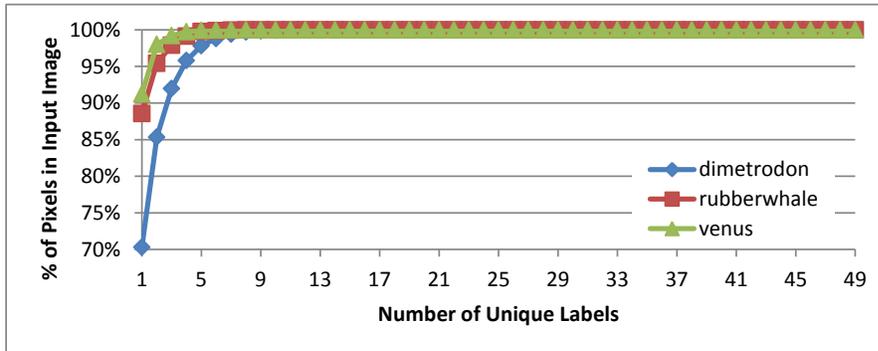


FIGURE 3.2: Cumulative distribution of pixels per number of unique labels in three input datasets [6] for motion estimation.

This process must be repeated for all pixels in frame t for a certain number of iterations (until the algorithm converges to the final solution) to obtain their motion vectors. CMOS specialization and pseudo-random number generation can be used to accelerate the computations required for updating each pixel. Previous work proposes a function unit for this purpose [105], which is briefly reviewed in Section 3.3.2. Furthermore, the structure of the MRF model provides opportunities for parallelism (explained in Section 3.3.3). However, there are challenges in realizing this parallelism due to the memory access patterns which require careful memory banking and access scheduling that are discussed in Sections 3.3.3 and 3.3.3.

3.1.2 Uncertainty Quantification

Probabilistic models and algorithms are “conceptually simple, compositional, and interpretable” [30], and provide the opportunity to determine why a given result is obtained. This is due to two reasons: i) models such as MRF inherently have transparent structures, and ii) these algorithms allow for quantifying the uncertainty to evaluate the confidence in the obtained result. Uncertainty quantification can be achieved by collecting a histogram of the RV’s labels after the warm-up period of the MCMC (i.e., the iterations at the beginning of the algorithm before mixing has

happened), which can then be used to derive statistics such as mode, variance, etc., that illuminate the uncertainty associated with the final result. In the example of image segmentation guiding a surgeon’s decision regarding tumor resection, if the variance in the final result is high, then the surgeon might decide to remove a larger section to be safe, without removing too much of the tissue.

However, naïvely storing these data imposes a significant memory capacity, bandwidth, and processing overhead and therefore, a more scalable solution is required for uncertainty quantification. Fortunately, there is an opportunity for optimization because after warm-up, the RVs tend to take on only a limited number of labels. Figure 3.2 illustrates this fact. In three input datasets [6] for motion estimation, which has 49 labels, at most only 14.7% of pixels take on more than two unique labels during the second half of the iterations (i.e., iterations 1500-3000 in this experiment). This allows for having a limited on-chip memory space to store more frequently picked labels, and occasionally send the rest to the off-chip memory. Section 3.3.3 presents a hybrid on-chip/off-chip memory system for collecting the histogram of labels based on this analysis.

3.2 Design Overview and Challenges

The characteristics of MCMC and MRF, covered in Sections 2.2 and 3.1.1, guide our design choices for the proposed accelerator. In this section, we provide an overview of our design, the challenges we face, and our proposed solutions.

MCMC is an iterative algorithm in which the computations of each iteration depend on those of the previous one (Section 2.2). Therefore, we decide to use on-chip memory to store data and intermediate iteration results to avoid frequent costly off-chip communication that uses up significant bandwidth and imposes high latencies. Furthermore, due to the structure of the first-order MRF, all computations are local, i.e., updating RVs only needs data from nearby memory locations. *Thus, we propose*

to use a tiled architecture where each tile has its own memory and is responsible for computations on the portion of the graphical model stored in its memory. This allows us to expose the existing parallelism in first-order MRF and take advantage of near-memory computing, and eliminates the need for complex centralized coordination. The tile’s architecture is discussed in detail in Section 3.3.3.

The proposed architecture needs a communication infrastructure to efficiently transfer data among tiles when needed. Particularly, due to singleton 2’s application-specific nature, designing such a communication infrastructure for it without sacrificing flexibility can be challenging. *We propose a topology and data mapping scheme tailored to the first-order MRF characteristics, which together ensure no communication longer than one hop will be required,* and therefore, the overheads of a full-blown Network-on-Chip (NoC) are avoided. Although our design is tailored to the first-order MRF neighborhood structure, it supports arbitrary accesses to the singleton 2 memory (S2Mem). In other words, our proposed topology and data mapping scheme for singleton 2 do not limit the MRF applications the accelerator can run. Sections 3.3.4 and 3.3.5 explain the proposed network topology and data mapping schemes. Different types of data are involved in the computations of the MRF model (i.e., read-only singletons and read-write labels) and unique access patterns to these data. Thus, we dedicate separate memories to each data type.

Moreover, exposing the potential parallelism inherent in the model requires a suitable scheduling technique that allows updating multiple conditionally independent RVs simultaneously. We use known techniques to develop a chromatic schedule of conditionally independent RVs that can be updated in parallel. The implication of this scheduling technique is that in addition to the parallelism between tiles, we can include more than one function unit in each tile to exploit intra-tile parallelism. However, this introduces competing accesses to S2Mem. Furthermore, labels memory (LMem) must be accessed at four different locations for each RV. *Therefore, we*

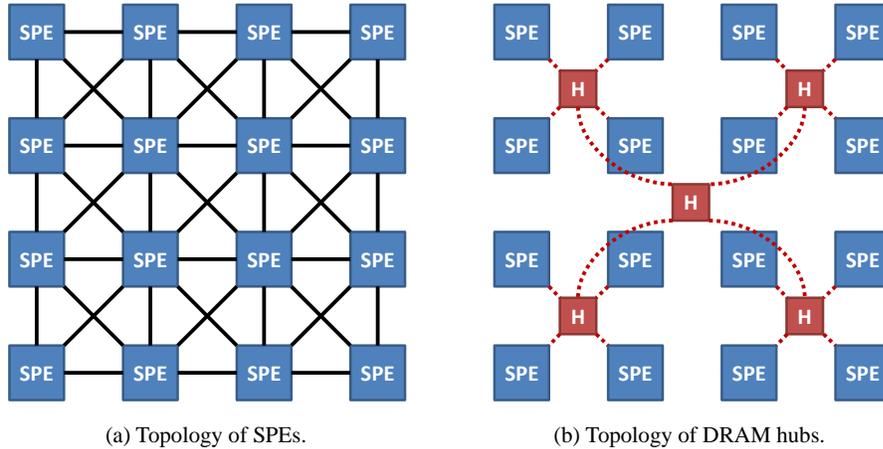


FIGURE 3.3: Overview of the proposed accelerator’s architecture. Communication between SPEs is bi-directional, but between SPEs and DRAM hubs and among DRAM hubs is uni-directional. The diagonal links between SPEs are only for communicating singleton 2 data, whereas the vertical and horizontal links transfer both label data and singleton 2 data.

utilize memory banking mechanisms for each of S2Mem (Section 3.3.3) and LMem (Section 3.3.3) to facilitate stall-free execution in tiles.

Finally, to support uncertainty quantification, we need to track how many times a label is chosen for a given RV. A naïve implementation requires either i) having enough counters on the chip to keep track of all possible labels for all RVs, which is prohibitive in terms of area, or ii) sending the result of all label updates off chip, which needs significant communication bandwidth. *Because of the limited memory capacity on the chip, and inspired by the insights from Figure 3.2, we design a hybrid on-chip/off-chip memory system to store the histogram for RVs during the MCMC iterations in the form of a log, which will be helpful for uncertainty quantification.* We augment LMem entries with counters that keep track of how many times each label has been picked, and only transfer this information to off-chip memory when necessary (Section 3.3.3).

3.3 Stochastic Processing Accelerator

3.3.1 Overview

Our proposed architecture for accelerating first-order MRF inference using MCMC with Gibbs sampling is presented in this section. Figure 3.3 shows an overview of the accelerator’s architecture. It is composed of a number of computation tiles or SPEs (Section 3.3.3), and DRAM hubs that route communication to the off-chip DRAM. Each SPE is responsible for processing a portion of the input to take advantage of near-memory computing and exploit the inherent parallelism of the model. It comprises a number of Stochastic Processing Units (SPUs) [105], which perform the main MCMC computations (Section 3.3.2), in addition to a scheduler which sequences through RVs (Section 3.3.3), a portion of the singleton memories (Section 3.3.3) and the label memory (Section 3.3.3), on which it performs the MCMC updates, and communication components that transfer data between different SPEs and between the accelerator and the off-chip DRAM that stores the histogram log of the labels. Each SPE is connected to all its nearest SPEs and only communicates with those (Section 3.3.4). To ensure that communications longer than one hop are never required, appropriate data mapping and replication schemes are adopted, which are handled by the runtime (Section 3.3.5).

Figure 3.3 shows an overview of the proposed accelerator’s architecture. It is composed of a number of computation tiles or Stochastic Processing Elements (SPEs), and DRAM hubs that route label histogram entries to the off-chip DRAM. Each SPE is responsible for processing a portion of the input to take advantage of near-memory computing and exploit the inherent parallelism of the model. It comprises a number of Stochastic Processing Units (SPUs), which perform the main MCMC computations (Section 3.3.2), in addition to a scheduler which sequences through RVs (Section 3.3.3), a portion of the singleton memories (Section 3.3.3) and the label

memory (Section 3.3.3), on which it performs the MCMC updates, and communication components that transfer data between different SPEs and between the accelerator and the off-chip DRAM that stores the histogram log of the labels. Each SPE is connected to all its nearest SPEs and only communicates with those SPEs (Section 3.3.4). To ensure that communications with SPEs more than one hop away are never required, appropriate data mapping and data replication schemes are adopted which are handled by the runtime (Section 3.3.5).

3.3.2 Stochastic Processing Unit

Zhang et al. propose a Gibbs sampling function unit, called Stochastic Processing Unit (SPU), that utilizes specialization and pseudo-random number generation to accelerate MCMC computations [105]. Figure 3.4 demonstrates the microarchitecture of this function unit. It is composed of four main pipeline stages, namely, energy computation (Equation 3.1), dynamic energy scaling (Equation 3.2), energy to probability conversion (Equations 3.3 and 3.4), and sampling.

$$E(l) = \alpha E_{singleton}(l) + \beta \sum E_{neighborhood} \quad (3.1)$$

$$E_s(l) = E(l) - E_{min} \quad (3.2)$$

$$P_s(l) = (2^{P_{bits}} - 1) \times \exp(-E_s(l)/T) \quad (3.3)$$

$$P_{tr}(l) = \lfloor 2^{\lceil \log_2 P_s(l) \rceil} \rfloor \quad (3.4)$$

Energy computation takes the singleton data and neighbor labels, all 6-bit values, and computes the energy of a possible label, $E(l)$ in Equation 3.1, where α and β are application parameters. Next, $E(l)$ is dynamically scaled by subtracting the minimum energy of all labels from it to maximize the dynamic range. Energy values (raw and scaled) are 8-bit unsigned integers. The scaled energy $E_s(l)$ is then converted

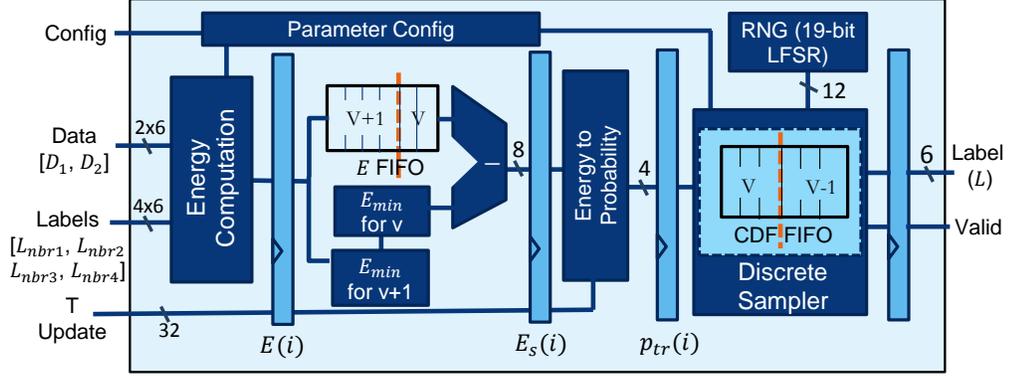


FIGURE 3.4: SPU microarchitecture, reproduced from [105].

to a scaled probability represented by a 4-bit unsigned integer. The original probability (real number in $[0, 1]$) is calculated using $\exp(-E_s(l)/T)$, in which T is a per iteration parameter. To avoid using floating-point function units, though, the probability is scaled using Equation 3.3, and then truncated using Equation 3.4. $P_{bits} = 4$ ensures the scaled probability is in $[0, 16]$, which allows for representing the number using 4 bits. Afterward, Equation 3.4 approximates the scaled probabilities to the nearest power of two, i.e., $P_{tr} \in \{0, 1, 2, 4, 8\}$. The possible values of $P_{tr}(l)$ can be pre-computed and stored in a look-up table (LUT). These values must be updated if T changes. The last stage generates a sample per RV based on all $P_{tr}(l)$, where L is the number of labels, using the least significant twelve bits of a 19-bit Linear Feedback Shift Register (LFSR) to implement the inverse transform sampling. The SPU's throughput is one RV update per L cycles, if it receives the appropriate input (i.e., neighborhood labels and singleton data) at every cycle. Our goal in Sections 3.3.3 and 3.3.4 is to design an architecture that ensures this condition is realized.

The SPU can be used in one of two modes: i) pure sampling, or ii) optimization. The main difference between these two modes is that in pure sampling, the parameter T is the same for all Gibbs sampling iterations, whereas in optimization (simulated annealing), T gradually decreases to help faster convergence to a final solution [29].

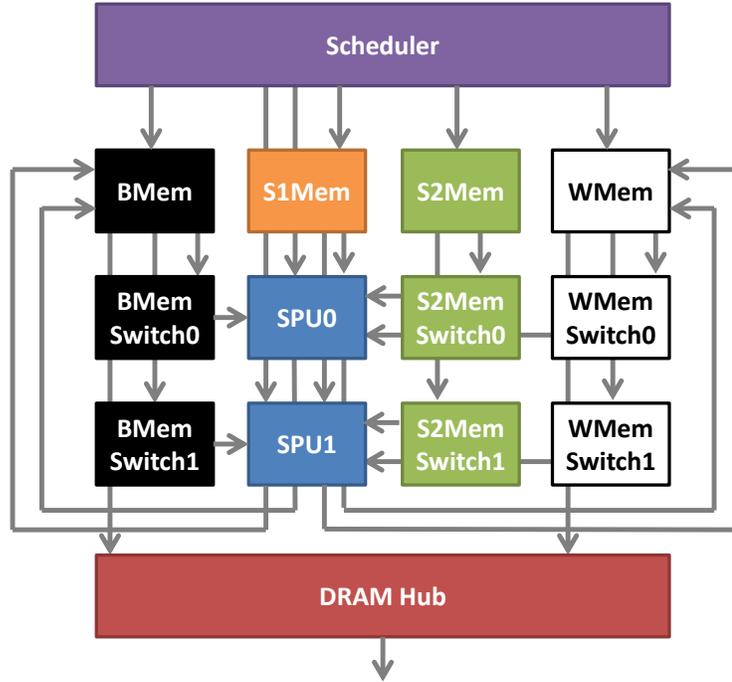


FIGURE 3.5: Architecture of an SPE with two SPUs. For the sake of clarity, non-local outgoing communications from BMem, WMem, and S2Mem modules and incoming communications to BMem Switches, WMem Switches, and S2Mem Switches have been omitted. BMem and WMem send data to the BMem and WMem Switches in the top, down, left, and right neighbors. S2Mem sends data to the S2Mem Switches in all the eight neighbors shown in Figure 3.3a.

The implication of this difference is that in pure sampling mode, when the algorithm converges to a final solution, the estimated distribution of a RV can be generated by collecting the histogram of the latest N samples. Since in this work we are interested in the uncertainty quantification capability of MCMC, we only focus on the pure sampling mode. However, the proposed accelerator can operate in optimization mode as well.

3.3.3 Stochastic Processing Element

An SPE incorporates the components that carry out the operations needed to feed the necessary data to SPUs every cycle and write back the result of their computations to the LMem. These operations include sequencing through RVs and generating

memory addresses corresponding to the singleton and neighborhood data, reading the data at those addresses and passing them to the appropriate SPU, and writing the results of the computations back to the correct addresses in LMem. Furthermore, to maintain the histogram of labels, it might be necessary that while writing data back to LMem, some data should be sent to the off-chip DRAM. Figure 3.5 shows the SPE's components and the interactions between them. The next section is dedicated to describing the various components inside an SPE, followed by a more detailed explanation of scheduling and different types of memory in the SPE.

Components

Scheduler: This component's main job is to generate the update schedule and coordinate the operations of most of the other components in an SPE. It interacts with SPUs and various memory blocks, and its functionalities include sending the SPUs some parameters including the T in MCMC equations in Figure 2.1, and other information such as whether a pixel is on the boundary or whether it is a black or a white pixel (to determine the destination of the computations result). It also sends the computed addresses to different memory blocks, so that they can return the requested data to the SPU.

Singleton 1 Memory: Denoted by S1Mem in Figure 3.5, it stores singleton 1 data as the name suggests, or in the example of motion estimation in Section 3.1.1, the data in the blue box in Figure 3.1. It receives addresses from the Scheduler and sends data to the SPUs once for every RV.

Singleton 2 Memory: Similar to S1Mem, it is referred to as S2Mem in Figure 3.5, and stores singleton 2 data (i.e., the data in the green box in Figure 3.1). Because each singleton 2 corresponds to an individual label, as opposed to singleton 1 which is fixed for all labels of the same RV, S2Mem receives a base address from the Scheduler, and computes addresses for the appropriate singleton 2 data point for

each label by reading from an offset look-up table (LUT) populated by the runtime in an application-by-application basis. For instance, in the case of motion estimation, the LUT stores the offsets that define the 7×7 window shown in Figure 3.1. It then sends that data to S2Mem Switches for every label. Since this data is needed for every label at every SPU, it is required that multiple reads from different addresses be issued at the same cycle. We address this problem by devising a banking scheme that is described in Section 3.3.3. Both singleton memories are read-only, meaning they get initialized in the beginning by the runtime and will never change throughout the execution.

Singleton 2 Switch: These switches receive data from the appropriate S2Mem, i.e., either the local S2Mem or one of the memories in one of the eight neighbors, and send it to the SPUs they are connected to.

Label Memories: BMem and WMem in Figure 3.5, together form the LMem. These memories store the results of the computations done by the SPUs. They receive addresses from the Scheduler to send neighborhood data to the corresponding switches, which in turn send those data to the SPUs. They also receive the new labels from SPUs. Although neighborhood data is needed only once per RV, due to the model's structure, multiple reads must be issued simultaneously to provide the data necessary for beginning the computations to the SPUs. We solve this problem by banking the LMem and pipelining accesses to them. In addition to storing the labels computed by the SPUs, LMem is also part of the hybrid on-chip/off-chip memory system that stores the information required for generating the labels histogram. LMem is explained in more detail in Section 3.3.3.

Label Switches: These switches are similar in functionality to S2Mem Switches, i.e., they receive neighborhood data from the local LMem as well as the LMem in the top, down, left, and right SPEs and pass them to their corresponding SPU.

Updating Order and Inter-variable Parallelism

In general, MCMC is a sequential algorithm since updating each RV depends on the latest value of all other RVs. However, as explained in Section 3.1.1, in the first-order MRF model, each RV is only conditionally dependent on its top, down, left, and right neighbors. This means there is an opportunity to develop a chromatic schedule for updating conditionally independent variables in parallel and thus, significantly reduce the execution time. For first-order MRF, this schedule is a simple checkerboard scheme which divides the random field into a black (BMem) and a white (WMem) subset, where all RVs in each subset are independent [31, 46, 51]. In our proposed accelerator, the Scheduler in each SPE is responsible for generating this schedule. The Scheduler first goes through all black RVs, then flushes the pipeline of all other components, and repeats the same process for white RVs.

Another benefit of this chromatic schedule is that it puts restrictions on access types to different parts of LMem, i.e., while black variables are being updated, there will be no writes issued to WMem and vice versa. This allows for simplifying the memory structure by dividing it into a black and a white region, knowing that the Scheduler takes care of avoiding conflicting accesses to these regions.

Singleton Memory Structure for Multiple SPUs

According to the MCMC details explained in Section 3.1.1, there are potentially two types of singleton data, both of which are stored in read-only memories: 1) singleton 1, which is always present and is required once for a RV, and 2) singleton 2, which, if present, is required for each possible label a RV can take on. The implication of singleton 2's access pattern is that if we decide to have more than one SPU in an SPE to amortize the area of the Scheduler and other control logic, servicing singleton 2 reads becomes a challenge. Figure 3.6a illustrates this problem with an example of two SPUs in an SPE running in parallel. Multiple pieces of singleton 2 data at

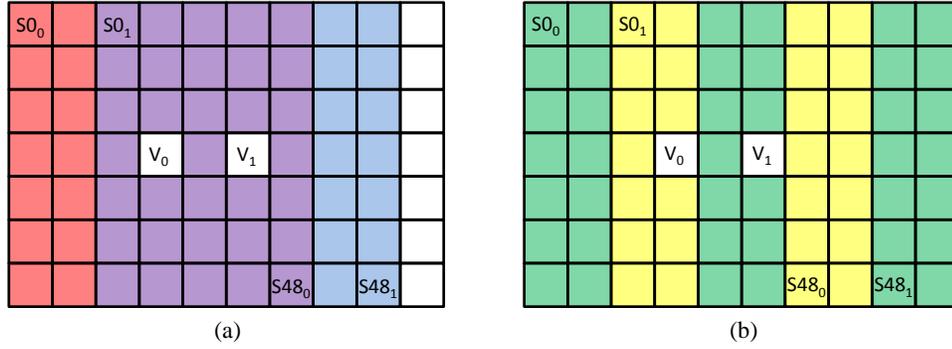


FIGURE 3.6: (a) An example of the singleton 2 access pattern for motion estimation in an SPE with two SPUs. Boxes shaded in red and purple illustrate the singleton 2 window for RV V_0 , and boxes shaded in purple and blue show that window for V_1 , and (b) the proposed banking scheme to solve the multiple simultaneous accesses issue. Each color represents one bank. Locations denoted by $S0_0$ and $S0_1$ must be accessed together, and so is the case for $S48_0$ and $S48_1$.

different addresses must be read at the same cycle. There are three possible solutions to accommodate this access pattern:

1. S2Mem must support a read size larger than one singleton 2, and an intermediate register must handle the feeding of data to the appropriate SPU. This option also allows exploiting the temporal locality of singleton 2, i.e., a piece of data can be read once and used multiple times if it is required for multiple RVs. However, determining when to issue new reads, shifting and moving data around, and developing an update schedule that matches this design make it complicated, particularly due to the application-specific patterns of singleton 2 accesses.
2. S2Mem must be a multi-port structure to be able to straightforwardly read the required data from it. Nevertheless, multi-port memories are area- and power-hungry and are generally not preferable [99]. This option also does not take advantage of singleton 2's temporal locality.
3. S2Mem must be divided into separate banks with only one port each, that are

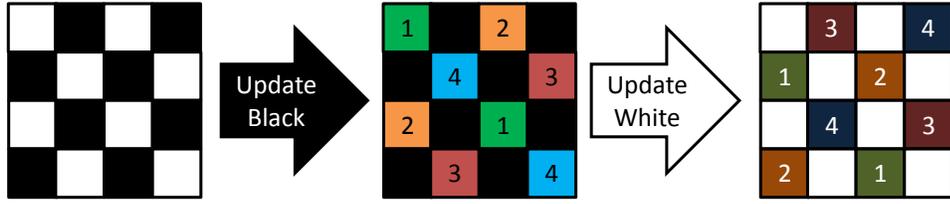


FIGURE 3.7: Memory banking scheme for white and black sections of LMem. Note that while updating RVs in BMem, their neighbors are in WMem and vice versa.

accessed simultaneously. Similar to the previous solution, the drawback of this design is that it too necessitates reading the same piece of data multiple times. However, it allows for a simpler Scheduler and memory structure and therefore, we choose this option for S2Mem.

Our proposed banking scheme exploits the knowledge of the update order discussed in Section 3.3.3. More specifically, we take advantage of the stride of two consecutive RVs in the same row. Because we know the next RV will always be two locations ahead and the singleton 2 access pattern is the same for all RVs, it logically follows that the next singleton 2 will also be two locations ahead. Thus, we put every two columns of singleton 2 in a separate bank, for a total number of banks equal to the number of SPUs inside the SPE. Figure 3.6b demonstrates this for an example SPE with two SPUs. The runtime is responsible for correctly populating these banks.

Labels Memory and Labels Log

Similar to singleton 1, neighborhood data is needed once for each RV (Section 3.1.1). Nevertheless, since in the first-order MRF the neighborhood structure consists of four RVs, a simple monolithic single-port memory does not accommodate the requirements of the model. A key difference with S2Mem, though, is that neighborhood data is needed only once for each RV. Therefore, reads to the LMem can be

pipelined to provide data to multiple SPUs in an SPE. To overcome the challenge of accessing four locations in LMem simultaneously, we use a memory banking scheme shown in Figure 3.7. This pattern is repeated to cover the input. It ensures that for every RV, its top, down, left, and right neighbors reside in unique banks.

In addition to storing the labels data, LMem is also a part of the on-chip/off-chip memory system that collects the labels histogram for uncertainty quantification. Collecting an accurate histogram needs a counter per each possible label, in our case 64, and each counter must be able to hold the maximum number of iterations, which could be 10-12 bits. Storing such a huge amount of data on chip is neither practical nor efficient. Fortunately, it is not necessary either.

As Figure 3.2 shows, a significant portion of RVs take on only a few unique labels after the warp-up period has passed. This inspired us to have room for a few labels and their corresponding counters on chip, and once a counter is saturated or a new label is selected that is not present in the LMem, send a message consisting of the evicted label's ID, the RV's address, and the count associated with it, to an off-chip memory. This data is stored in the form of a log, which at the end of the execution is processed by the runtime and translated into a histogram. The operation of this memory structure is similar to a write-back, write-allocate, no fetch-on-write cache. The main advantage of such a design is that unlike normal caches where data travels in both directions (i.e., on-chip to off-chip and vice versa), here data only go out from on-chip memory and hence, with deep enough FIFOs to store the messages until they can be sent to the off-chip memory, the computation units will not be forced to stall.

The remaining challenges are: 1) choosing an efficient replacement policy, and 2) determining the optimal size of the on-chip LMem (i.e., how many label+counter pairs to keep per RV). We considered two replacement policies, Least Frequently Picked (LFP), and Least Recently Picked (LRP). Intuitively, LFP makes the most sense because we want to keep the label that is selected most often on-chip. However,

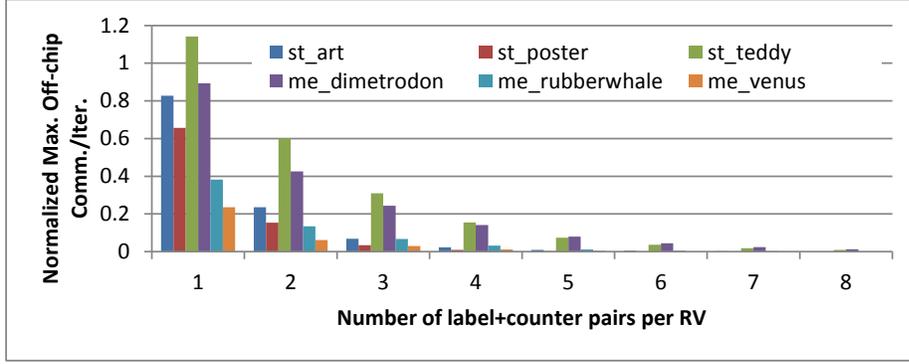


FIGURE 3.8: Maximum values of Equation 3.5 for three input data sets for stereo vision [9] and motion estimation [6] per LMem size. The replacement policy is least recently picked.

it is both more complicated to implement and more sensitive to the time we start to collect the histogram. To be more specific, if we start collecting the histogram too soon, i.e., before the end of the warm-up period, it is possible that a label which is not among the top few most frequently picked labels overall is picked enough times that it prevents the actual frequent labels from remaining in the on-chip memory. LRP, however, avoids this by evicting the aforementioned label because it is not selected anymore after the warm-up period. For these reasons, we choose LRP as the replacement policy.

To determine the size of the on-chip memory, we must take into account the trade-off between this size and the off-chip bandwidth and the size of the off-chip log. Ideally, we want the smallest on-chip memory that the off-chip bandwidth allows. We use Equation 3.5 to arrive at this size:

$$\frac{\frac{\#SPUs}{\#Labels} * EvictionRate * MessageSize}{Bandwidth} < 1 \quad (3.5)$$

$\#SPUs$ is the total number of SPUs in the accelerator, $\#Labels$ is the number of possible labels a RV can take on (an application-specific value), $EvictionRate$ is the rate at which labels are evicted to off-chip memory, $MessageSize$ is the size of the

Table 3.1: Values used to calculate Equation 3.5 for Figure 3.8.

Parameter	Value
#SPUs	2048
#Labels	Stereo Vision: 28, 30, 56; Motion Estimation: 49
Message Size	32 bits
Bandwidth	512 bits/cycle

messages in bits, and *Bandwidth* is the available off-chip bandwidth. Equation 3.5 indicates that the amount of off-chip communication must not exceed the available bandwidth. Figure 3.8 shows the maximum value of Equation 3.5 for three input data sets for stereo vision [9] and motion estimation [6] for different sizes of LMem (Table 3.1 lists the values used to compute the result of Equation 3.5). To generate this graph, we first collect a trace of the labels of all RVs at every iteration. We then process this trace to simulate the behavior of our proposed LMem with sizes of 1-8 label+counter pairs per RV. The figure indicates that with a LMem large enough to hold only two label+counter pairs per RV, the off-chip bandwidth utilization will not exceed 60% of the available bandwidth. Therefore, we select two label+counter pairs per RV.

Given the number of label+counter pairs per RV, and the number of RVs that the accelerator supports (1M RVs in this example), Figure 3.9 shows the structure of a LMem entry and a message sent to off-chip memory. It is possible to change the width of the counter and the address fields depending on the size of the target input sets. It is also possible to reduce the required bits for addressing by directing messages from certain SPEs to pre-defined offsets in the off-chip DRAM.

With this proposed scheme, writing to the LMem is transformed to a read-modify-write, where depending on the current labels in the target LMem entry and the new label, a re-ordering of the data in the entry or sending a message to the off-chip memory may be needed.

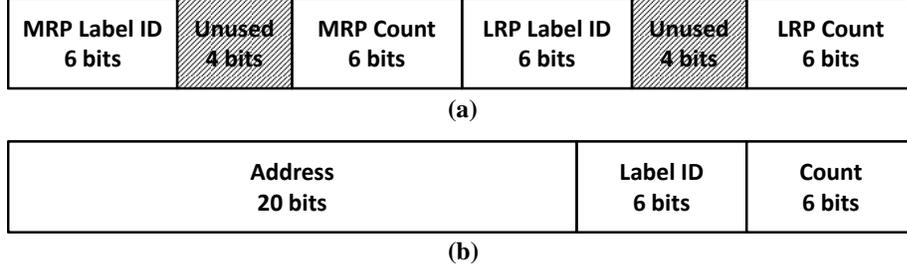


FIGURE 3.9: Structure of (a) a LMem entry, and (b) a message to off-chip memory, in which the RV address and label ID are together the bin identifier in the histogram. For the LMem entry, we assume a 32-bit word due to FPGA limitations. More area savings are possible in ASIC design.

Algorithm 1 Writing to the label memory.

```

1: procedure WRITELABEL(addr, new_lbl)
2:   {mrp_lbl, mrp_cnt, lrp_lbl, lrp_cnt} ← mem[addr]
3:   if new_lbl = mrp_lbl then
4:     if mrp_cnt = MAX_VALUE then
5:       {addr, mrp_lbl, MAX_VALUE} → DRAM
6:       mem[addr] ← {mrp_lbl, 1, lrp_lbl, lrp_cnt}
7:     else
8:       mem[addr] ← {mrp_lbl, mrp_cnt + 1, lrp_lbl, lrp_cnt}
9:   else if new_lbl = lrp_lbl then
10:    if lrp_cnt = MAX_VALUE then
11:      {addr, lrp_lbl, MAX_VALUE} → DRAM
12:      mem[addr] ← {lrp_lbl, 1, mrp_lbl, mrp_cnt}
13:    else
14:      mem[addr] ← {lrp_lbl, lrp_cnt + 1, mrp_lbl, mrp_cnt}
15:  else
16:    {addr, lrp_lbl, lrp_cnt} → DRAM
17:    mem[addr] ← {new_lbl, 1, mrp_lbl, mrp_cnt}

```

3.3.4 Accelerator Topology

There are two networks in our proposed accelerator, one that connects the SPEs which transfers label and singleton 2 data (Section 3.3.4), and another that connects the label memories in SPEs to the interface to off-chip memory (Section 3.3.4). These two networks carry traffic with different characteristics and requirements, and thus, have different topologies.

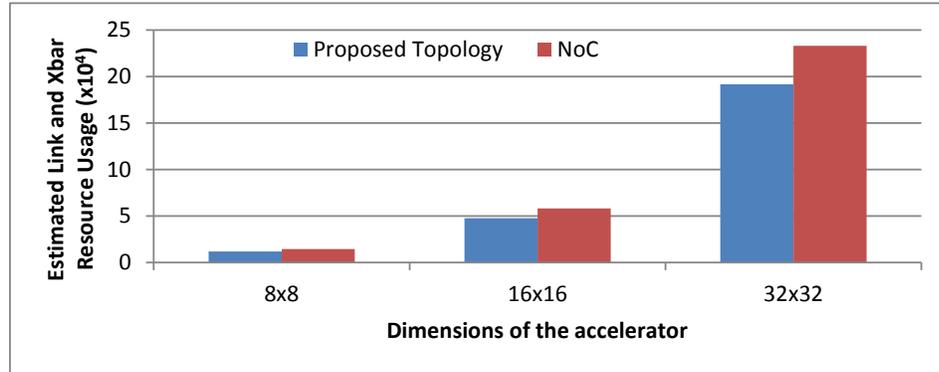


FIGURE 3.10: Comparison of the estimated amount of resources needed for links and crossbars in the proposed topology and a 2-D mesh NoC.

SPE Network

Communications among SPEs follow a regular pattern, e.g., when an SPU inside an SPE needs a piece of data that resides in the left neighbor, the right neighbor also needs a piece of data at that same address in the current SPE. This is true for both label data and singleton 2 data communication, and is due to the characteristics of the first-order MRF model and the even distribution of work to SPEs guaranteed by the runtime.

Due to this regular pattern of communication, we propose to use a topology in which every SPE is connected to its top, down, left, and right SPEs for transferring label data, and to all eight nearest neighbors (as shown in Figure 3.3a) to communicate singleton 2 data. The runtime then ensures that all the data an SPE could possibly need reside in those SPEs to which it is directly connected. This way, there is no need for a full-fledged Network-on-Chip (NoC). Whenever SPEs need data from their neighbors, they also push data in the opposite direction because their neighbor needs the same type of data. This ensures a stall-free execution. Additionally, this topology avoids the area overhead of a NoC router. Nevertheless, crossbars and links are still required for moving data around. Figure 3.10 demonstrates the estimated

amount of resources needed for our proposed topology compared to a 2-D mesh NoC, for accelerators with three different dimensions in which each SPE has 2 SPUs. The values are derived from Equations 3.6, 3.7, 3.8, and 3.9, in which D denotes the dimension of the accelerator, S shows the number of SPUs per SPE, NN refers to our proposed topology, and NoC indicates the 2-D mesh NoC. Also, $I : O$ means a crossbar with I input and O output ports. To estimate the amount of resources needed for a crossbar, we simply multiplied its number of input and output ports. We substituted S with 2 and added the two values for each topology to generate Figure 3.10. Although this is not an accurate measure of the required resources (for instance, one could argue that links and crossbars should not have the same weight), combined with the reduced design complexity enabled by our proposed topology, we chose that over a generic NoC.

$$NN_{Links} = 2(2(D - 1)D(S + 1) + 2(D - 1)^2S) \quad (3.6)$$

$$NN_{XB} = D^2(2(4 : 8) + (S : 9S) + 8(2 : S) + (9S : S)) \quad (3.7)$$

$$NoC_{Links} = 2(2(D - 1)D(S + 1)) \quad (3.8)$$

$$NoC_{XB} = D^2(2(4 : 8) + (S : 5S) + 8(2 : S) + (5S : 5S)) \quad (3.9)$$

DRAM Hub Network

Unlike the regular communications between SPEs which depending on the application can be intensive during some periods of execution, communications between SPEs and DRAM Hubs are irregular and designed to be infrequent. Although we cannot guarantee the latter is always the case, our workload characterization discussed in Section 3.3.3 demonstrates that by carefully designing the memory system, we can achieve this in practice. Guided by this assumption, we use a tree topology for the DRAM Hub network, as shown in Figure 3.3b. Every four SPEs are connected to one DRAM Hub, forming a region, and then every four DRAM Hubs are connected

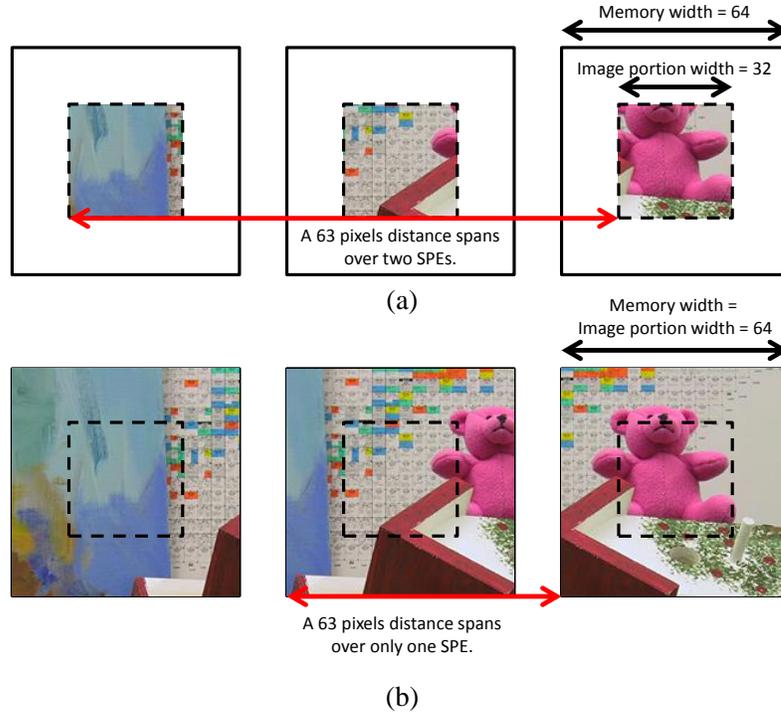


FIGURE 3.11: (a) Example of an input set for stereo vision in which communications with SPEs farther than one hop is necessary to transfer singleton 2 data, and (b) solving this problem by replicating singleton 2 data.

to each other. This pattern continues up until the interface with the off-chip DRAM. This topology is scalable and does not cause communication with the DRAM to become a bottleneck. Furthermore, communication with the DRAM is one-way during the execution, i.e., data only flows from the accelerator toward DRAM. Therefore, the high latency of off-chip communication does not stall the execution pipeline of the accelerator. At the DRAM interface, messages are aggregated to form 512-bit lines and are written to the DRAM. A log index is kept at the DRAM interface which is both used for writing new values to DRAM throughout the execution, and reading valid values from the DRAM at the end of execution.

3.3.5 Runtime

The runtime is responsible for handling memory allocation, parameter initialization, data padding when necessary, and data placement and movement. In this section, we only discuss data padding and data placement because the other operations are only a matter of implementation. Data padding might be necessary depending on the input size, because work must be distributed among SPEs evenly as the correct communication of data between SPEs relies on this assumption. Another assumption that our proposed communication scheme builds upon is that all the data an SPE might possibly need, whether label or singleton 2 data, must be available in at most a single hop distance. Although this assumption always holds for label data (only the labels of immediate neighboring RVs are needed), it might not necessarily hold for singleton 2 depending on its access pattern and how small the input data set is. Figure 3.11a illustrates this with an example. In this example, the application is stereo vision [9] in which the singleton 2 accesses could reach 63 locations to the left of any given RV. In this case, if the width of the portion of the input assigned to each SPE is smaller than the reach of singleton 2, then communication longer than one hop will be necessary. Fortunately, replicating the singleton 2 is a simple fix for this problem and the runtime can handle it.

3.3.6 Limitations and Future Work

Some limitations of our proposed accelerator are inherent to the specific Gibbs sampling algorithm selected, e.g., the lack of support for continuous RVs. Some other limitations are due to our design and implementation. For example, because of the design choice to represent labels with six bits, the proposed accelerator cannot support problem instances with more than 64 labels. However, 64 labels is enough for many applications [51, 79], and expanding the number of supported labels is future work. In addition, previous work shows that slightly increasing the bit width in some

places in the SPU datapath increases the result quality to be closer to floating-point software implementations. Incorporating those changes in our design is future work, and we expect the effects on the area to be small.

Another limitation specific to our design is that it only supports first-order MRF. Although this model can represent a wide range of applications [49, 54, 88], a more flexible label memory design is required to expand the coverage to more applications which we intend to do in the future. In addition to the label memory, the singleton 2 memory and the interconnect between the tiles will need modification too, but we expect that many of the same techniques used in our design will be able to guide the design of a more generalized accelerator too.

Additionally, we plan to optimize the execution time of MCMC by avoiding unnecessary RV updates. To be more specific, we can skip a RV whose PDF is concentrated on only one value, i.e., there is only one label to choose, and the labels of its neighborhood has not changed. In other words, if a RV has a concentrated PDF, its PDF will remain concentrated until something in its neighborhood (i.e., the only changing input for MCMC update) changes. We implement and evaluate this optimization for GPUs in Chapter 4, and show that it is an effective way to gain additional speedup. Thus, adopting this optimization in a hardware accelerator is future work.

3.4 Methodology

3.4.1 Applications and Metrics

We use two image analysis applications to evaluate our design, namely, motion estimation [53] and stereo vision [9, 79]. Motion estimation is covered in detail in Section 3.1.1. Stereo vision reconstructs the depth information of objects in a field captured from two cameras by matching the pixels between the two images. The farther the location of the pixel in the two images, the deeper it is in the field. Therefore, sin-

Table 3.2: Application parameters used in evaluations.

Motion Estimation						
	α	β	T	Labels	Size	Iters
dimetrodon	6	6	1	49	584×388	3000
rubberwhale					584×388	
venus					210×190	
Stereo Vision						
	α	β	T	Labels	Size	Iters
art	6	7	2	28	348×278	3000
poster				30	435×383	1500
teddy				56	450×375	3000

gletton 1 data comes from the right view, and singleton 2 comes from each of the L pixels preceding the target pixel in the left view, where L is the number of labels in the model.

We evaluate each application using three input image sets from Middlebury [6, 79]. Table 3.2 summarizes the parameters used for each input. Parameter names correspond to those in Figure 2.1. To generate outputs, we calculate the mode of the labels in the last 1,000 iterations for each input. We compare the results against a MATLAB implementation which uses double-precision floating-point to assess the quality of the results. We use end-point error (EPE) as the metric for evaluating motion estimation results [53], and bad-pixel (BP) percentage as the metric for stereo vision [79].

3.4.2 HLS Implementation

To implement the FPGA prototype and perform ASIC analysis of our proposed accelerator, we use High-Level Synthesis (HLS) to compile code written in C++ to Hardware Description Language (HDL). We utilize Intel HLS compiler from Quartus 18.0 [40] to implement the FPGA prototype. We implement the components shown in Figure 3.5 individually, and then connect them together using Platform Designer [42], and synthesize the final design for a Programmable Acceleration Card (PAC)

with Arria 10 GX FPGA [41] using Quartus 17.1. We utilize Open Programmable Acceleration Engine (OPAE) 1.2 to develop the runtime that controls the FPGA prototype.

For ASIC analysis, we use Mentor Catapult [33] and adapt our C++ code to use Algorithmic C datatypes [61] which allow for using custom precision data types in the HLS design. We utilize Design Compiler [87] to synthesize our design using a 15nm library [65] to derive area and power results for non-memory logic. In addition, we use CACTI 7.0 [93] to estimate the area and power of memory components. Since the smallest technology node in CACTI is 22nm, we conservatively use those numbers for area and power calculation. Power numbers are calculated by feeding the switching activity based on a 32-label application to Design Compiler, conservatively assuming all input ports switch every time new data arrives.

3.4.3 GPU Implementation

We implement the two applications using CUDA [66], and conduct evaluations on an Nvidia RTX 2080 Ti GPU [71]. The same chromatic schedule for updating conditionally independent RVs in parallel is used in the GPU implementation. We applied spatial-tiling [75] to take advantage of spatial locality, i.e., we divided the input image into equal-sized rectangles and assigned each region to a specific thread block. The size of the thread blocks were 16×16 , which means they covered a 32×16 region (due to the chromatic schedule we use for updates).

3.5 Evaluation

Figure 3.12 demonstrates the application result quality for the two applications discussed in Section 3.4.1, using their corresponding metrics. The results are consistent with prior work [105]. It is possible to further improve the quality of the results by slightly modifying the bit width of some places in the SPU datapath, which is

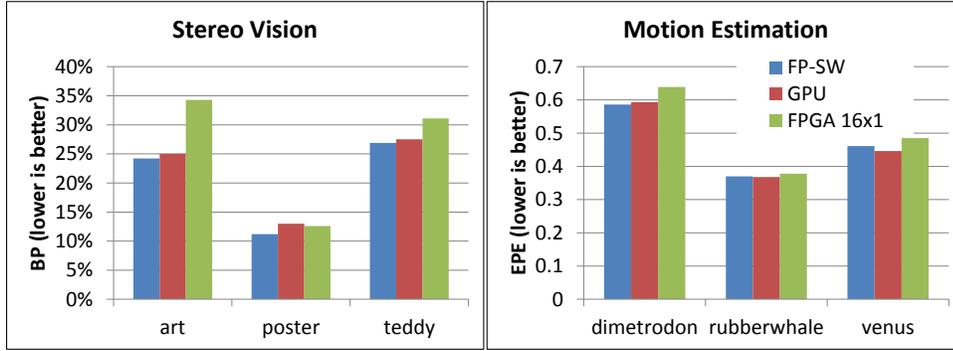


FIGURE 3.12: Comparison of output quality results between a MATLAB floating-point implementation, a GPU implementation, and our FPGA prototype.

Table 3.3: FPGA prototypes results.

Design Point	16×1	16×2	Device Max
Adaptive Logic Module (ALM)	182,690	247,815	427,200
20K-bit Memory Block (M20K)	1,376	1,545	2,713
DSP	160	320	1,518
Clock Rate	185MHz	146MHz	667MHz
Total Performance (Labels/sec)	2.96B	4.672B	

discussed in detail elsewhere [105].

Performance

Resource requirements and clock rate for two design points on an Intel Arria 10 GX FPGA are presented in Table 3.3. (16×2 means 16 SPEs and two SPUs/SPE.) These FPGA implementations support up to 256K RVs, big enough for the input datasets used in our evaluations. We only implement designs with one and two SPU(s)/SPE, because as discussed in Section 3.3.3, accesses to LMem are pipelined for SPUs in the same SPE. The implication is that if an application has less labels than there are SPUs in an SPE, then additional SPUs will not be utilized. Since we can guarantee that all applications have at least two labels (otherwise there would be no problem to solve), we implement designs with at most two SPUs/SPE.

As it is expected, the design with two SPUs/SPE occupies less than twice the

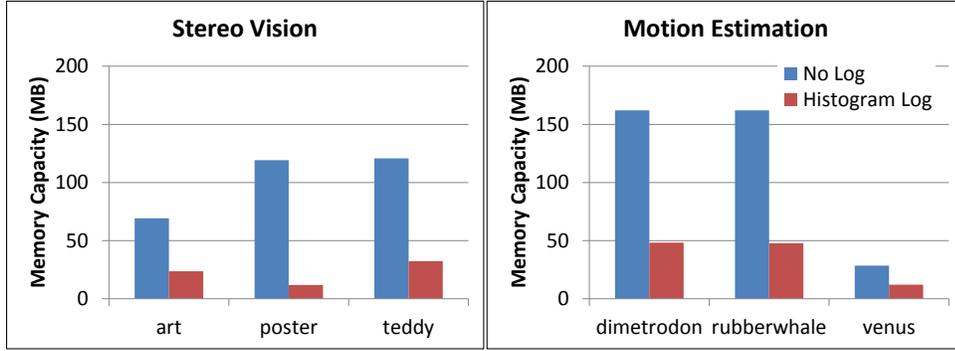


FIGURE 3.13: Memory space to store data for generating labels histogram. “No Log”: results are written to off-chip memory every iteration. “Histogram Log”: off-chip capacity used to store the log, and the on-chip memory that stores the labels.

area of the other design, which means the area of some components (e.g., scheduler, memory control, etc.) are successfully amortized. However, the clock rate drops due to the more complicated routing required between the SPEs.

Equation 3.10 shows that compared with prior work on FPGAs [51], our proposed accelerator (16×2 design point) achieves $26 \times$ speedup. (See Table II in [51].) This is mainly due to the better memory design and avoiding off-chip communication as much as possible. Nevertheless, superior performance is not the only advantage of our work. Due to our proposed tiled architecture, efficient memory system design, and incorporating the scheduling logic into SPEs, our design provides far more flexibility compared to [51], which only supports MRF models up to a certain row size.

$$\frac{4.672 * 10^9 \text{ labels/sec}}{2 \text{ labels/sample} * 88.588 * 10^6 \text{ samples/sec}} = 26.37 \quad (3.10)$$

Uncertainty Quantification

The amount of memory used to store information for generating the labels histogram in our FPGA prototype and a hypothetical design which stores all labels in off-chip memory are compared in Figure 3.13. Another baseline would be a design that has a counter for each possible label of each RV. However, if the counters reside on the chip,

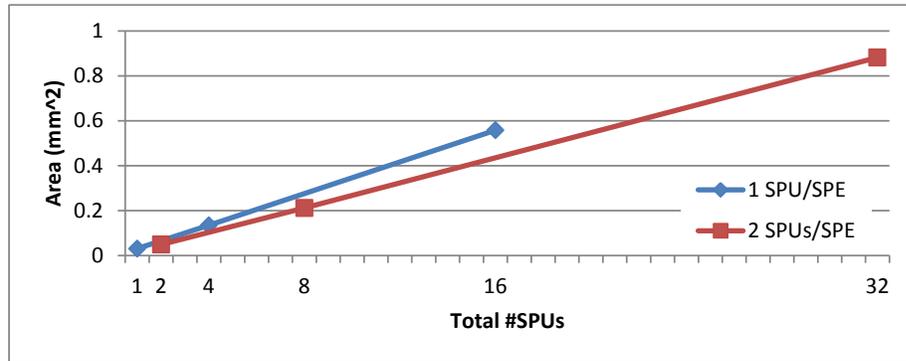


FIGURE 3.14: Accelerator area at different design points.

they require enormous area (e.g., with 10-bit counters, 80 bytes for each RVs) which will not be utilized for applications with less than 64 labels. Even for applications with a large number of labels, our analysis (Figure 3.2) shows that only a few unique labels are chosen throughout the execution. Moreover, if the counters are stored in off-chip memory, two-way communication is needed to update the counts. Therefore, we do not include it in our comparisons. The figure shows that our hybrid on-chip/off-chip memory system and logging scheme saves an average of 71% in memory space for generating the histogram.

3.5.1 ASIC Analysis

We implement and synthesize ASIC designs with one, four, and 16 SPEs, each with one and two SPU(s)/SPE. Figures 3.15-3.16 show the area and power breakdown by component of an SPE with one and two SPU(s). Compared with the one SPU/SPE design, we observe the area and power amortization trend in two SPUs/SPE design for ASIC designs too. The design with two SPUs/SPE uses 20.6% less area and 21.2% less power per SPU compared to that with one SPU/SPE. In addition, another indicator of successful overhead amortization is that the fraction of area and power used by SPUs, which perform the main computations, increases with two SPUs/SPE

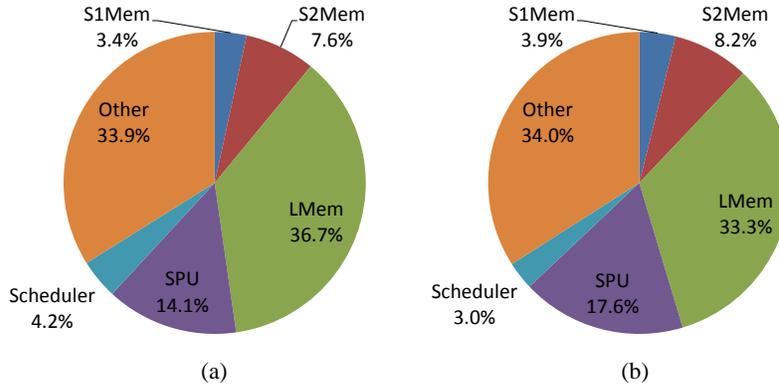


FIGURE 3.15: SPE area breakdown with (a) one SPU, and (b) two SPUs. Total area: (a) $30,760\mu\text{m}^2$, and (b) $48,862\mu\text{m}^2$.

by 24.8% and 28.2%, respectively. (Note that these are post-synthesis results, place and route might yield different numbers).

The overall accelerator area results for multiple design points are depicted in Figure 3.14. In each design, 1K RV is assigned to each SPU, i.e., a design with 2048 SPEs with 1SPU/SPE supports the same amount of memory as a design with 1024 SPEs with 2 SPUs/SPE, and both support Full-HD images. As expected, due to the homogeneity of the proposed tiled architecture, the area scales almost linearly with the number of SPEs. We predict the area of an accelerator with 1024 SPEs, each with two SPUs by extrapolating this graph and adding the area of the required DRAM hubs to be 58mm^2 , which is 92.3% smaller than an RTX 2080 Ti GPU (754mm^2) [71].

In terms of performance, compared to an RTX 2080 Ti, the aforementioned accelerator achieves $135\times$ and $158\times$ speedup for motion estimation and stereo vision, respectively, as demonstrated in Figure 3.17. The implication of these numbers is that our proposed accelerator can process Full-HD images at 30fps with 64 labels for 1500 iterations per frame.

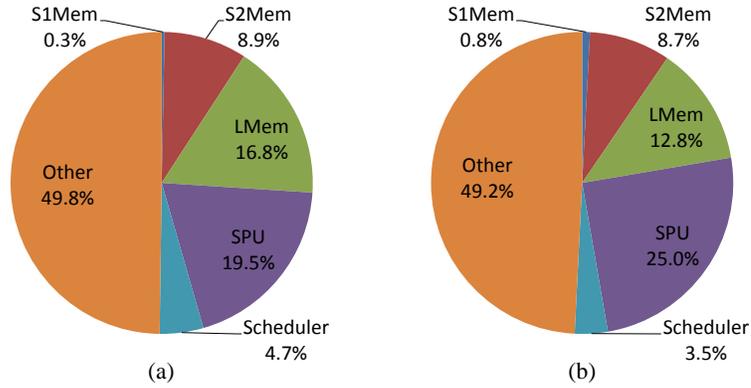


FIGURE 3.16: SPE power breakdown with (a) one SPU, and (b) two SPUs. Total power: (a) 50.076mW, and (b) 78.917mW.

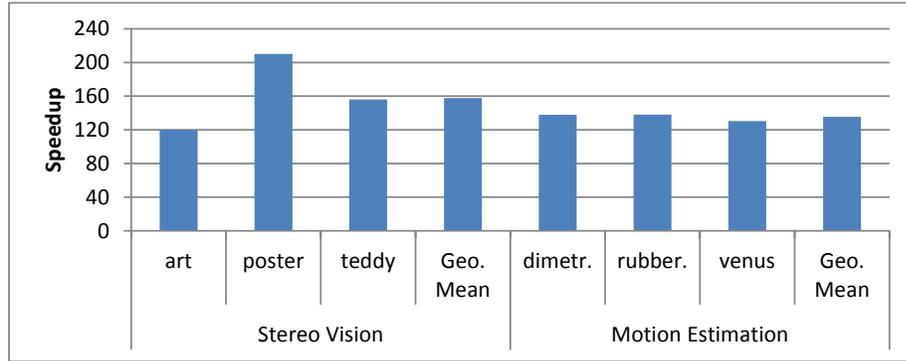


FIGURE 3.17: Speedup of a 1024x2 SPE ASIC design at 15nm, compared to an RTX 2080 Ti GPU. GPU execution times in ms: art: 242, poster: 303, teddy: 839, dimetrodon: 1082, rubberwhale: 1082, venus: 204.

3.6 Related Work

Methods for parallelizing Gibbs sampling based on the conditional independence of RVs have been proposed [31, 46, 91]. Our work takes advantage of similar principles to create a schedule to update conditionally independent RVs in parallel.

So et al. [82] present a custom data layout approach in multiple memory banks for array-based computations. Wang et al. [96] propose a polyhedral model that attempts to detect memory bank conflicts for generalized memory partitioning in

Table 3.4: Qualitative comparison of our proposed accelerator with other Bayesian inference accelerators.

	Application Flexibility	Input Flexibility	Memory System	Uncertainty Quantification
Gibbs tile [46]	Medium	High	On-chip	×
SPU [105]	Medium	–	–	–
FlexGibbs [51, 52]	Medium	Low	Off-chip	High Overhead
VIP [39]	High	Very High	3D-stacked	×
AcMC ² [7]	Very High	Very High	Off-chip	Trades Accuracy for Memory Capacity
This work	Medium	High	Hybrid On-chip/Off-chip	Efficient and Accurate

HLS. Cilaro and Gallo [18] present a lattice-based method that takes advantage of the Z-polyhedral model [35] for program analysis and adopt a partitioning scheme based on integer lattices. Escobedo and Lin [21] use memory space tessellation to find patterns in data accesses and cover the memory access space with the found pattern. In other works, Escobedo and Lin [22, 23] present approaches that create the data dependence graph of memory accesses in the iteration domain, and use graph coloring to assign data elements to memory banks. These works address the problem of determining the proper memory structure for a specific problem that uses HLS, whereas our goal in this paper, while being an instance of this problem, is to design a fixed memory structure that can support a wide range of applications. Moreover, in these works, the communication among different compute units is not accounted for, which could further constrain the data placement solution.

Table 3.4 presents a summary of the differences between our accelerator and the related work (Application and Input Flexibility in the table refer to the diversity of applications and input sizes supported). To the best of our knowledge, other MCMC and inference accelerators in the literature do not address the memory subsystem

challenges and uncertainty quantification as comprehensively and effectively as this work. Jonas [46] presents a tiled Gibbs sampling architecture, but does not include an efficient memory system design and support for uncertainty quantification. Zhang et al. [105] propose and analyze a microarchitecture for a Gibbs sampling function unit, but does not propose an actual accelerator design. and only includes back of the envelope calculations for performance. Ko et al. [51, 52] design an FPGA-based parallel Gibbs sampling accelerator for MRF, which does not provide the level of flexibility supported in this work. Hurkat and Martínez [39] propose a vector processor for deterministic inference algorithms which utilizes 3-D stacking to address high memory bandwidth requirements. Banerjee et al. [7] design a compiler that transforms probabilistic models into hardware accelerators. Although their work supports more general models, it produces a new accelerator per model and is different from our work, in that our goal is to design an accelerator that supports a reasonable range of problems. Additionally, support for uncertainty quantification in their work is limited due to the use of on-chip counters, which can impose significant overheads when the problem size grows, and the adoption of binning and Bloom filters [25] to approximate the histogram when the number of entries is high.

3.7 Summary

Probabilistic algorithms, such as MCMC, are an attractive approach in statistical machine learning which offer interpretability and uncertainty quantification in the final results. These algorithms, however, require probabilistic computations which are not a good fit for conventional processors. We propose a specialized accelerator to significantly improve the performance of MRF inference using MCMC compared to general-purpose processors. Our proposed architecture takes advantage of near-memory computing, as well as memory banking and communication schemes tailored to the characteristics of first-order MRF model. The accelerator also supports un-

certainty quantification by employing a hybrid on-chip/off-chip memory system. We prototype the proposed design with 32 function units on an Arria 10 FPGA using Intel HLS compiler and achieve a 146MHz clock rate. The FPGA implementation outperforms the previous work by $26\times$. ASIC analysis using Mentor Graphics HLS compiler shows that in 15nm technology, the accelerator runs at 3GHz and achieves $135\times$ and $158\times$ speedup over GPU implementations of motion estimation and stereo vision, respectively.

Optimizing Markov Random Field Inference via Event-driven Gibbs Sampling on GPUs

An important application with a large amount of data-parallelism that can benefit from the high throughput offered by GPUs is statistical machine learning, which has widespread applications such as image analysis [54], natural language processing [27], global health [37], wireless communications [38], autonomous driving [5], etc. [12, 63, 64, 88]. Many such approaches use probabilistic algorithms, e.g., Markov chain Monte-Carlo (MCMC), which can be adopted to create generalized frameworks for solving a wide range of problems, and in some cases, are the only viable approach to solve certain classes of problems, e.g., high-dimensional inference.

Orthogonal to the techniques mentioned in Chapter 3 to parallelize the execution of Gibbs sampling, we can adopt algorithmic optimizations to avoid performing unnecessary work and thus, further speed up the execution.

In this chapter, we build on three observations that reveal when RVs cannot change their labels during the current iteration: i) after the warm-up period in the optimization mode, most RVs tend to not change labels very often, ii) a RV can only

change its label if either it has a non-concentrated probability distribution function (PDF), i.e., it has non-zero probabilities of taking on multiple labels, or at least one of the RVs on which it is conditionally dependent has changed its label, i.e., its PDF has changed, and iii) approximation techniques make it increasingly likely that RVs have concentrated PDFs.

We introduce event-driven Gibbs sampling (EDGS). In this scheme, queues are used to keep track of RVs that must be updated. To be more specific, a RV is added to the queue if i) another RV on which it is conditionally dependent changes its value, or ii) it does not have a concentrated PDF. We implement EDGS for GPUs to take advantage of the high amount of parallelism provided by them. Our evaluations show that 26.3%-30.3% speedup can be gained for two image analysis applications, namely, motion estimation and stereo vision, with some loss in the quality of the results. However, for another image analysis application, i.e., image segmentation, the overheads of our approach outweigh its benefits. Our observations also indicate that for motion estimation with a large number of labels, the approximation technique used in our work actually increases the quality of the final results.

4.1 Motivation

4.1.1 *Approximation in Gibbs Sampling*

In the optimization mode, as the iterations proceed, picking the labels with higher energy becomes less and less likely, thus making the probability distribution function (PDF) more concentrated. Since the goal in the optimization mode is to more quickly converge to a final solution, there is an opportunity to utilize approximation techniques to further accelerate this process. One such approximation, inspired by a hardware Gibbs sampling accelerator [105], is truncating very small label probabilities to zero to prevent the algorithm from picking them. Equation 4.1 demonstrates this approximation.

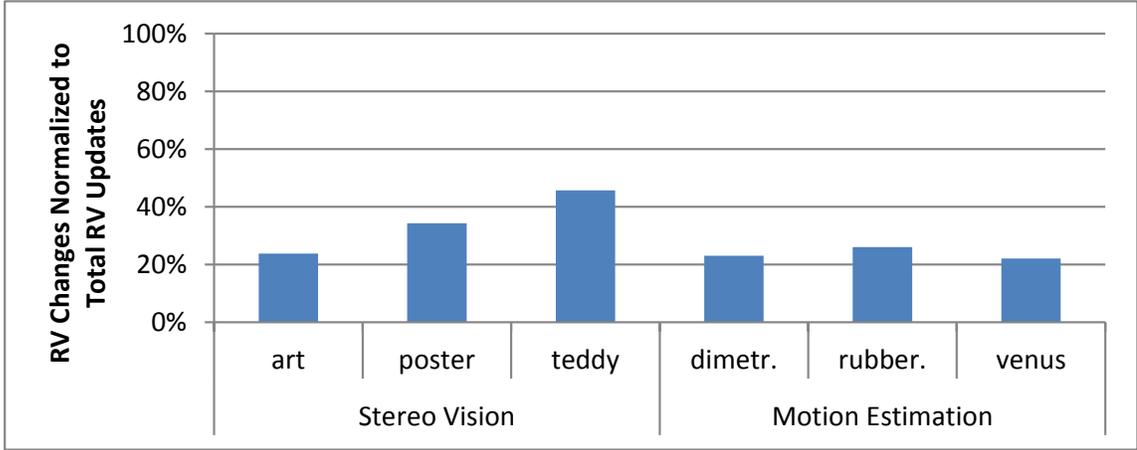


FIGURE 4.1: Number of times that RVs change labels normalized to the total number of times they are updated, for stereo vision and motion estimation.

$$P_{tr}(l) = \begin{cases} P(l), & \text{if } \frac{\max_{l \in L} P(l)}{P(l)} < C \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

In the equation, $P(l)$ is the probability of selecting label l (from all possible labels L) before truncation, $P_{tr}(l)$ is the truncated probability, and C is the cutoff threshold used for truncation. Although this approximation technique degrades the statistical properties of the distribution generated for each RV, our observations show that in some cases, it can improve the application endpoint quality, which is ultimately the goal of the optimization mode.

Our goal in Section 4.2 is to detect and avoid updating the RVs that have concentrated PDFs and unchanged neighbors, and consequently speeding up the execution of the algorithm.

4.1.2 Stable Random Variables

As the execution of Gibbs sampling algorithm makes progress, RVs gradually converge to their final labels. This process is further accelerated in the optimization

mode. Therefore, it becomes unnecessary to update all RVs at every iteration because some of them simply cannot change their label. Figure 4.1 shows the number of times RVs change labels normalized to the total number of times they are updated in three input sets for two applications of stereo vision and motion estimation each. Based on the figure, only 22% – 46% of RV updates result in changing labels. Consequently, there is an opportunity for up to 54% – 78% speedup. However, not all of this speedup can be gained. If a RV simply *does not* change its label does not necessarily mean that it *cannot* do so. Some other conditions must be met for us to be able to skip updating RVs, which is explained in Section 4.2.

4.2 Event-Driven Gibbs Sampling

A RV can only change its label if i) it has more than one label with non-zero probability (i.e., non-concentrated PDF), or ii) at least one of the RVs on which it is conditionally dependent changes its label and thus, changing this RV’s PDF. In other words, if a previous computation resulted in a PDF concentrated on one label, which is likely due to the decreasing T in the optimization mode and the probability cut-off technique, the PDF is going to remain that way until something in its neighborhood changes. Therefore, we update a variable in two cases, i) if at least one of its neighbors changes, or ii) it did not have a concentrated PDF to begin with. We call this optimization event-driven Gibbs sampling (EDGS). This technique is similar to vertex programming in graph algorithms [56, 58], but we customize it for the context of MRF inference with Gibbs sampling.

We utilize two queues to keep track of RVs that must be updated in alternate rounds (i.e., a queue for black RVs and another queue for white RVs). Figure 4.2 compares EDGS and the plain vanilla update scheme that updates all RVs at every iteration. The shaded regions show the extra work that must be done in EDGS to read RVs from the queues and write new RVs to them. To simplify writing new

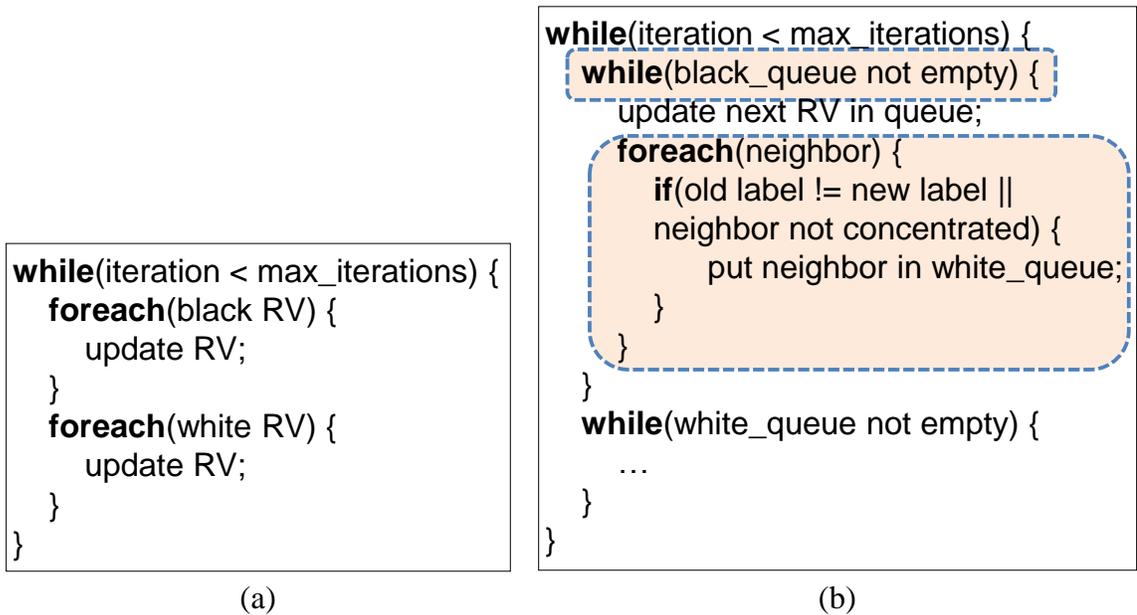
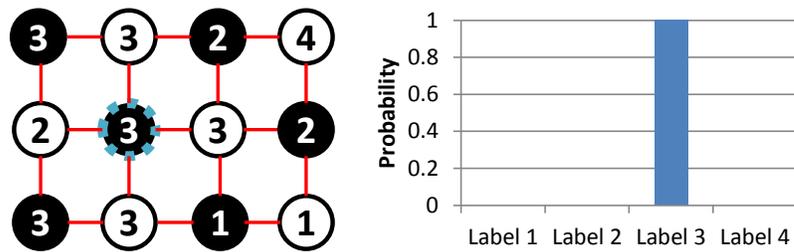


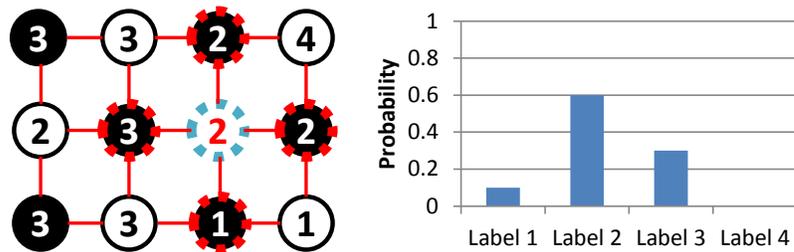
FIGURE 4.2: Comparison of update procedure when (a) all RVs are updated at each iteration, and (b) when EDGS is adopted. The shaded regions highlight the additional work that must be done in EDGS compared to the baseline. The process for white RVs in EDGS is omitted for the sake of brevity.

RVs to the queue, we check the necessary conditions when an RV’s neighbors are being updated. This works because the conditional independence in first-order MRF is mutual. Therefore, while updating a RV, we compare its new label with its old label, and if the two labels do not match, we put all neighbors in their corresponding queue. Additionally, we have to check if the neighbors have concentrated PDFs. To do so, we use an extra matrix whose entries correspond to RVs and are set only if the corresponding RV has a concentrated PDF.

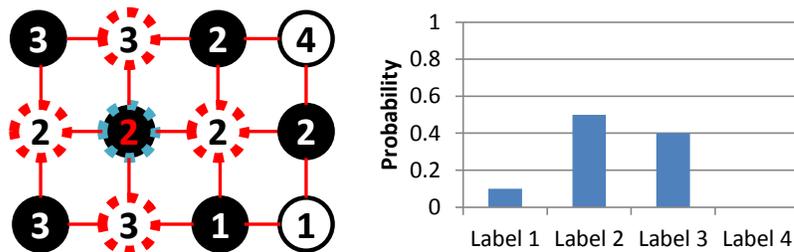
Figure 4.3 demonstrates an example scenario to better explain when EDGS updates RVs. Some intermediate steps and details in the example are omitted for the sake of brevity. In the figure, each part shows the state of the MRF after updating the RV shown in the blue dotted circle on the left, and its PDF on the right. RVs in red dotted circles are added to the appropriate queue for update after the



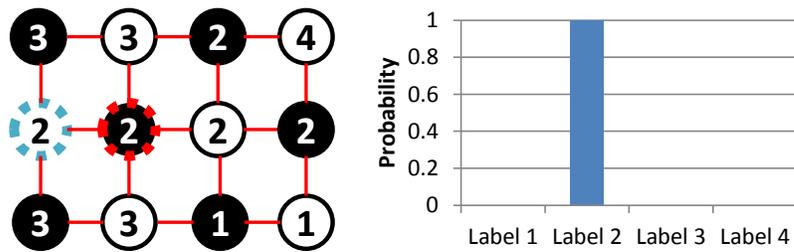
(a)



(b)



(c)



(d)

FIGURE 4.3: An example scenario of updating RVs in a first-order MRF using EDGS. Circles are RVs in the MRF, and the colors demonstrate conditional independence among them. Numbers inside RVs are their current labels. Graphs on the right are the PDF of the RV shown in blue dotted circle.

computation is complete. In Figure 4.3a, the black RV being updated has a PDF concentrated on label 3 and thus, it can only take on this label. In Figure 4.3b, the white RV changes its label from 3 to 2. Therefore, all its neighbors must be added to the black queue for update, because a neighbor of theirs has changed its label and conditional dependencies in MRF are bidirectional. Next, in Figure 4.3c, the black RV from step 4.3a is updated again despite having a concentrated PDF in the previous iteration. The reason is that its right neighbor had added it to the black queue for update due to changing its label. Moreover, since this RV has changed its own label, it adds all its neighbors to the white queue for update. This time around, the black RV does not have a concentrated PDF anymore. Thus, in Figure 4.3d, after the white RV is updated, although it does not change its label, it adds the right neighbor to the black queue for update because that neighbor does not have a concentrated PDF.

4.3 EDGS Implementation for GPUs

To evaluate the effectiveness of EDGS, we implemented it for execution on GPUs due to the massive parallelism provided by them. Our implementation mostly reflects the pseudo-code shown in Figure 4.2b, with two exceptions. The first one is the consideration of the cut-off threshold after updating the RV to determine whether it has a concentrated PDF. If it does, the corresponding entry in the concentrated matrix (see Section 4.2) is set.

The second difference is due to the single-instruction multiple-thread (SIMT) execution model of the GPU, which introduces some challenges for the implementation. To be more specific, because all threads in a warp execute more efficiently when they are in lockstep, skipping updates of stable RVs is better done at the granularity of warps instead of individual RVs. To address this issue, we break the MRF into regions at least as large as a warp and keep track of the conditions for updating RVs

at the granularity of these regions instead of individual RVs.

Although tracking RVs at a coarser granularity might decrease the opportunity to skip updating stable RVs (because a region must be updated even only one RV in it is not stable), an upside of this approach is lower pressure on the memory system. Since we use queues to store the indices of RVs to be updated, tracking regions instead of individual RVs means that much smaller queues are needed. The actual queue size depends on the dimensions of the regions, but it will be at least $32\times$ smaller because that is the warp size on the GPU and we want our regions to be at least as large as a warp. We perform design space exploration with different regions sizes in Section 3.5. In addition to the smaller queue size, there will be less contention for queue operations. This is important because adding regions to the queues must be done atomically for the Gibbs sampling to work correctly, and atomic operations at very fine granularity could impose a large overhead.

To ensure each RV region is added at most once to the queue, we utilize a 2D matrix where each entry corresponds to a region in the MRF. All entries in the matrix are initialized to zero. We use a separate matrix for each iteration to avoid the overhead of re-initializing the matrix during the execution. Before adding a region to the queue, we atomically exchange the value at the corresponding index in the 2D matrix with ‘1’. If the value read from the matrix is also ‘1’, we know that the region is already added to the queue and thus, we stop here. On the other hand, if the value read from the matrix is ‘0’, we atomically increment a queue index variable and put the region at the end of the appropriate queue using the index.

The process of updating regions and adding their neighbors to the alternate queue is continued until the current queue is empty (as shown in Figure 4.2b).

Table 4.1: Parameters for design space exploration.

Parameter	Value
Region Size	8×8 , 8×16 , 8×32 , 16×8 , 16×8 , 16×8 , 32×8 , 32×8 , 32×8
Thread Blocks (TB)/SM	1, 2, 4, 8, 16
Cut-off Threshold (only for EDGS)	$1/2$, $1/4$, $1/8$, $1/16$

4.4 Evaluation

4.4.1 Methodology

We use three image analysis applications to evaluate our design, namely, image segmentation [89], stereo vision [9, 79], and motion estimation [53]. Image segmentation divides a single image into multiple regions. Stereo vision reconstructs the depth information of objects in a field captured from two cameras by matching the pixels between the two images. Finally, in motion estimation, the goal is to determine the motion vectors of pixels between two consecutive frames of a video.

We implemented these applications in CUDA [66] for both the baseline and EDGS. Our baseline is a parallel version of Gibbs sampling that does not skip updating any RVs. We used cooperative groups to synchronize all thread blocks at the end of each iteration and therefore, avoid the overhead of numerous kernel launches to achieve synchronization. As a consequence, the size of the grid is limited to the capacity of the device for our kernel, i.e., all threads in the grid must be present on the GPU simultaneously, and thus, each thread block is responsible for updating multiple regions. We used 8-bit integers for labels (i.e., our implementation can support at most 256 labels, but this is not a hard constraint and can be easily modified to support more labels), 32-bit integers for energy values, and single-precision floating-point representation for probability and random numbers.

We performed design space exploration in terms of region size, region structure, number of active thread blocks per SM, and cut-off threshold for probability trun-

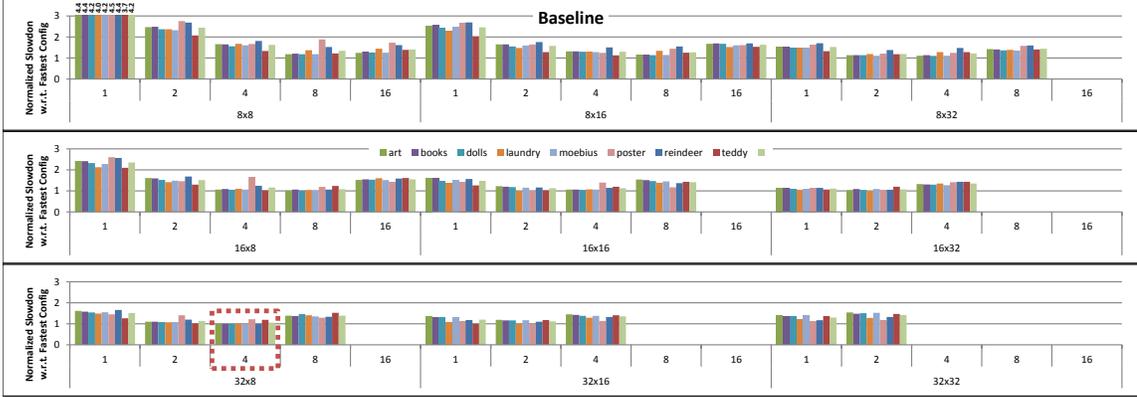


FIGURE 4.4: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for baseline implementation of stereo vision. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

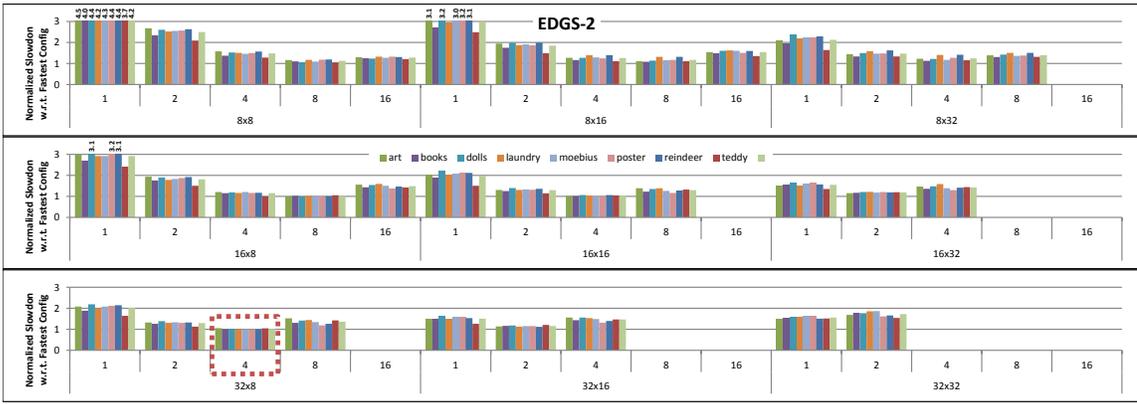


FIGURE 4.5: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-2 implementation of stereo vision. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

ca-tion. Table 4.1 summarizes the values used for design space exploration. Not all combinations of these parameters were feasible to run, therefore our graphs will have missing points wherever this was the case. We repeated the experiments with each set of parameters 10 times and we report average of the runs, with error bars where appropriate. We ran all experiments on Nvidia RTX 2080 Ti GPU and used the time for kernel execution to measure performance.

After performing the design space exploration, we select the best-performing configurations of both baseline and EDGS with each cut-off threshold, called EDGS-

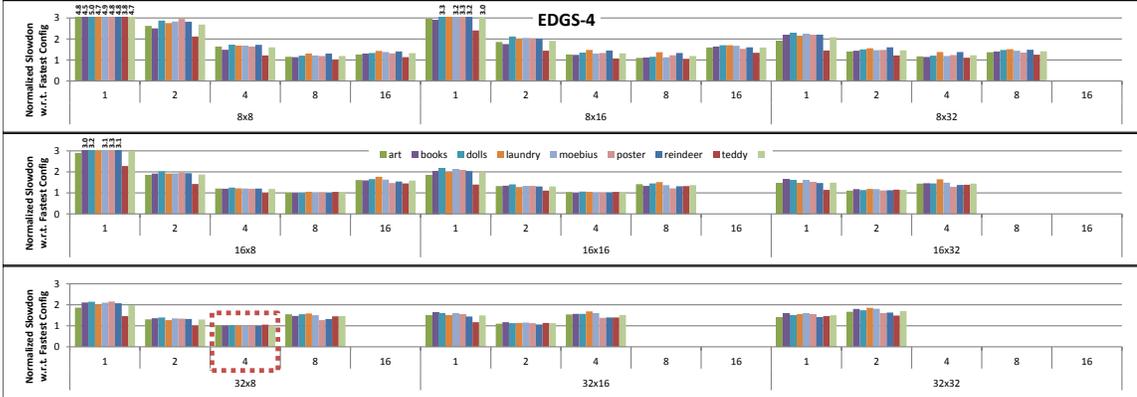


FIGURE 4.6: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-4 implementation of stereo vision. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

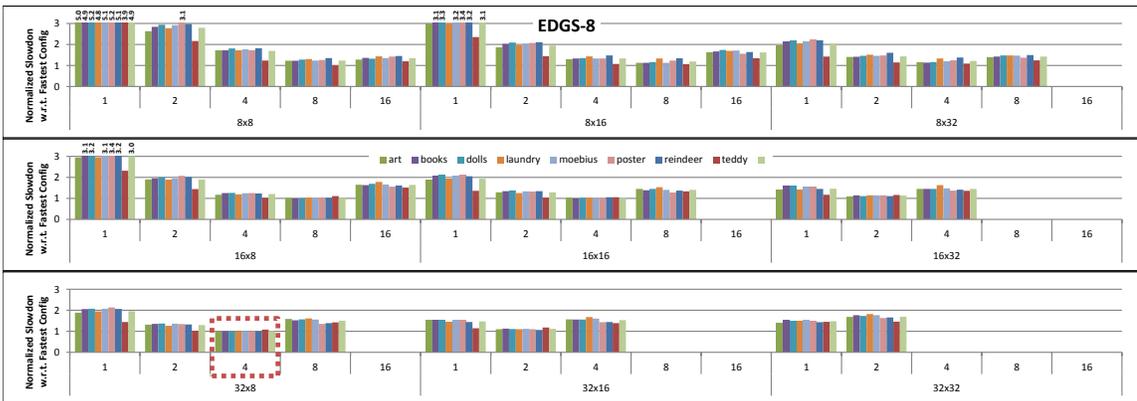


FIGURE 4.7: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-8 implementation of stereo vision. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

2, EDGS-4, EDGS-8, and EDGS-16 in the graphs, for comparison. The process for selecting the best configuration is the following. We measured the execution time of all benchmarks for each application using different configurations, and normalized those execution times to that of the fastest configuration for each benchmark to obtain the slowdown for all other configurations. Then, we averaged the slowdowns for all benchmarks for each configuration, and selected the configuration with the lowest average slowdown.

We evaluate image segmentation against 30 images randomly selected from Berke-

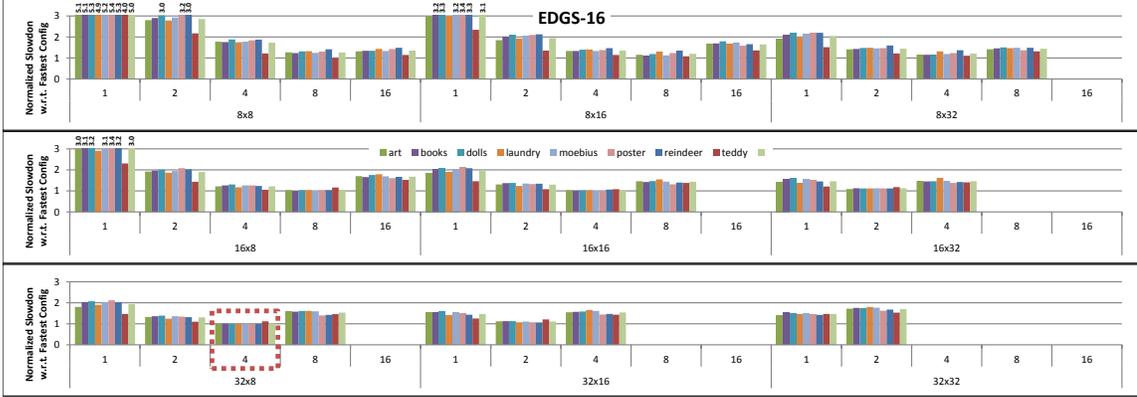


FIGURE 4.8: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-16 implementation of stereo vision. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

Table 4.2: Application parameters used in the evaluations.

Parameter	Image Segmentation	Motion Estimation	Stereo Vision
α	0.01	1	1
β	16	8, 2	2
initial T	10	80	10
Labels	2, 4, 8	49, 225	30-56
Size	321×481 , 481×321	380×420 , 388×584 , 480×640	370×447 , $370, \times 463$, 375×450 , 383×435
Iterations	100	1000	1000

ley Segmentation Database (BSD300) [59]. For stereo vision, we used eight input image sets from Middlebury [79], and for motion estimation, we utilized six input image sets from Middlebury [6]. Table 4.2 summarizes the parameters used for each application. Parameter names correspond to those in Figure 2.1. To compare the quality of the results, we compare both baseline and EDGS against a human-generated ground truth. We use variation of information (VoI) as the quality metric for image segmentation [104]. For stereo vision, we use bad-pixel (BP) percentage as the metric [79], and for motion estimation, we measure end-point error (EPE) as the metric for evaluating the results [53].

In addition to the execution time and result quality, we also report the percentage of skipped updates when using regions of RVs to quantify the effects of EDGS on the execution of the Gibbs sampling algorithm. Moreover, we report the maximum percentage of skipped updates possible by implementing EDGS at RV-level granularity. In this implementation, we do not use queues to keep track of update conditions, instead we rely only on the PDF concentration matrix. As a result, each region in the input is processed by the same thread block throughout all iterations.

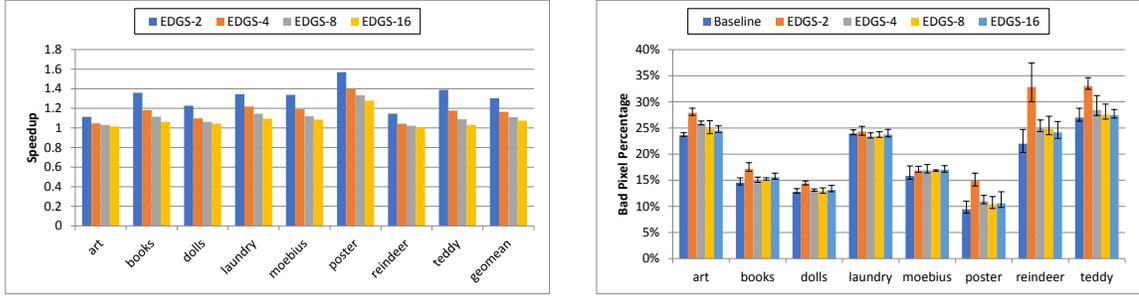
4.4.2 Results

Stereo Vision

Figures 4.4-4.8 show the results of our design space exploration for stereo vision. The trend in these graphs is that at each design point and region size, the performance tends to increase up to a point and then slightly decrease. The reason is that the extra parallelism provided by additional thread blocks per SM improves performance until the working set exceeds the cache size, at which point the cache misses more than offset the benefits of extra threads. In fact, even between region sizes at the same design point, those that have the same number of threads per SM tend to have similar performance.

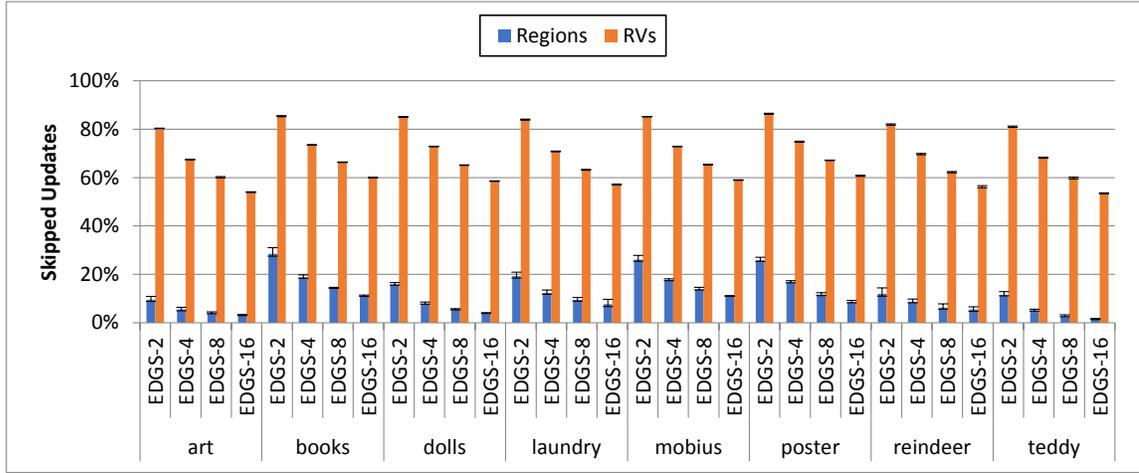
For all design points running stereo vision (both baseline and EDGS), the configuration with 32×8 region size and four thread blocks per SM outperformed other configurations, which is the configuration used in our comparisons. Due to the access pattern in stereo vision, configurations with wider regions (e.g., 32×8) perform better than those with taller regions (e.g., 8×32), since they can better leverage the existing spatial locality exhibited by the application.

Figure 4.9c compares the percentages of skipped updates for two granularities of region-level and RV-level. The figure illustrates the trend of declining skipped updates as the cut-off threshold decreases for both granularities, which has the effect



(a)

(b)



(c)

FIGURE 4.9: Speedup of EDGS (a), bad pixel percentage (b), and percentage of skipped updates of EDGS for region-level and RV-level granularity(c) for stereo vision.

of smaller speedups as the cut-off threshold shrinks. Furthermore, it shows the best case opportunity for skipping updates if we were to track the update conditions at RV-level instead of region-level. Although at a finer granularity the opportunity for skipping updates would grow by 52.5%-82.2%, the mismatch between the SIMT execution model and RV-level updates leads to 10.9%-43.5% slowdown. On the other hand, the highest speedup for region-level updates is 30.3%, which is achieved by EDGS-2. However, it comes with a high quality loss in some cases. EDGS-4 limits this loss to less than 3% and provides a 16.4% speedup.

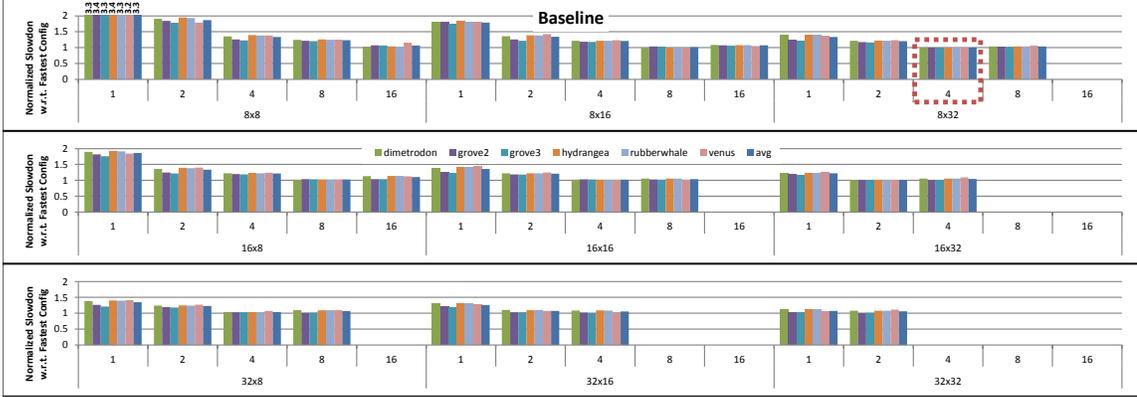


FIGURE 4.10: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for baseline implementation of motion estimation with a 15×15 window. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

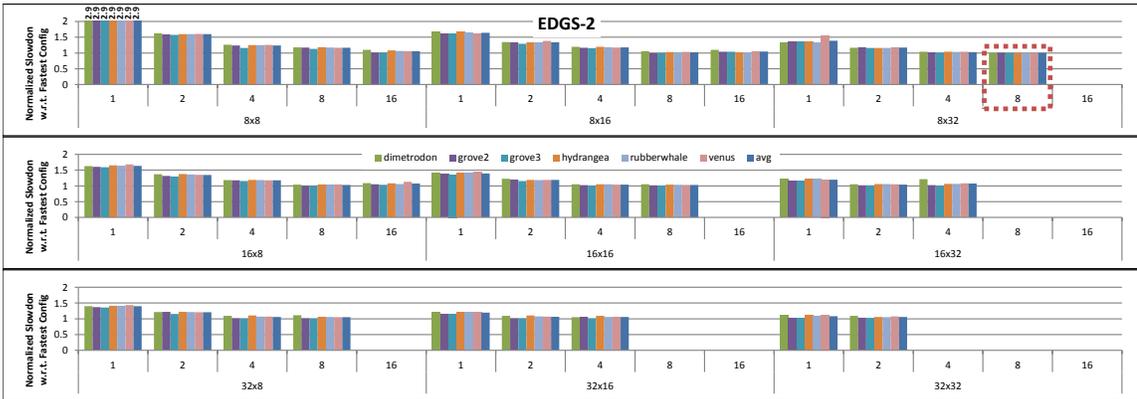


FIGURE 4.11: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-2 implementation of motion estimation with a 15×15 window. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

Motion Estimation

We performed the experiments with two window sizes of 15×15 and 7×7 to investigate the effects of the number of labels on the performance of EDGS. However, not all benchmarks produced acceptable results with a 7×7 window, i.e., they include motion vectors that exceed this window size. Therefore, we only present results for those that do produce output with good quality.

Figures 4.10-4.14 show the results of design space exploration for 15×15 . Take-

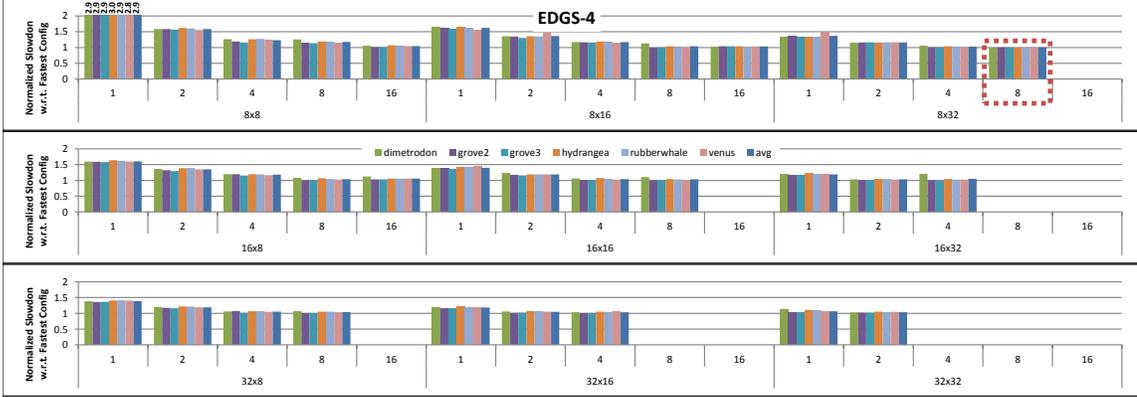


FIGURE 4.12: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-4 implementation of motion estimation with a 15×15 window. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

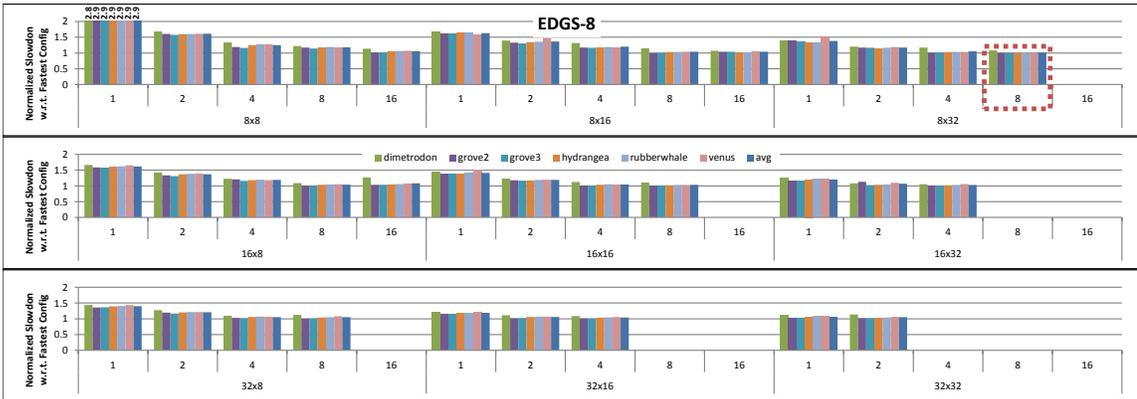


FIGURE 4.13: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-8 implementation of motion estimation with a 15×15 window. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.

always regarding configurations and performance are similar to stereo vision. One key difference is the shape of the best-performing regions. Table 4.3 lists the best performing configuration for each design point. We use these configurations in our comparisons in the rest of this section.

Due to the large number of labels with a 15×15 window, it is less likely that regions of RVs become stable. The reason is that it is more difficult for the same label to stand out in all RVs in a region. This can be observed in Figure 4.15c, where the percentage of skipped updates does not exceed 4.5% for region-granularity. As a

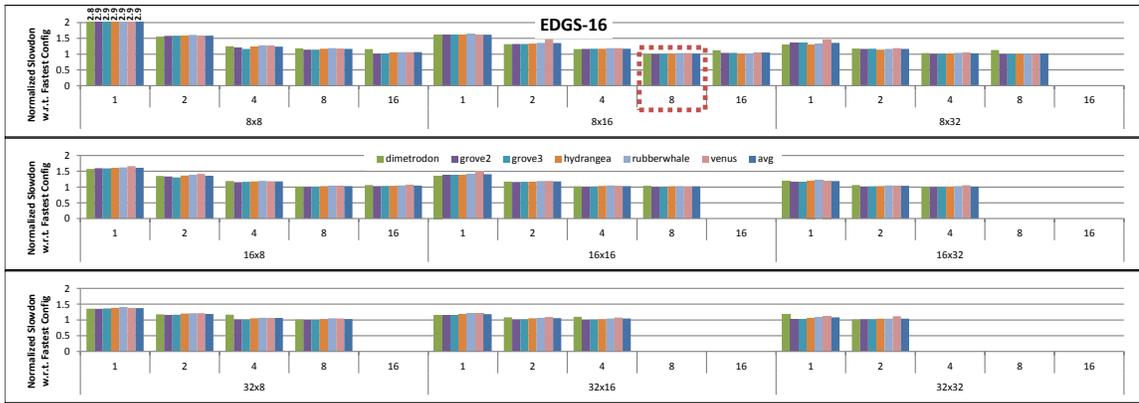
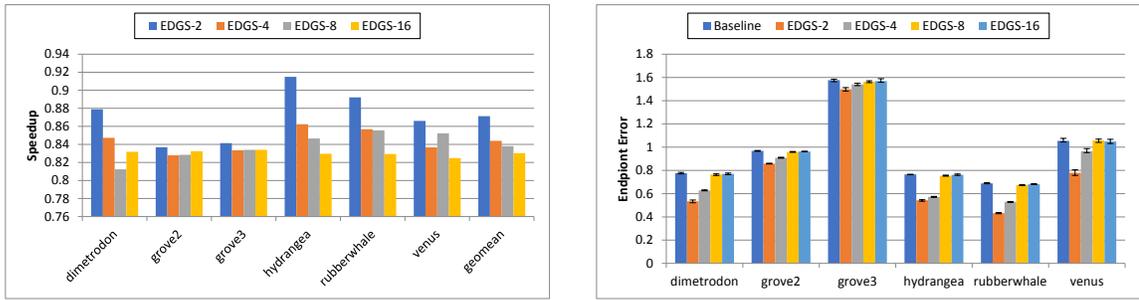
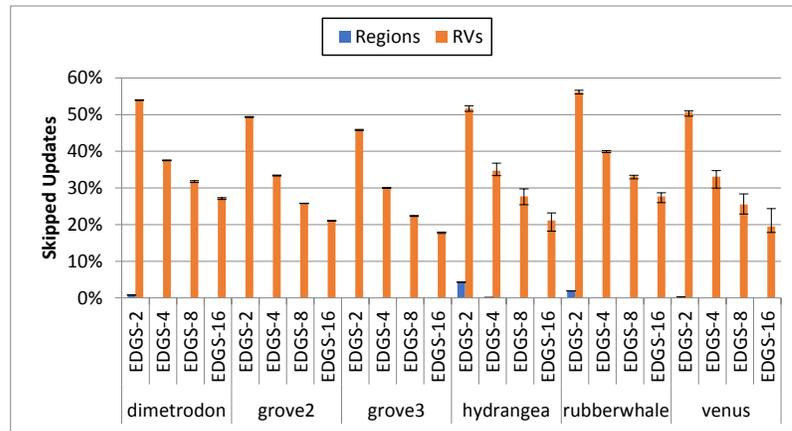


FIGURE 4.14: Normalized slowdown of all configurations in Table 4.1 with respect to the fastest configuration for EDGS-16 implementation of motion estimation with a 15×15 window. The fastest configuration is shown using the dotted red rectangle. In the x-axis, the bottom row shows the region size and the top row shows the number of thread blocks per SM.



(a)

(b)



(c)

FIGURE 4.15: Speedup of EDGS (a), endpoint error (b), and percentage of skipped updates of EDGS for region-level and RV-level granularity (c) for motion estimation with a 15×15 window.

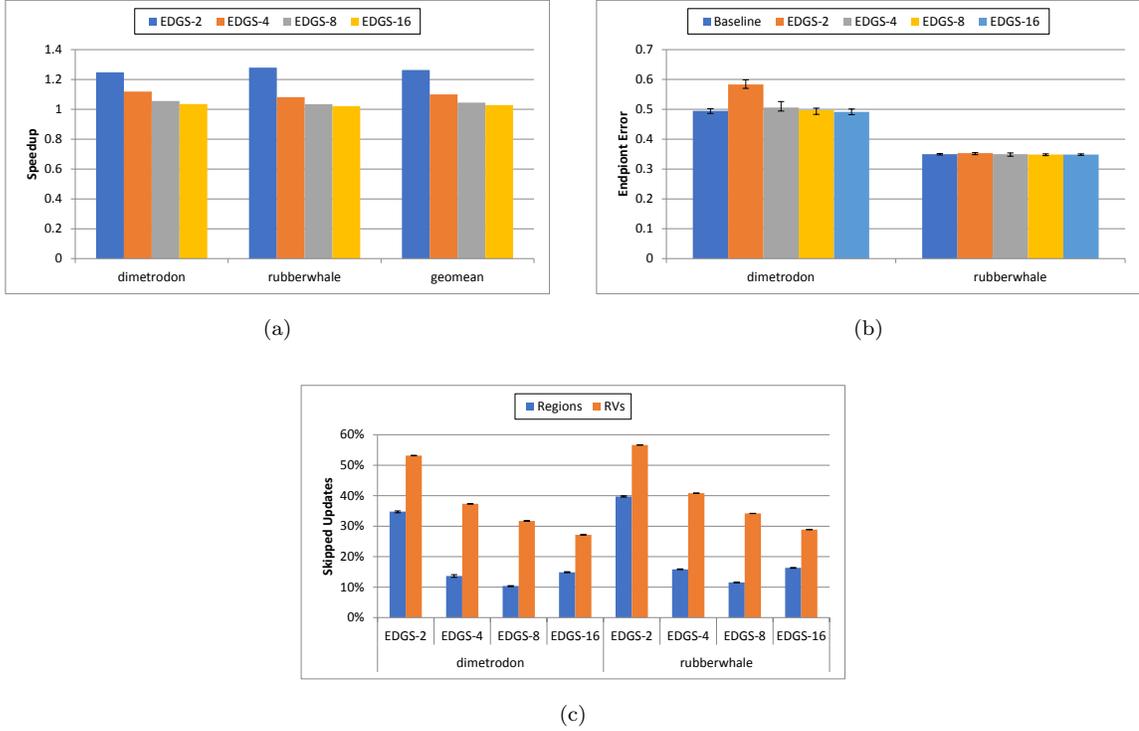


FIGURE 4.16: Speedup of EDGS (a), endpoint error (b), and percentage of skipped updates of EDGS for region-level and RV-level granularity (c) for motion estimation with a 7×7 window.

result, the overheads of EDGS offset any gain made by the small amount of skipped updates, which is evident in the negative speedup numbers in Figure 4.15a. Even though tracking update conditions at RV-level improves the skipped updates by 17.8%-55.3%, it does not result in improved performance either.

Despite the lower performance of EDGS, an interesting phenomenon is observed

Table 4.3: Best performing configurations for the baseline and different design points of EDGS for motion estimation.

Design Point	Motion Estimation 7×7	Motion Estimation 15×15
Baseline	$(32 \times 8, 2)$ ¹	$(8 \times 32, 4)$
EDGS-2	$(8 \times 8, 8)$	$(8 \times 32, 8)$
EDGS-4	$(8 \times 32, 4)$	$(8 \times 32, 8)$
EDGS-8	$(8 \times 32, 4)$	$(8 \times 32, 8)$
EDGS-16	$(8 \times 8, 8)$	$(8 \times 16, 8)$

¹Region size of 32×8 and 2 TB/SM.

in terms of quality of the output. Figure 4.15b shows that the more aggressive the probability cut-off approximation is, the smaller the EPE becomes. The reason is that with such high number of labels, the algorithm has too many bad choices. When we restrict those bad choices by rounding down the low probabilities to zero, the quality increases. One can think of this as a better annealing schedule for this application.

With a 7×7 window, the results are more similar to stereo vision. Figures 4.16a and 4.16c show that by reducing the cut-off threshold, the amount of skipped updates decreases, which results in lower speedup and better quality of the results. The difference between tracking at region-level and RV-level is also less dramatic than the 15×15 window scenario. The exception to the trend of decreasing skipped updates is between EDGS-8 and EDGS-16. The reason is the larger region size in EDGS-4 and EDGS8 compared to that of EDGS-16. In general, a smaller region size allows for skipping more updates, but this is not the only factor affecting the performance.

Overall, EDGS-4 provides comparable quality to the baseline while gaining a 10% speedup. However, if a higher loss in quality is acceptable to the application, EDGS-2 provides a speedup of 26.3%.

Image Segmentation

Image segmentation is different from the other two applications in some aspects. First, it operates on only one image instead of two, which means less pressure on the cache for the baseline. Second, it converges much more quickly, which means the temperature parameter T is still high. The implication of this is the higher likelihood of a few RVs in a region to be unstable, even though the rest of the region might be stable. This is a shortcoming of our approach for tracking RVs over fixed regions. Finally, the neighborhood energy function in image segmentation is stricter than stereo vision and motion estimation, i.e., in those two applications, the

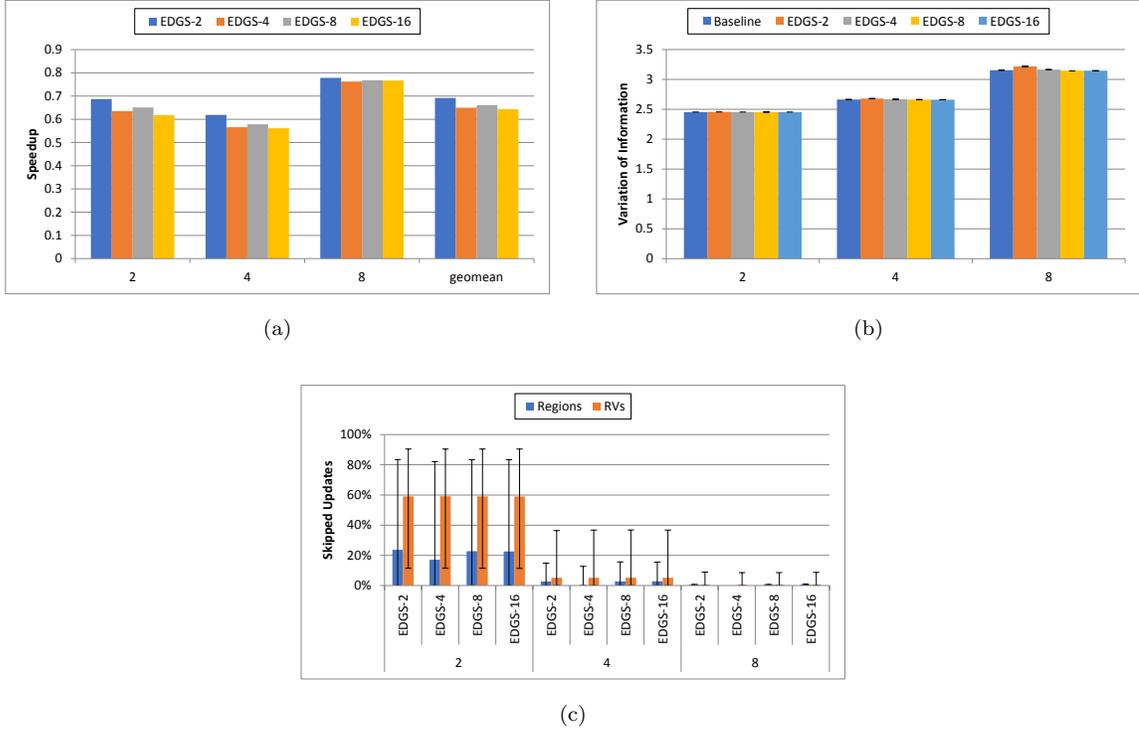


FIGURE 4.17: Speedup of EDGS (a), variation of information (b), percentage of skipped updates of EDGS for region-level and RV-level granularity (c) for image segmentation. The x-axis shows the number of labels used in segmentation.

neighborhood energy function calculates the absolute and square of the difference with neighboring RVs' labels, respectively, whereas in image segmentation it is a delta function. The higher β to α ratio of image segmentation compared to the other two applications, in addition to the stricter neighborhood energy function only exacerbates the possibility of having a few unstable RVs in a mostly stable region.

The combination of the reasons mentioned above results in the benefits of our approach being eliminated by the overheads it introduces. Figure 4.17a shows the average speedup of EDGS among all 30 images for two, four, and eight labels. Overall, there is 22%-44% overhead, although it is different for each image. In fact, a few of the images gain a speedup of up to 41%, but for the majority the overheads suppress the benefits. The high amount of variation in skipped updates in Figure 4.17c supports this observation. In terms of the quality of the results, however, Figure

4.17b demonstrates that EDGS performs very similar to the baseline.

Another observation that further confirms the difference in behavior between image segmentation and the other two applications is that, unlike stereo vision and motion estimation, tracking update conditions at RV-level in image segmentation produces better performance compared to region-level, albeit still not better than the baseline. For 2-label image segmentation, the overhead is less than 2.5%. For 4-label and 8-label cases, it is, respectively, 10% and 4.9%. Furthermore, more individual images obtain speedups compared to the region-level case. The main reasons contributing to this contrast are avoiding processing of the same region by different thread blocks at different iterations, i.e., exploiting locality.

4.4.3 Takeaways

The following are the key takeaways from our experiments:

- Increasing the number of threads improves the performance until the point where thrashing starts to happen in the cache.
- Tracking update conditions at region-level does not exploit the full opportunity for skipping updates, but it is a good fit for the SIMT execution model of the GPU.
- The amount of skipped updates is higher when the region size is smaller, but this is not the only factor that affects the performance. The number of thread blocks per SM and cache hit rate are also important contributing factors.
- In addition to the region size, among the factors that impact the amount of skipped updates are the number of labels, number of iterations, application behavior, i.e., α , β , and annealing schedule, as well as the energy function.
- The best region structure depends on the application memory access pattern.

- When the application has a lower memory footprint, e.g., in image segmentation, only one image is considered as opposed to in motion estimation and stereo vision which process two images, the improvements of EDGS reduce because the baseline can take better advantage of caches.
- Although approximation leads to degrading the statistical properties of the algorithm, it might improve the application end result quality under certain circumstances, e.g., when the number of labels is large.

To summarize, EDGS performs best when there is the right balance between the number of threads, cache hit rate, and skipped updates. Optimizing for each of these factors alone is not enough to obtain the optimal performance. The balance among these factors is influenced by application characteristics, such as algorithm parameters and memory footprint. Additionally, the SIMT execution model of the GPU does not allow for exploiting the full opportunity for skipping stable RVs. Another framework that is a better match for the irregular access pattern arising from the convergence of some RVs, e.g., a specialized hardware accelerator, will potentially offer higher speedups.

4.5 Related Work

There have been works that implement Gibbs sampling for execution on GPUs. Suchard et al. [85] present a CUDA implementation for Gibbs sampling. Terenin et al. [91] break down the Gibbs sampling operations into multiple stages and use libraries and custom kernels to run them on the GPU. There are also more application-specific implementations [4, 19, 102]. Although we do not target the general Gibbs sampling problem on the GPU, our focus is on a wide class of problems represented by first-order MRF models. We exploit the model’s structure, as well as an approximation technique to accelerate the Gibbs sampling computations. Moreover, the

detection of stability in RVs is orthogonal to other techniques used for accelerating Gibbs sampling.

Vertex programming is a popular programming interface for graph processing [56, 58, 86]. In this model, the graph algorithm is represented by operations on a single vertex and its edges, and each vertex processes incoming edges from other active vertices, i.e., those who have been updated in the last iteration. This is similar to MRF inference in our work, which is a special case of vertex programming for a general graph. However, the addition of the approximation technique and tracking concentrated PDFs is what actually makes this connection more meaningful because it creates the notion of active vertices. Without it, all vertices are always active.

Using approximations in MCMC and machine learning accelerators in general is commonplace, because they allow for simplifications in hardware by trading off accuracy [50, 51, 83, 106, 108]. Zhang et al. [106] study the effects of precision at different places in an MCMC accelerator from an empirical perspective. There are also works that study the effects of approximation in MCMC and Monte Carlo simulations in general [8, 17, 76, 81]. A more detailed survey is presented elsewhere [78]. However, to the best of our knowledge, the effects of the particular approximation technique we used have not been studied. Our empirical results show acceptable effects on the result quality, and in some cases even quality improvement, but a deeper theoretical analysis of the effects on the statistical behavior of the MCMC is also important.

4.6 Summary

A Markov Random Field (MRF) is a powerful graphical model for representing numerous applications. It encodes the conditional dependence among random variables (RVs). Probabilistic algorithms such as Gibbs sampling [29] can be used to solve problems represented by MRF. Gibbs sampling is an iterative method which goes through all RVs in the MRF and updates them until converged to the final result.

The update process needs sampling from probability distributions, which is computationally intensive. Therefore, it is desirable to avoid unnecessary RV updates if possible. To this end, in this paper we propose event-driven Gibbs sampling (EDGS) and implement EDGS for GPUs. In this scheme, we divide the first-order MRF into smaller regions and track these regions to see if they are stable and thus, can be skipped for updates. Our evaluations using two image analysis applications show that speedups of 26.3% and 30.3% can be gained, although with some loss in result quality. This loss can be bounded to smaller values by trading off speedup. Nevertheless, our experiments demonstrate that for image segmentation, which operates on a smaller amount of data and converges more quickly compared to the other applications, the overheads of our approach outweigh the benefits. Our observations also show that although using approximation techniques degrades the statistical properties of the algorithm, it can improve the application end result quality when the application has a large number of labels. In this case, using approximation limits the bad choices in the algorithm by rounding very small label probabilities to zero, which performs similar to a better annealing schedule for this case.

Adaptive Simultaneous Multi-tenancy for GPUs

Graphics Processing Units (GPUs) are massively parallel accelerators that were originally intended to execute graphics applications, but their high throughput and energy efficiency motivates their use in broader application domains. In the previous chapter, we utilized GPUs to accelerate a certain class of applications, i.e., first-order MRF inference using Gibbs sampling. In this chapter, however, we broaden our focus to improve the resource utilization of GPUs for more general applications in multi-tenant settings such as cloud environment.

Numerous cloud service providers offer GPUs as part of their solutions [3, 32, 62]. In such environments, a large number of kernels with different memory access and compute behaviors request running on GPUs. Running only one kernel on the GPU in these environments underutilizes resources, since a single kernel cannot utilize all resources on the device most of the time [72]. Therefore, always dedicating the entire GPU to only a single kernel is not cost-efficient either for the service provider or for the customer. One example to address this issue is Amazon Web Services' elastic GPUs for applications that have high compute, storage, or memory needs that still

could benefit from additional GPU resources[3]. Another example is the capability of NVIDIA GPUs that started with the Volta architecture to support statically dividing the GPU into multiple smaller virtual GPUs [70].

Sequential execution of kernels on GPUs in multi-tenant environments, such as datacenters, leads to long wait times and reduces system throughput. Overcoming this limitation requires a method for sharing the device among multiple users that is efficient and adaptive to the events in the system, i.e., the arrival and departure of kernels. NVIDIA GPUs support simultaneous execution of multiple kernels and memory operations in a single application via Hyper-Q technology [67]. In addition, the CUDA Multi-Process Service (MPS) [69] facilitates concurrent execution of kernels and memory operations from different applications. However, the first-come-first-served (FCFS) and left-over resource allocation policies make concurrent execution on existing GPUs inefficient. The reason is that the FCFS policy creates a head-of-line blocking situation where the running kernel blocks other kernels until it has all its thread blocks mapped to Streaming Multi-processors (SMs). Additionally, simply allocating the left-over resources of the running kernel to the waiting kernels might not be the optimal solution, since such a policy ignores the different requirements of the kernels and only depends on the order in which the kernels arrive at the GPU. The Volta and newer architectures try to overcome this head-of-line blocking by adding the capability to statically divide the GPU into smaller virtual GPUs, but the above problems apply to each virtual GPU too.

An effective and low-overhead scheme for sharing the GPU among multiple kernels should address both the resource underutilization and the adaptiveness issues. This requires overcoming the head-of-line blocking problem in thread block scheduling on the GPU to address the adaptiveness problem, and having a simple yet effective policy for resource allocation to tackle the underutilization issue. Previous work attempted to support multi-tenancy on the GPU either by a software-based

approach [15, 101] or by adding the necessary hardware support [73, 90]. These works solely support preemption to make the system responsive, i.e., to force low-priority kernels to yield control of the GPU to high-priority kernels, and hence, do not alleviate the resource underutilization problem. A different class of work addresses multi-tasking on the GPU by modifying the hardware [1, 44, 55, 74, 97, 103] or artificially fusing the kernels from different applications together [34, 55, 72, 100]. In the hardware-based work, the resource allocation policy is fixed and cannot be changed. Furthermore, most of the necessary hardware support is not present in existing GPUs. Software-based approaches that rely on merging applications together are impractical in real world scenarios since it requires merging every possible combination of kernels beforehand. Our work does not suffer from these shortcomings since we use a low-overhead software approach to solve the GPU multi-tenancy problem at run-time.

These challenges inspired us to design a system that realizes multi-tenancy for commodity GPUs. In this chapter, we propose adaptive simultaneous multi-tenancy for GPUs. Our system dynamically adjusts the resources allocated to kernels based on the requirements of all kernels requesting execution on the GPU at run-time. We achieve this by adopting a cooperative approach between applications and a host-side service, supported via minimal application modifications and a lightweight API. Our approach focuses on a single server, as the problem of assigning work to specific servers in datacenters is addressed elsewhere [77]. Therefore, we assume that the work assigned to this machine is optimized by the higher level scheduler. This means that we execute all kernels assigned to the server simultaneously, and do not aim to select an optimized subset of kernels that may result in better performance since this problem is shown to be NP-complete [43].

In our proposed system, we manage the resources allocated to each kernel and control the mapping of kernels' thread blocks to SMs. Naïvely applying resource

allocation policies can lead to unintended mappings of thread blocks to SMs and result in further underutilization of resources. To avoid this, we build on the concept of persistent threads [36] with a few modifications to implement our desired mapping policy on the GPU. Our work differs from previous work that uses persistent threads to support preemption [15, 101] in that we show how to control the assignment of thread blocks to SMs and use it to have control over resource allocation to kernels. Moreover, support for preemption comes almost for free when we adopt this approach.

To realize adaptive simultaneous multi-tenancy, we implement a host-side service with which applications communicate to obtain launch parameters for their kernels. The service monitors the kernels running on the GPU and makes decisions for launch parameters based on the adopted allocation policy. In this work, we use offline profiling of kernels and implement a greedy policy using this data with the goal of minimizing the maximum execution time among all running kernels, or in other words, maximizing system throughput (STP). We show that using our design, STP is improved by an average of 9.8% (and up to 22.4%) for combinations of kernels that include at least one low-utilization kernel, with respect to the sequential execution of the kernels. Compared to a system in which persistent threads transformation is applied to the kernels, the average STP improvement for these kernel combinations is 4.3%. We do not compare our system with other software-based multi-tenant systems [15, 101], since the target of those systems is to improve the turnaround time of high-priority kernels whereas our goal in this work is to improve the throughput of the whole system. Improving STP, assuming Service Level Agreements (SLAs) are not violated, translates into less energy consumption of the datacenter by allowing for reduction in the number of servers for the same amount of work, or in higher scalability by doing more work with the same number of servers, both of which are crucial factors in determining the Total Cost of Ownership (TCO) [13].

In summary, we make the following contributions in this work:

- We identify the need for sharing the GPU among multiple kernels by characterizing the behavior of a set of benchmark kernels. Our observations show that running only one kernel at a time leads to underutilization of different types of resources on the GPU. On the other hand, simply running two kernels together without considering resource utilization does not realize the potential STP gains.
- We use the concept of persistent threads to control the resources allocated to each kernel at run-time. This allows us to solve the head-of-line blocking in the GPU block scheduler.
- We design and implement an adaptive simultaneous multi-tenant prototype system that runs on current GPUs. Adaptive simultaneous multi-tenancy is a generalization of single-kernel multi-tenancy [15, 101], and static simultaneous multi-tenancy supported by the NVIDIA Volta architecture and newer architectures, in which the GPU is divided between multiple kernels at the same time. Our system is composed of a host-side service that makes decisions regarding the allocation of resources to kernels and preemption/relaunch of running kernels, and an application-side API that encapsulates the communications with the service in a few function calls.
- We evaluate the proposed system against a set of benchmark kernels using a full prototype on real GPUs and show the effectiveness of our approach in terms of improving STP.

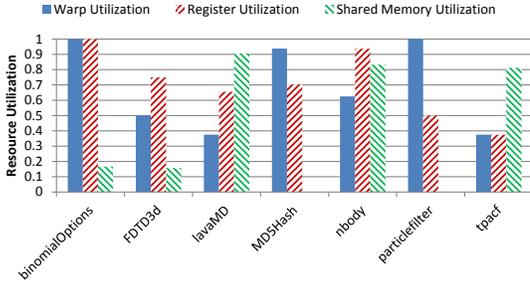


FIGURE 5.1: Spatial utilization of different resource types in SMs for the benchmark kernels.

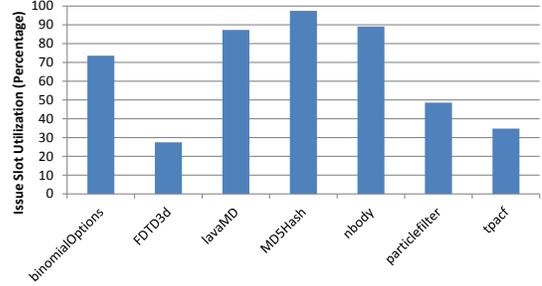


FIGURE 5.2: Issue slot utilization for the benchmark kernels.

5.1 Motivation

5.1.1 Resource Requirements

Figure 5.1 shows the amount of SM resources occupied by the benchmark kernels. The data are obtained from the system described in Section 5.3.1 using NVIDIA profiler and the details of the benchmarks are discussed in Section 5.3.2. This figure does not show how often each of these resources are used, but demonstrates how much of each type is occupied by kernels when run in isolation. To distinguish the utilization of resources over time, we refer to this metric as spatial resource utilization. There is a limiting resource for every kernel, i.e., the kernels exhaust one or two types of resources while there are more of the other types left unused. This creates opportunities to simultaneously accommodate more kernels with complementary requirements to maximize the throughput of the system. For instance, MD5Hash kernel needs more than 70% of the registers on the device, but uses no shared memory. It can be combined with lavaMD kernel which needs more than 90% of the shared memory to improve the spatial resource utilization of the GPU. *Taking advantage of this opportunity requires a method that shares each SM among multiple kernels, because sharing the GPU among multiple kernels while each SM is dedicated to a single kernel does not alleviate the SM resource underutilization.*

5.1.2 Issue Slot Utilization

A different metric for utilization is Issue Slot Utilization (ISU). ISU refers to the percentage of issue slots that issued at least one instruction. It is an indication of how busy the kernel keeps the device. Figure 5.2 shows ISU for the benchmark kernels. The contrast between ISU and spatial resource utilization is visible in Figures 5.1 and 5.2. The former suggests that MD5Hash and lavaMD are good candidates to be combined for throughput improvement, whereas the latter shows that both kernels keep the device busy more than 70% of the time. Thus, although the resource requirements of the two kernels are complementary, there are not many stall cycles during the execution of each of them that the other kernel can take advantage of. Based on the ISU values, lavaMD and tpacf are better candidates to run together, because despite their similar resource requirements, they have complementary ISUs. On the other hand, without complementary resource requirements, it is impossible to fit both kernels on the GPU. *Therefore, an efficient solution is needed to tune the resources allocated to each kernel such that the requirements for both metrics are met.*

5.1.3 Non-overlapping Execution

CUDA MPS [69] combines multiple CUDA contexts into one to allow for simultaneous execution of multiple kernels from different applications on the GPU. However, our observations show that the block scheduling algorithm on the GPU does not properly take advantage of this capability. This issue is covered in prior work too [15]. When multiple applications want to launch kernels on the GPU, their thread blocks are queued in the order they have arrived at the device. As the resources become available by completion of older thread blocks, newer ones are assigned to SMs. This FCFS policy leads to a head-of-line blocking situation where more resource-consuming kernels that arrived earlier block the execution of less resource-

consuming kernels, even though there might be enough resources to accommodate thread blocks of the smaller kernels. To address this issue, we use persistent threads [36] to restrict the number of CTAs of each kernel on the device, thus constraining the resources it uses.

To summarize, the challenges to sharing a GPU among multiple kernels are i) managing the resources allocated to each kernel such that the GPU can accommodate all kernels at the same time, ii) allocating resources to kernels to create complementary utilizations, iii) addressing the head-of-line blocking at the GPU block scheduler caused by the FCFS policy on the GPU, and iv) doing all of these at run-time in an adaptive fashion.

5.2 Adaptive Simultaneous Multi-tenancy

Our proposed solution to the challenges mentioned in Section 5.1 is adaptive simultaneous multi-tenancy. This concept is a generalization of single-kernel multi-tenancy proposed in previous works [15, 101], and static simultaneous multi-tenancy first supported by the NVIDIA Volta architecture [70]. The idea is to adaptively tune the resources allocated to each kernel to accommodate more kernels on the GPU while supporting kernel preemption to enhance the throughput of the multi-tenant system. To this end, we propose kernel code transformations and an application API to add flexibility to kernels, and employ a host-side service that monitors the kernels running on the GPU to make decisions regarding resource allocation which are then communicated to applications. In the rest of this section, we explain the details of our design.

5.2.1 Overview

Our proposed system is composed of a host-side service that manages the resources allocated to each kernel and determines when kernel adaptation (i.e., preemption

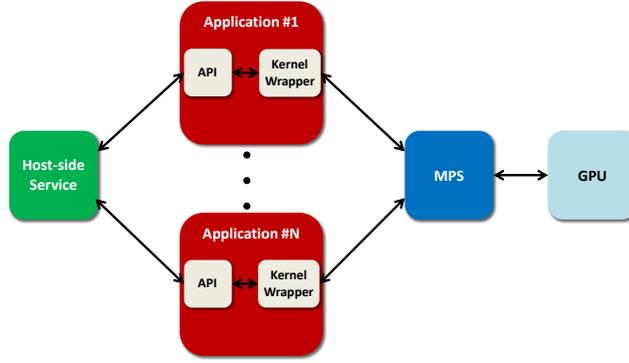


FIGURE 5.3: Overview of the proposed adaptive simultaneous multi-tenant system.

for reducing the grid size and relaunch for increasing the grid size) needs to occur, and an API for programmers to utilize the service. Figure 5.3 shows the overview of the adaptive simultaneous multi-tenant system. On the arrival of a new kernel or the departure of a running one, the service takes the following actions: i) it asks the applications for the number of thread blocks their kernels have executed, used in estimation of the remaining execution time. The remaining execution times are used in combination with profiling data in the allocation policy to maximize STP (addressing the challenge in Section 5.1.2); ii) it computes new parameters for the kernels that are going to run on the GPU. The parameters, which in this work is the number of CTAs but could be extended to different types of resources such as the number of registers or the amount of shared memory, must not cause the resources required by the kernels to exceed the available resources on the GPU (addressing the challenges in Sections 5.1.1 and 5.1.3); iii) it communicates the new parameters to the applications.

On the application side, the kernel launch is wrapped inside a function that communicates with the host-side service. We provide an API for the common actions that need to be taken when an application wants to launch a kernel. Ideally, these function calls would be included in the CUDA libraries so that the programmer does not have to add anything to their code, but for now they need to be included in the

application code manually or via a source-to-source transformation.

In short, the applications that want to run kernels on the GPU have to contact the host-side service using the provided API. The service controls when and which applications should preempt/relaunch kernels and the kernel launch parameters. The kernel launch parameters are passed via the API to the kernel wrapper function that launches the kernel. MPS intercepts kernel calls and merges them in a single context to run on the GPU concurrently. The fact that all concurrent kernels share a virtual address space creates security concerns in a multi-tenant environment. This issue is addressed in the Volta architecture by supporting the separation of virtual address spaces for kernels that run on different SMs. Since we share SMs among multiple kernels, this capability does not eliminate the security limitation of our work. However, adding a software address translation layer [80] can isolate the address spaces of different applications with minimal overhead when required.

5.2.2 Host-side Service

Applications communicate to the service via the API (Section 5.2.3) at two occasions: i) launching a kernel, and ii) starting the execution of the last thread block. The reason that we notify the service at this point, and not once the kernel is finished, is that after the last thread block begins execution, no changes can be made to the number of CTAs of the kernel. Therefore, by sending the notification before the kernel finishes, we can overlap the communications with and the parameter computation at the host-side service with the execution of the last round of thread blocks of the kernel, effectively hiding the latency of these operations without affecting the number of kernel's CTAs.

The service receives messages on a shared message queue in a loop. Whenever it sees a message in this queue, it keeps reading until the queue is empty to aggregate the effects of back-to-back messages from different applications on the system

in a single step. After reading all messages in the queue, the service opens two dedicated message queues for communication with each client application that has a kernel to run. These queues are used for sending preemption commands and launch parameters, and receiving the progress of the kernel.

When the service is notified by an application of a new event, i.e., a new kernel is arriving or an existing kernel begins the execution of its last thread block, it queries other applications for kernel progress via dedicated message queues. Previous work [101] used elapsed time for this purpose, and thus, there was no need to query the application. Nevertheless, this metric is not suitable for our purpose. Elapsed time can be used to measure the progress when only one kernel runs on the GPU at a time, whereas in our proposed system multiple kernels share the device simultaneously and therefore, do not make progress with the same rate as they do when they run in isolation. To overcome this issue, we use the number of executed thread blocks as an indicator of kernel progress.

The service then waits for the response from all applications, as those data are necessary for making allocation decisions. We use asynchronous memory copy operations to overlap these queries with kernel execution. Having the number of executed thread blocks and kernels profiling data, we then estimate the remaining execution time of the kernels (Section 5.2.6). Once this operation is done, the parameters for each kernel are sent to the corresponding application via the dedicated message queues and the application makes the appropriate adjustments.

5.2.3 Application Side

On the application side, we initialize the shared and dedicated message queues, obtain launch parameters for the kernel, wait for notifications from the service for preemption and new launches, and release the resources on completion of the kernel. Table 5.1 summarizes the application API to support these actions.

Table 5.1: Application API.

Function	Description
<code>init(kernel_args)</code>	Initializes the necessary variables for communication with the service and launching new instances of the kernel.
<code>obtainParameters(kernel_name, total_blocks, block_dim)</code>	Contacts the service with kernel's information and obtains the number of CTAs to launch the kernel. Furthermore, upon receiving the response, it creates threads for listening to the service and monitoring kernel progress.
<code>release()</code>	Releases the allocated resources.

init(): On a kernel launch, the application host code initializes the necessary variables. These include shared and dedicated message queues, necessary memory allocations for communication between the host and the GPU, and streams for asynchronous memory operations and kernel launches. The dedicated message queues are created based on the process ID of the application to ensure uniqueness. There are also pointers to kernel input arguments (`kernel_args`) that are used when launching a new instance of the kernel.

obtainParameters(): Once the initialization is complete, the host code obtains parameters for the kernel it wants to launch from the host-side service. To this end, it sends a message composed of the kernel name (`kernel_name`, to retrieve its corresponding profiling data at the host-side service), the names of the dedicated message queues (created using process ID, to open connections to the queues at the service), the total number of thread blocks the kernel wants to run (`total_blocks`, to be used for remaining execution time estimation), dimensions of a thread block (`block_dim`, to be used for resource usage calculation), and indication that this message is a request for a new kernel (as opposed to notification for the beginning of the execution of the last thread block of an existing kernel). After the message is sent, the host code waits to receive a response from the service. Once the response arrives, the kernel is

launched and two threads are created: one for listening to the host-side service for new launch parameters, and the other for monitoring the progress of the kernel. The first thread uses the stream for memory operations to asynchronously read the number of executed thread blocks from the device and to write to the memory location holding the preemption variable (`max_blocks` in Figure 5.4).

Once a new message with launch parameters comes from the service, there are three possible scenarios: i) the new number of CTAs is less than what the kernel is currently running with, ii) the new and old numbers of CTAs are equal (i.e., no actions required), or iii) the new number is greater than the old number of CTAs. In the first case, the thread preempts the proper number of CTAs to match the new parameter by writing to the preemption variable (described in the following section). In the last case, the thread launches a new instance of the kernel on a new stream to run in parallel with the current instance. The second thread is responsible for sending notification to the service once the last thread block of the kernel has started execution.

release(): Finally, when the kernel finishes, the host code deallocates all resources used for these communications.

5.2.4 *Kernel Code Transformation*

To have control over the number of CTAs and consequently, the resources allocated to the kernel, we use persistent threads [36]. The concept of persistent threads refers to limiting the number of threads to a value that the GPU can run simultaneously. In addition to control over resources, using persistent threads provides support for preemption at thread-level granularity. Preempting kernels only at thread completion mitigates the need for handling any remaining work due to preemption.

Using a persistent thread transformation, we override the `blockIdx` variable in CUDA, which refers to the logical thread block index. Figure 5.4 shows the required

```

1  __global__ void TransformedKernel(/*Original Arguments*/,
2                                     int grid_size,
3                                     int *block_index, int *max_blocks,
4                                     volatile int *concurrent_blocks) {
5      int smid = get_smid();
6      __shared__ int logicalBlockIdx;
7      __shared__ int physicalBlockIdx;
8      if(threadIdx.x == 0) {
9          physicalBlockIdx = atomicAdd(&(block_index[smid + 1]), 1);
10     }
11     __syncthreads();
12     while(physicalBlockIdx < *max_blocks) {
13         if(threadIdx.x == 0) {
14             logicalBlockIdx = atomicAdd(&(block_index[0]), 1);
15             *concurrent_blocks = logicalBlockIdx;
16         }
17         __syncthreads();
18         if(logicalBlockIdx >= grid_size) {
19             break;
20         }
21         /*
22         ...
23         Kernel Code
24         ...
25         */
26     }
27     if(threadIdx.x == 0) {
28         atomicSub(&(block_index[smid + 1]), 1);
29     }
30 }

```

FIGURE 5.4: The kernel transformation required for supporting persistent threads and preemption.

transformation to the kernel code to implement persistent threads. It also includes support for preemption and control over the assignment of CTAs to SMs.

We add four input arguments to the original kernel: i) the size of the original grid to check whether the execution of the kernel has finished, ii) an array of block indices to keep the last logical block index and the last CTA index per SM, iii) a pointer to a memory location that keeps the maximum number of CTAs per SM, and iv) a pointer to a pinned memory location (as opposed to a pageable memory location) on the host memory that keeps the last logical block index for the host to read it.

At the kernel start, we determine the CTA index on the hosting SM. This requires the SM ID which is read from the `%smid` register on NVIDIA GPUs (line 5). Then, in one of the threads of the thread block, we atomically add to the value stored at the corresponding location to the hosting SM in the indices array, and store the result in a variable in shared memory (lines 8-10). We use shared memory for this purpose because it is faster than global memory. This is followed by a barrier to ensure all threads in the thread block observe the operation (line 11). This CTA index is then compared to the maximum number of CTAs per SM at every iteration of the loop so that the extra CTAs are preempted when necessary (line 12). Note that this feature, i.e., indexing CTAs on SMs to control the number of CTAs running on each SM, is our contribution and is not part of the original persistent threads transformation (the reason this is required is explained in Section 5.2.7 with the example scenario).

Inside the while loop that wraps the original kernel code, a new logical block is obtained in one of the threads of the thread block by atomically adding to the value stored at the memory location holding the last logical block index (lines 13-16). In addition, the last logical block index is written to the pinned memory pointed to by the `concurrent_blocks` variable (line 15). We allocate this memory on the host by calling the `cudaMallocHost()` function. This type of memory is accessible by both the host and the GPU. This way, we can monitor the progress of the kernel at any time without doing an explicit memory copy operation. After the new block index is obtained, all threads in the thread block are synchronized to observe it (line 17). Then, by comparing the logical block index to the size of the original grid, we can determine whether the execution of the kernel is finished (lines 18-20). If the logical block index is less than the size of the original grid, the original kernel code that has the `blockIdx` variable replaced by `logicalBlockIdx` is executed and the loop proceeds to the next iteration (lines 21-26).

The while loop terminates due to either completion of the kernel or preemption.

If the reason is preemption, we need to update the last CTA index on the SM (lines 27-29). This is necessary since it is possible that later on, we need to launch more CTAs of the same kernel, because, for instance, another kernel is completed and there is more room for other kernels on the GPU. In this case, the newly added CTAs must have valid indices in case in the future another preemption needs to occur.

5.2.5 Profiling and Pruning the Parameter Space

We use offline profiling of kernels in isolation to help estimate the remaining execution time of kernels in a multi-tenant environment, which is consumed by our allocation policy (Section 5.2.6). Once the transformation in Section 5.2.4 is applied to the kernel, we can use the number of CTAs as the control knob for the amount of resources allocated to it. After these data have been obtained (we will discuss the results of our profiling in Section 5.3.3), we sort the configuration points based on the number of CTAs and then prune the space such that the execution times of the remaining set of configurations monotonically decrease. In other words, this means that we eliminate all points for which another configuration exists that uses less resources and executes faster. Once the pruning is done, we store the remaining set of configurations in an array in the host-side service to be later retrieved by the allocation algorithm. Equation 5.1 shows how we use the profiling data for estimation of the remaining execution time of the kernel in a multi-tenant environment:

$$T_m^c = T_i^c \times \frac{TB_t - TB_e}{TB_t} \quad (5.1)$$

In Equation 5.1, T_m^c is the remaining execution time of the kernel in multi-tenant environment when it is running with c CTAs, T_i^c is its execution time in isolation when it has c CTAs, TB_t is the total number of its thread blocks, and TB_e is the number of thread blocks it has executed so far.

5.2.6 Sharing Policy

The policy that we implemented aims to minimize the maximum remaining execution time among the kernels to maximize STP. For this purpose, we use the execution times of the kernels in isolation as an estimation for their execution time in a multi-tenant environment (Equation 5.1).

Algorithm 2 Greedy resource allocation algorithm

```
1: procedure ALLOCATERESOURCES(KernelsList)
2:   Allocate the minimum resources to each kernel
3:   Descendingly sort kernels based on their estimated remaining execution times
4:   marked  $\leftarrow$  0
5:   while marked  $\leq$  KernelsList.size() do
6:     if KernelsList[marked].nextConfig() then
7:       KernelsList[marked].advanceToNextConfig()
8:       if New configurations fit the device then
9:         Re-sort the KernelsList from marked onwards
10:      else
11:        KernelsList[marked].rollBackToPrevConfig()
12:        marked++
13:      end if
14:    else
15:      marked++
16:    end if
17:  end while
18: end procedure
```

Our policy is a greedy method, in which the service starts at the point where all kernels have minimum resources, i.e., one CTA per kernel (line 2 in Algorithm 2.) The algorithm then descendingly sorts the kernels based on their estimated remaining execution times and initializes a variable, *marked*, to indicate the kernels whose resource allocation is determined (lines 3-4.) It then iteratively advances to the next configuration point for the first unmarked kernel (lines 6-7) until all kernels are marked. If this kernel currently has all the resources it can use, it is marked (line 15) and the loop proceeds to the next iteration. Note that due to sorting the kernels,

this operation minimizes the maximum remaining execution time among all kernels. If the new configuration fits the device, i.e., the resources required for it do not exceed those available on the GPU, the kernels are re-sorted and the loop proceeds to the next iteration (lines 8-9.) Otherwise, the operation is rolled back and the kernel is marked (lines 11-12.) It continues until no kernel can have more resources allocated to it. Note that these steps occur only in the host-side service and the final result is communicated to the applications, i.e., the incremental resource allocation is only in computation, we do not incrementally add to the CTAs a kernel runs with. Algorithm 2 shows the pseudo-code for this policy. If no feasible configuration exists, we simply launch all kernels with one CTA. In this case, kernels will be queued at the block scheduler on the GPU and will start execution once resources become available. The complexity of this algorithm is linear with respect to the number of configurations of all kernels, i.e., $O(\sum_{k \in K} C_k)$ where C_k is the number of configurations for kernel k and K is the set of all kernels that want to run.

It must be noted that any other policy can be easily plugged into our proposed system without affecting any of the parts related to the mechanisms necessary for supporting simultaneous multi-tenancy. The only requirement is that the policy needs to take a list of kernels and their profiling data as the input and determine the configurations for each kernel.

5.2.7 Example Scenario

An example scenario of two applications running kernels on the GPU concurrently is presented in Figure 5.5. At step ①, Application #1 contacts the host-side service requesting to run kernel A on the GPU. The service runs the resource allocation algorithm and responds to Application #1 to run kernel A with five CTAs per SM, and consequently Application #1 launches A with the specified number of CTAs.

Then at step ②, Application #2 sends a message to the host-side service and

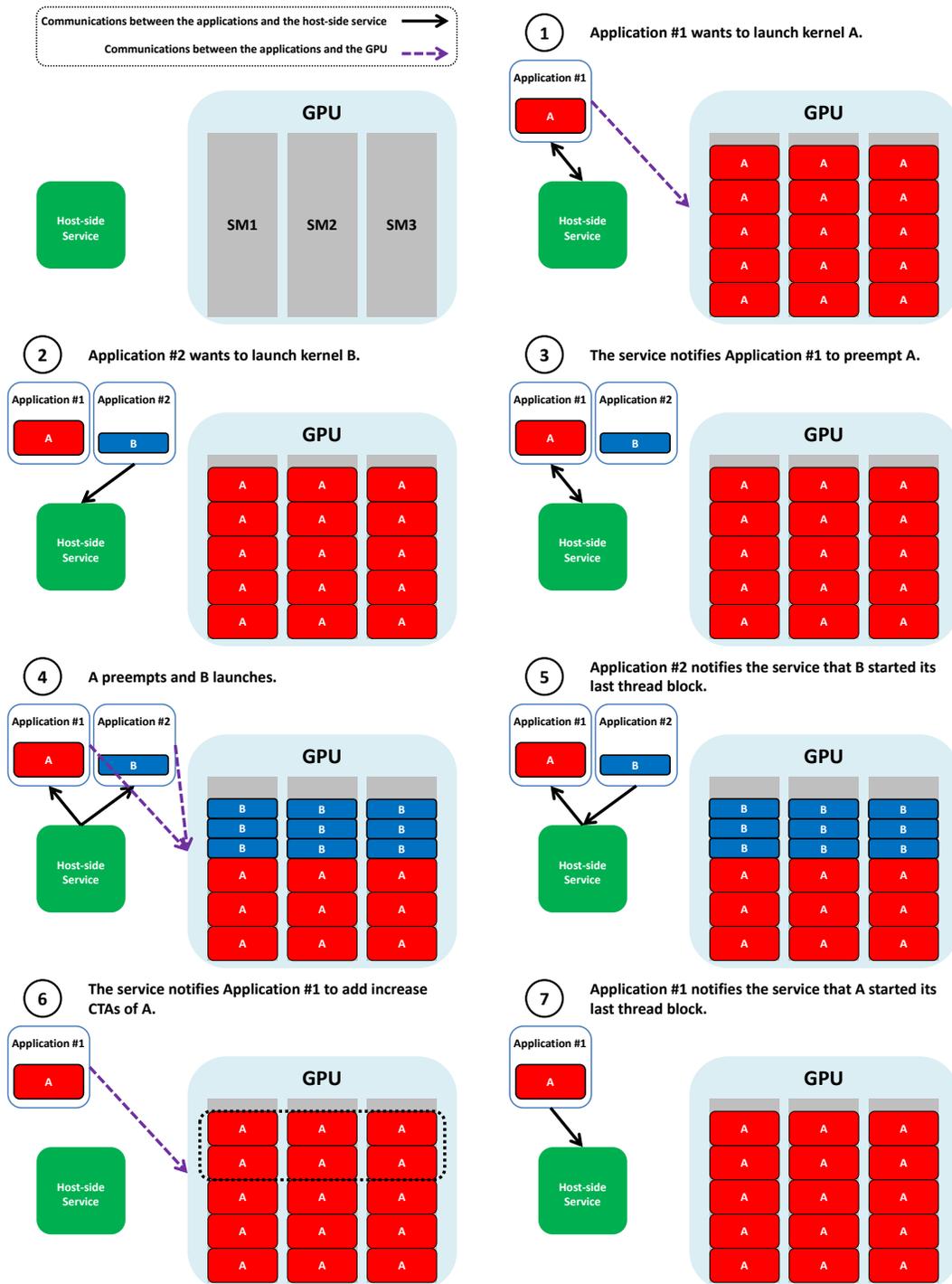


FIGURE 5.5: An example scenario of two applications running kernels on the GPU simultaneously.

requests to run kernel B on the device. The service queries Application #1 about the progress of kernel A at step ③, and Application #1 responds with the number of thread blocks that A has executed. Based on that information, the service runs the allocation algorithm and sends new launch parameters, i.e., three CTAs for each of A and B, to Applications #1 and #2 at step ④. At this step, Application #1 compares the new number of CTAs with what A is running with currently. Since the new number is smaller than the old one, preemption has to happen. Therefore, Application #1 writes the new value to the `max_blocks` location mentioned in Section 5.2.4. At the next iteration of the internal loop of A, the last two CTAs in each SM will preempt and make room for CTAs of kernel B. In parallel with this operation, Application #2 launches B with three CTAs per SM. Thread blocks of B will wait at the block scheduler on the GPU until resources become available to start execution.

Since CTAs of kernel A can preempt in an arbitrary order, it is possible that at some point all SMs are still occupied, except one with enough resources to host more than three CTAs of kernel B (normally we would want to fill SMs with as many CTAs as possible, but there are exceptions that having fewer CTAs will result in better performance as shown in [48] and also discussed in Section 5.3.3). For instance, suppose two CTAs of A preempt on SM1 before any of its CTAs preempt on SM2 or SM3. This makes enough room to accommodate four CTAs of B on SM1. The block scheduler on the GPU has nine CTAs of B waiting to be mapped on SMs, and since the mapping happens as soon as there are enough resources on the SM, four CTAs of B will be assigned to SM1 instead of the three intended. However, when this occurs in our system, extra CTAs will automatically terminate because their indices will exceed the value at `max_blocks`. This prevents having extra CTAs per SM, but in the end the total number of CTAs will be less than what is desired because some of them terminated early. To address this, applications that preempt CTAs can notify the host-side service, and then the service can notify those applications that launch

kernels to correct the number of CTAs if needed. However, since in our observations this situation did not occur, we did not include this fix in our implementation.

Once kernel B grabs its last thread block at step ⑤, Application #2 notifies the host-side service. This triggers running the allocation algorithm again. The service sends the new number of CTAs to Application #1, and since the new number is greater than the old one, new CTAs have to be launched. At step ⑥, Application #1 does so in a separate stream (CTAs inside the dotted rectangle), in order for them to run in parallel with the previous instance of the kernel. Kernel A grabs its last thread block at step ⑦ and Application #1 notifies the service of this event. Finally, Application #1 is finished and the GPU is empty.

5.2.8 Limitations and Future Work

A limitation of our work is that the host-side service logically shares a single SM among all kernels, and then extrapolates that configuration to all other SMs, although there might be kernels that benefit from having the entire SM to themselves due to their intensive usage of cache. This is an area for future work. One way to alleviate this shortcoming is to launch placeholder kernels from the host-side service to occupy a subset of SMs and force the GPU block scheduler to assign thread blocks to the free SMs. The overhead associated with this approach is expected to be minimal, since the placeholder kernels are only required for the short period of time between arriving a kernel launch request at the host-side service and the launch of the kernel at the GPU.

There are other limitations imposed by the choice of platform. One is due to the use of MPS in our system. NVIDIA GPUs do not support dynamic parallelism [47] while running MPS. Therefore, our system does not support running kernels that use this feature. The other, perhaps more important, limitation is that NVIDIA GPUs prior to the Volta [70] do not support running kernels with separate virtual address

Table 5.2: NVIDIA Tesla K40c specifications.

Resource	Value
Threads per SM	2048
Registers per SM	65536
Shared Memory per SM	48 KB
Warps per SM	64
Thread Blocks per SM	16

spaces, which creates security concerns. As discussed earlier in the chapter, adding a software address translation layer [80] can solve this problem by isolating the address spaces of different applications.

5.3 Evaluation

In this section, we first describe the platform for our experiments. Then we discuss the characteristics of the benchmark kernels we used. After that, the effect of the transformation in Section 5.2.4 on the performance of the benchmark kernels is evaluated. Finally, we present results for multi-kernel evaluations.

5.3.1 Platform

The machine we used for the experiments has an Intel Xeon E5-2640 CPU, and the experiments are conducted on an NVIDIA Tesla K40c GPU. The OS is Ubuntu 16.04, and NVIDIA driver version 375.26 and CUDA 8.0 were used to compile and run the benchmarks. Table 5.2 shows the specifications of the GPU card accounted for while making decisions about the feasibility of kernel configurations in the host-side service.

5.3.2 Benchmark Kernels

Our goal was to have a mixture of kernels from various areas with different behaviors and requirements. To this end, we picked seven benchmark kernels, `binomialOptions`, `FDTD3d`, `lavaMD`, `MD5hash`, `nbody`, `particlefilter`, and `tpacf`, from CUDA SDK samples [66], Rodinia [14], SHOC [20], and Parboil [84] benchmark suites, for

Table 5.3: Benchmark kernels characteristics.

Kernel	TBs¹	TB Size	Regs / T²	Shmem / TB³	Time⁴	ISU
binomialOptions (BO) [66]	1024	128	28	524	5.476	73.6
FDTD3d (FD) [66]	288	512	58	3848	8.821	27.5
lavaMD (LM) [14]	512	128	64	7208	8.958	87.3
MD5Hash (MD) [20]	25432	384	30	8	71.475	97.4
nbody (NB) [66]	128	256	49	8208	39.155	89.1
particlefilter (PF) [14]	512	128	16	8	43.105	48.6
tpacf (TP) [84]	201	256	49	13320	11.23	34.8

¹ Thread Blocks.

² Registers per Thread.

³ Shared Memory per Thread Block in Bytes.

⁴ Execution Time in ms.

our evaluations. Figure 5.6 shows the behavior of these benchmarks that are representative of a variety of kernels. The values on the axes are on a scale of 0-10 and are obtained from NVIDIA profiler. The figure shows that some kernels are compute-intensive (MD5Hash, lavaMD), some demonstrate intensive use of memory and cache (FDTD3d), and some have a mixture of requirements (binomialOptions, nbody, particlefilter, tpacf).

Table 5.3 summarizes the characteristics of these benchmark kernels, after undergoing the transformations to support adaptive simultaneous multi-tenancy explained in Section 5.2.4. The abbreviations in front of kernels' names are used in multi-kernel evaluation figures.

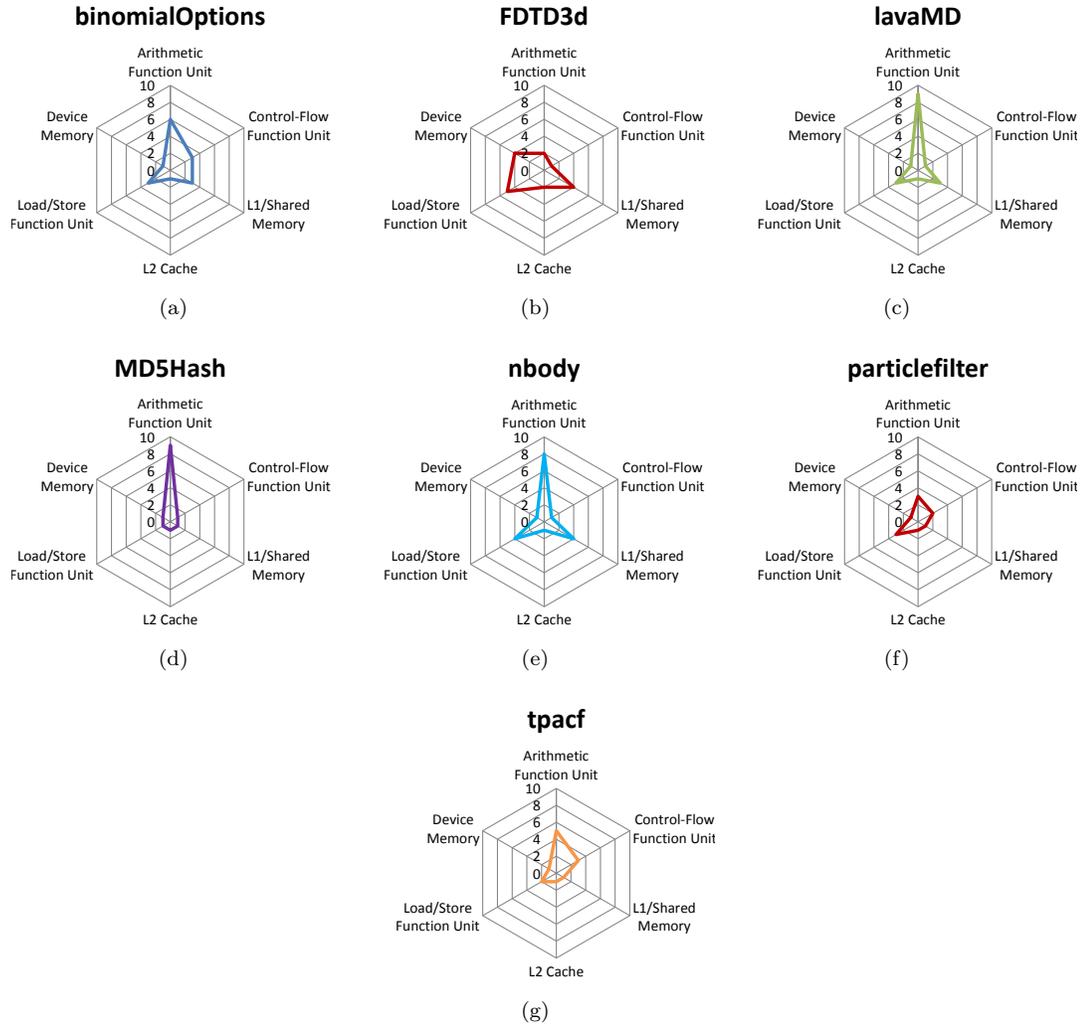


FIGURE 5.6: Utilization of various resource types in (a) binomialOptions, (b) FDTD3d, (c) lavaMD, (d) MD5Hash, (e) nbody, (f) particlefilter, and (g) tpacf kernels.

5.3.3 Single Kernel Performance

Figure 5.7a shows the normalized execution time of the transformed kernels with respect to the original code. As the figure illustrates, applying the transformation to the kernels has a negligible impact of 1.7% on the average performance of all kernels. However, it increases the register and shared memory usage of the kernels due to introducing additional variables. The register usage is increased by 23%, as shown in Figure 5.7b. Increasing the number of registers per thread might result in

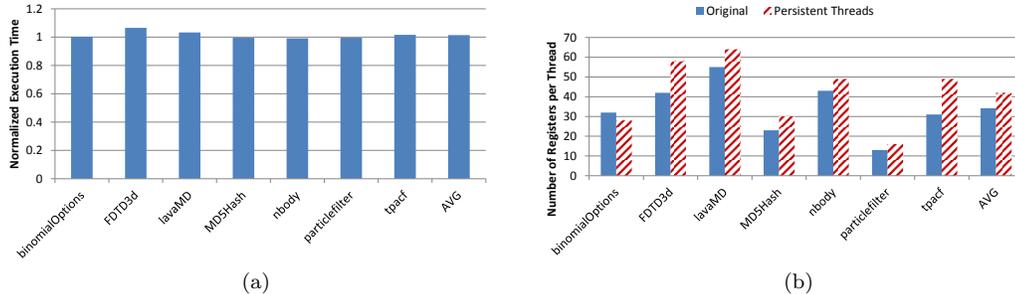


FIGURE 5.7: (a) Performance and (b) register usage of benchmark kernels under persistent threads transformation.

fewer CTAs fitting on the GPU. Nevertheless, as shown in Figure 5.8, increasing the number of CTAs has a marginal gain and after some point, the performance does not considerably improve. Nonetheless, a possible solution for reducing the register overhead of the proposed code transformations is to restrict the compiler to compile kernels with fewer registers. This incurs some performance overhead to the kernels, but based on our observations, accepting a 2% performance overhead results in the elimination of register usage overhead. The reason that we did not take this into consideration is that the same could be applied to the original kernels for register reduction. Thus, we picked the best-performing register configuration for both the original and transformed kernels. We did not include the shared memory usage figures, because all kernels need a fixed eight bytes additional shared memory to store the logical block index and CTA IDs in SMs required for the persistent threads transformation.

As stated in Section 5.2.4, applying persistent threads transformation to kernels allows us to control the allocated resources by running them with the desired number of CTAs. Figure 5.8 shows the performance of transformed kernels for varying numbers of CTAs per SM. We use these data as input for our greedy allocation algorithm. Most of the time, there is a direct trade-off between the number of CTAs of a kernel and its performance. However, there are some exceptions for binomialOptions

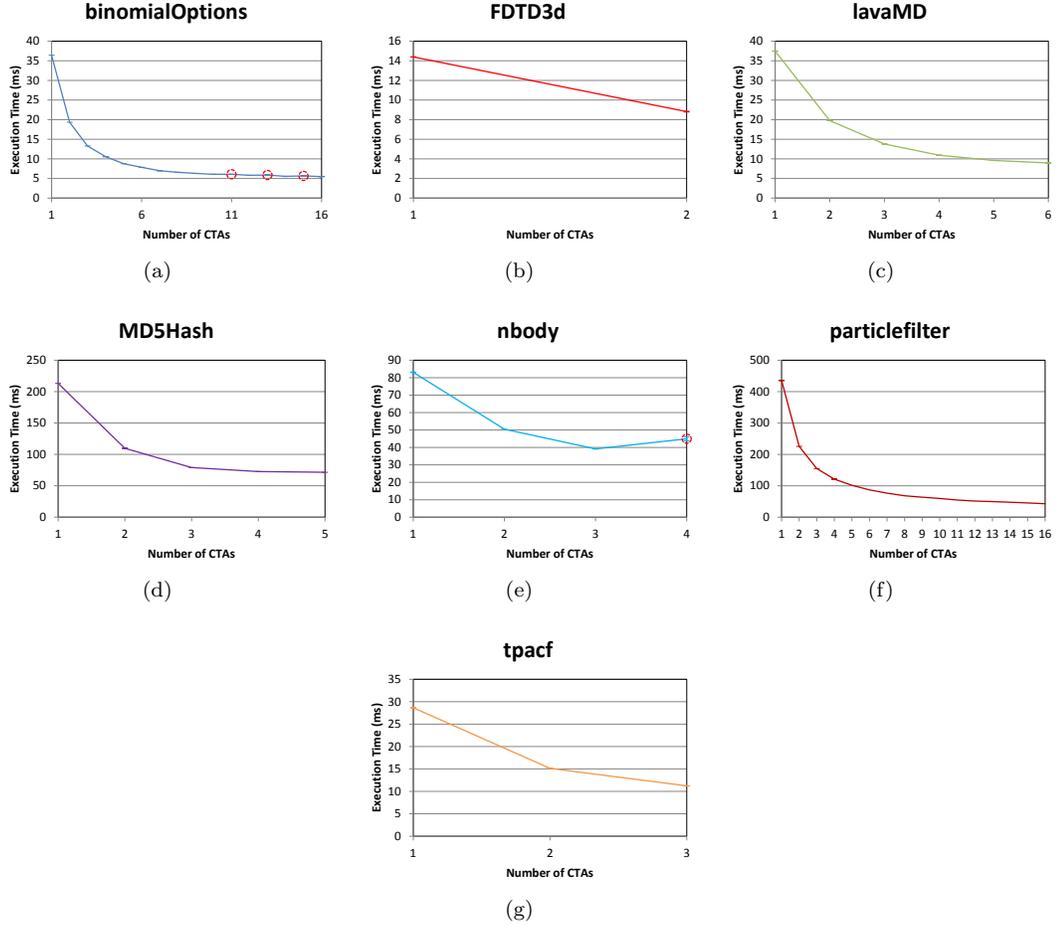


FIGURE 5.8: Performance of (a) binomialOptions, (b) FDTD3d, (c) lavaMD, (d) MD5Hash, (e) nbody, (f) particlefilter, and (g) tpacf kernels with different numbers of CTAs.

and nbody that are distinguished with red dotted circles. We omit these points from the decision-making process in the allocation algorithm, since they do not offer any beneficial trade-off. In other words, there exists another point in the space that uses less resources but delivers higher performance.

5.3.4 Multi-Kernel Performance

In this section, we report two metrics that are common for measuring the performance of multi-program workloads [24]: i) system throughput (STP), and ii) average normalized turnaround time (ANTT) for kernels. We use the time it takes for all

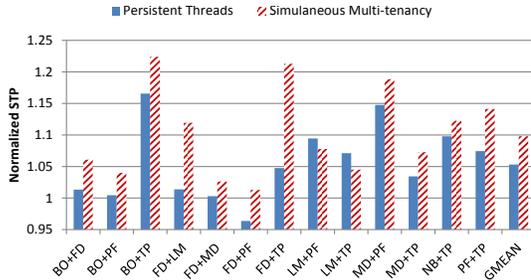


FIGURE 5.9: Normalized STP under different combinations of kernels including at least one kernel with low ISU.

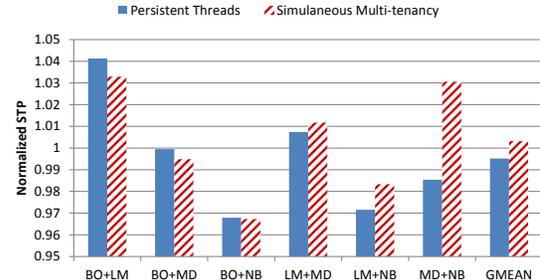


FIGURE 5.10: Normalized STP under different combinations of kernels with high ISUs.

kernels to finish, i.e., the completion time of the last kernel that finishes minus the start time of the first kernel that begins, as an indication for STP. ANTT is the ratio of the time it takes for a kernel to finish in a multi-tenant environment and the time it takes for the same kernel to finish in isolation. Unfortunately, we cannot report ISU for multi-kernel experiments since NVIDIA profiler does not report it when MPS is running.

We also report results for two systems: i) a system in which there is no host-side service, but persistent thread transformation is applied to the kernels (PT), and ii) our proposed adaptive simultaneous multi-tenant system (SiM).

We repeated our experiments five times for every ordering of the kernels (e.g., for the combination of BO+FD, five times when BO arrived at the service first, five times when FD was the first, and for SiM only, five times when both arrived at almost the same time such that their effects were aggregated), and report the average of the results.

Kernels with Low ISU

We refer to kernels with an ISU of less than 50% as low ISU kernels. During the execution of such kernels, the execution units are idle for more than half of the cycles due to various reasons, including synchronization, data request, execution dependency, busy pipeline, etc. It is expected that co-scheduling these kernels with

another kernel results in higher STP, since the idle cycles can be taken advantage of. Figure 5.9 demonstrates the normalized STP for kernel combinations that include at least one kernel with low ISU. On average, PT improves STP by 5.3% with respect to the sequential execution of kernels. This improvement is due to the alleviation of the head-of-line blocking in the GPU block scheduler explained in Section 5.1.3. Nevertheless, addressing this issue alone is not sufficient to realize the potential STP improvement created by the underutilization of resources. To this end, by tuning the resources allocated to each kernel, SiM increases STP by 9.8%.

Not all kernel pairs experience similar improvements in STP. The higher the ISU of one of the kernels is, the less opportunities there are for STP improvement. This is evident in FD+MD and MD+TP pairs, because MD kernel has an ISU of 97.4%. This means that MD alone can utilize the device very well. There are other factors that impact the achieved STP improvement as well, such as the resource requirement of the kernels, the execution time of a single thread block of the kernels, and non-optimal allocation of resources by our greedy algorithm. The first one shows itself in the high STPs achieved when running BO+TP and FD+TP pairs, since these kernels utilize different units on the device. The other two, however, explain the low STP improvement for LM+TP. In this kernel pair, whenever LM arrives first, there is no room for TP to run any thread blocks until the first round of thread blocks of LM are completed and preemption can occur. This takes long enough to offset a large enough fraction of the improvement achieved by the co-run of the rest of the thread blocks of the two kernels to cause lower improvements than simply running the two kernels with persistent thread transformation. In addition, the launch parameters determined by the greedy algorithm for the two kernels do not result in the best possible output. This highlights the need for a more accurate and sophisticated allocation policy, which is part of our future work.

The gains in STP come at the expense of 49.2% increase in ANTT. We must note

that improving STP and keeping ANTT low are at odds with each other, and the goal of our allocation policy is to maximize STP. If ANTT is an important factor in the system, other allocation algorithms can replace our greedy algorithm without affecting other parts of the system. Besides, we did not define priorities for kernels in our work. The works that target improving ANTT do so for high priority kernels since it is impossible to improve this metric for all kernels in the system.

Kernels with High ISU

The STP improvement is not significant when all kernels running on the GPU can utilize it well enough when executed in isolation. Figure 5.10 shows the normalized STP for pairs of kernels that both have high ISUs (i.e., greater than 50%). In these cases, the overheads of preemption and multiple launches, as well as cache thrashing, result in a negligible STP improvement of 0.3% in SiM. PT even imposes a 0.5% overhead on STP. The other downside of running high-ISU kernels together is a large increase in ANTT (78.7%).

These observations mean that adaptive simultaneous multi-tenancy is more effective when individual kernels are not highly optimized to have high ISUs. The positive side is that this also means that without putting extra effort into optimizing kernels, a higher STP can be achieved by merely running multiple kernels together.

5.4 Related Work

Gupta et al. [36] studied the different use cases of persistent threads for a single kernel. In independent works, Chen et al. [15] and Wu et al. [101] proposed taking advantage of persistent threads for supporting preemption. EffiSha only supports the execution of one kernel at any given time on the GPU [15]. FLEP, on the other hand, has limited support for executing two kernels on the GPU at the same time, only in the case that one of the kernels is small enough to entirely fit on the GPU

[101]. Supporting preemption in these works helps favoring the high priority kernels over the low priority ones, but it does not solve the underutilization of resources on the device.

Pai et al. [72] proposed Elastic Kernels and showed that sharing the GPU among multiple kernels improves utilization. They artificially fuse multiple kernels together to form a super-kernel in a single GPU context. There are other works that adopt a similar approach [34, 55, 100]. This allows for concurrent execution of kernels, but such a scenario is impractical in the real world, since merging the kernels from different clients into a single kernel at run-time is impossible. We avoid this limitation by using a host-side service and taking advantage of MPS. Furthermore, in this work we focus on a different part of the solution. We propose a system to solve the problem of multi-tenancy while Elastic Kernels proposes various policies that can be employed in our host-side service.

Preemptive Kernel Model proposed by Basaran and Kang [10] slices the kernel into smaller grids, which in turn allows for sharing the GPU among multiple kernels. Several other works also rely on kernel slicing [44, 107]. This approach incurs the overhead of multiple launches that cannot be avoided even if we do not need to preempt the kernel at all. By taking advantage of the persistent threads model, our approach eliminates the unnecessary launch overhead introduced by kernel slicing.

The NVIDIA Volta architecture [70] and newer architectures support static simultaneous multi-tenancy. In other words, it is possible to divide the GPU into multiple smaller virtual GPUs. There are also works that introduce hardware extensions to support preemption or multi-programming. Tanasic et al. [90] proposed context switching and draining by supporting preemption in hardware. Park et al. [73] extended this work by identifying idempotent kernels to faster preempt the running kernel by flushing the SMs.

Adriaens et al. [1] proposed spatial multi-tasking. In this approach, each SM is

entirely allocated to one kernel. Wang et al. [97] propose partial context switching, which is similar to our approach in that it only preempts a portion of SMs. Nevertheless, they use different allocation policies and hardware support is necessary for its implementation. Xu et al. [103] propose a software-hardware mechanism that similarly shares an SM among multiple kernels. Park et al. [74] propose GPU Maestro that based on performance predictions, switches between spatial multi-tasking and partial context switching at run-time.

Jog et al. [45] proposed a method for making memory scheduling decisions based on misses-per-kilo-instructions and bandwidth information in multi-application environments in GPUs. We did not investigate the efficiency of memory operations in this work. Moreover, Wang et al. [94] demonstrated the effects of thread-level parallelism on the overall performance of the system when multiple kernels are running together, and showed that making bandwidth management decisions considering all kernels is more effective than allocating resources to each individual kernel based on its performance metrics in isolation. We did not study the effects of memory bandwidth on kernels' performance in this work. In addition, in their work each SM is dedicated to a single kernel as opposed to our proposed solution to share SMs among multiple kernels to improve resource utilization.

Aguilera et al. [2] investigated the fairness of spatial multi-tasking and proposed task assignment methods to improve fairness. Wang et al. [98] proposed quality of service support for fine-grained sharing on GPUs. Fairness was not our main goal in this paper, although our resource allocation policy can be easily swapped with any other policy that improves fairness or other desired metrics.

5.5 Summary

We identify the challenges of using GPUs in a multi-tenant environment. We propose adaptive simultaneous multi-tenancy for GPUs to overcome these challenges.

Our approach comprises a host-side service that makes decisions about the kernel launch parameters and when the kernels should preempt. We also provide an API to facilitate using the system for programmers and allow kernels to dynamically adapt resource usage at runtime. Our approach requires minimal kernel modifications. Evaluation of our prototype system on NVIDIA K40c GPUs shows that, on average, the system throughput is improved by 9.8% for combinations of kernels that include at least one low-utilization kernel. This improvement is achieved at the cost of 49.2% increase in the average normalized turn around time. Combinations of high-utilization kernels do not benefit from our system. Our observations indicate that using adaptive simultaneous multi-tenancy allows programmers to avoid highly optimizing their kernels to have high ISUs by providing higher STP for concurrent execution of low-utilization kernels.

6

Conclusions

Executing data-parallel applications in the age of big data is challenging, because it needs a significant amount of computational power. Furthermore, the end of Moore’s law only exacerbates this issue as we can no longer rely on scaling transistors to gain higher performance. Consequently, meeting the performance requirements of data-parallel applications necessitates the adoption of specialized accelerators such as Graphics Processing Units (GPUs) and more application-specific solutions. Furthermore, software techniques and algorithmic optimizations can be used in addition to hardware specialization.

In this dissertation, we identify multiple data-parallel application domains that are suitable targets for hardware acceleration and algorithmic optimization. First, we focus on statistical machine learning, and in particular, probabilistic algorithms such as Markov Chain Monte-Carlo (MCMC). These algorithms are computationally expensive, yet their statistical properties make them an attractive alternative approach to DNNs. We target accelerating these algorithms for Markov Random Field (MRF) inference in Chapter 3. We observe that problems represented by a first-order MRF exhibit a high degree of spatial locality in memory accesses, and

also have a regular memory access pattern. Thus, we design a tiled architecture to exploit near-memory computing, which allows us to avoid frequent costly off-chip communication. In addition, we design optimal communication schemes that take memory access patterns into account to eliminate the need for a full-blown network-on-chip. Our proposed accelerator achieves significant speedup compared to general-purpose processors and even GPUs. Furthermore, we are able to support uncertainty quantification by employing a hybrid on-chip/off-chip memory system, which is inspired by workload characterization for two image analysis applications. Our observations indicate that most Random Variables (RVs) only take on a few labels, and therefore, by caching the most recently picked labels on the chip and sending other labels to off-chip memory, we can strike a balance between the on-chip memory capacity and off-chip communication bandwidth.

Nevertheless, hardware specialization alone does not provide the optimal performance. Taking domain knowledge into consideration and tackling the problem from an algorithmic perspective can deliver sizable performance gains. To this end, orthogonal to hardware techniques, we demonstrate that algorithmic optimizations can and should be adopted to further accelerate probabilistic algorithms. In Chapter 4, we highlight the relation between MRF inference and the concept of vertex programming in graph algorithms. Moreover, we show that many RVs update operations do not result in new labels. Running MCMC in the optimization mode and using approximation techniques only amplifies this behavior, resulting in many RVs having Probability Distribution Functions (PDFs) concentrated on only one label, which then allows us to skip updating those RVs in the next iteration if the PDF remains unchanged. Based on these observations, we propose Event-Driven Gibbs Sampling (EDGS) to detect when RVs are stable and accelerate the application by skipping the update operation for those RVs.

Considering that we approached the problem of accelerating MRF inference using

MCMC from the hardware and algorithmic perspectives separately, an area for future work is enhancing the MRF inference accelerator by incorporating EDGS. Although the proposed accelerator in Chapter 3 achieves high performance, especially in the sampling mode, it can still benefit from EDGS in the optimization mode. However, the integration of EDGS will not be without challenges. The proposed accelerator relies heavily on the regular memory access and communication pattern in the MRF, whereas EDGS breaks that order. To be more specific, EDGS accelerates Gibbs sampling by skipping some RV updates, and this creates load imbalance in a tiled architecture because not all regions in the input converge at the same rate. As a result, the tiled architecture will not be able to exploit the full potential offered by EDGS, mainly because the performance of the accelerator will be determined by the slowest tile. To alleviate this issue, a centralized scheduler may be a better option.

Moreover, implementing EDGS for other platforms is another area of future work. GPUs offer high throughput and require relatively low programming effort. However, their SIMT execution model limits the effectiveness of EDGS. Other platforms that support fine-grained parallelism more efficiently might be able to gain higher performance.

In addition to probabilistic algorithms, GPUs can be utilized to accelerate a wide range of applications. GPUs offer massive parallelism by providing a variety of resources, including thousands of simple cores, register files, etc. The large amount and diversity of resources on the GPU ensures that diverse workloads with different characteristics achieve high performance, but at the same time, in Chapter 5, we show that it leads to under-utilization of some of the resources, which is inefficient in a multi-tenant environment. We identify the challenges to efficiently share a GPU between multiple kernels in such an environment, which include adaptiveness to the events in the system, and flexibility in terms of allocating resources to kernels. We propose a system to address these challenges by exploiting appropriate code

transformations, facilitated through an API provided by us, and a host-side service responsible for making decisions about what resources are allocated to each kernel, and when kernels need to resize. We analyze the effectiveness of the system under different workload scenarios and show that for combinations of kernels that include at least one low-utilization kernel, it improves the system throughput. As a result, programmers can avoid highly optimizing their kernels to have high resource utilization, because using our system for concurrent execution of low-utilization kernels enhances the system throughput.

Implementing more mapping, allocation, and scheduling algorithms for adaptive simultaneous multi-tenancy for GPUs is an area for future work. Our observation of some kernels' behaviors indicate that more sophisticated mapping algorithms and allocation policies to support asymmetric mapping of thread blocks to SMs can further improve the performance of our proposed system. This way, cache-intensive kernels can benefit from having the entire L1 cache to themselves. Additionally, more intelligent scheduling algorithms for dispatching thread blocks to SMs can improve locality and reduce destructive interference in L2 cache. Despite all of this, however, our observations show that due to the overheads of communication with the host and coarse granularity of thread block preemption, the full potential of sharing a GPU among multiple kernels is better realized with a hybrid hardware-software approach that enables quick and flexible context switches, control over mapping and scheduling, and takes kernel resource utilization information into consideration.

Bibliography

- [1] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for gpgpu spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
- [2] P. Aguilera, K. Morrow, and N. S. Kim. Fair share: Allocation of gpu resources for both performance and fairness. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 440–447, Oct 2014.
- [3] Amazon Web Services. Elastic gpus, 2017.
- [4] Shervin Rahimzadeh Arashloo and Josef Kittler. Fast pose invariant face recognition using super coupled multiresolution markov random fields on a gpu. *Pattern Recognition Letters*, 48:49–59, 2014.
- [5] Jon Arróspide, Luis Salgado, and Marcos Nieto. Video analysis-based vehicle detection and tracking using an mcmc sampling framework. *EURASIP Journal on Advances in Signal Processing*, 2012(1):2, 2012.
- [6] Simon Baker, Daniel Scharstein, J. P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, 92(1):1–31, Mar 2011.
- [7] Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. *AcMC²*: Accelerating markov chain monte carlo algorithms for probabilistic models. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 515–528, New York, NY, USA, 2019. Association for Computing Machinery.
- [8] Rémi Bardenet, Arnaud Doucet, and Chris Holmes. On markov chain monte carlo methods for tall data. *The Journal of Machine Learning Research*, 18(1):1515–1557, 2017.
- [9] Stephen T Barnard. Stochastic stereo matching over scale. *International Journal of Computer Vision*, 3(1):17–32, 1989.

- [10] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in gpgpus. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, pages 287–296, Washington, DC, USA, 2012. IEEE Computer Society.
- [11] Ramin Bashizade, Yuxuan Li, and Alvin R. Lebeck. Adaptive simultaneous multi-tenancy for gpus. In *Job Scheduling Strategies for Parallel Processing*, pages 83–106, Cham, 2019. Springer International Publishing.
- [12] J. . Cardoso. Blind signal separation: statistical principles. *Proceedings of the IEEE*, 86(10):2009–2025, 1998.
- [13] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 103–116, New York, NY, USA, 2001. ACM.
- [14] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [15] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling efficient preemptive scheduling of gpu. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '17, pages 3–16, New York, NY, USA, 2017. ACM.
- [16] Wenjun Cheng, Luyao Ma, Tiejun Yang, Jiali Liang, and Yan Zhang. Joint lung ct image segmentation: a hierarchical bayesian approach. *PloS one*, 11(9), 2016.
- [17] J. Andrés Christen and Colin Fox. Markov chain monte carlo using an approximation. *Journal of Computational and Graphical Statistics*, 14(4):795–810, 2005.
- [18] Alessandro Cilardo and Luca Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Trans. Archit. Code Optim.*, 11(4):45:1–45:25, January 2015.
- [19] D. Crookes, P. Miller, H. Gribben, C. Gillan, and D. McCaughey. Gpu implementation of map-mrf for microscopy imagery segmentation. In *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 526–529, 2009.

- [20] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3*, pages 63–74, New York, NY, USA, 2010. ACM.
- [21] J. Escobedo and M. Lin. Tessellating memory space for parallel access. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 75–80, Jan 2017.
- [22] J. Escobedo and M. Lin. Extracting data parallelism in non-stencil kernel computing by optimally coloring folded memory conflict graph. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018.
- [23] Juan Escobedo and Mingjie Lin. Graph-theoretically optimal memory banking for stencil-based computing kernels. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18*, pages 199–208, New York, NY, USA, 2018. ACM.
- [24] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.
- [25] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.
- [26] Bin Fang and Peng Zhang. *Big Data in Finance*, pages 391–412. Springer International Publishing, Cham, 2016.
- [27] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics, ACL '05*, page 363–370, USA, 2005. Association for Computational Linguistics.
- [28] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. Big data, simulations and hpc convergence. In Tilmann Rabl, Raghunath Nambiar, Chaitanya Baru, Milind Bhandarkar, Meikel Poess, and Saumyadipta Pyne, editors, *Big Data Benchmarking*, pages 3–17, Cham, 2016. Springer International Publishing.
- [29] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741, 1984.

- [30] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.
- [31] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 324–332, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.
- [32] Google. Google cloud platforms, 2017.
- [33] Mentor Graphics. Catapult® high-level synthesis, 2020.
- [34] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained Resource Sharing for Concurrent GPGPU Kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar’12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.
- [35] Gautam Gupta and Sanjay Rajopadhye. The z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’07, pages 237–248, New York, NY, USA, 2007. ACM.
- [36] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, pages 1–14, May 2012.
- [37] Ghassan Hamra, Richard MacLehose, and David Richardson. Markov chain monte carlo: an introduction for epidemiologists. *International journal of epidemiology*, 42(2):627–634, 2013.
- [38] Jonathan C. Hedstrom, Chung Him Yuen, Rong-Rong Chen, and Behrouz Farhang-Boroujeny. Achieving near map performance with an excited markov chain monte carlo mimo detector. *Trans. Wireless. Comm.*, 16(12):7718–7732, December 2017.
- [39] S. Hurkat and J. F. Martínez. Vip: A versatile inference processor. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–358, 2019.
- [40] Intel. Intel® high level synthesis compiler, 2019.
- [41] Intel. Intel® pac with intel arria® 10 gx fpga, 2019.
- [42] Intel. Intel® platform designer, 2019.

- [43] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.
- [44] Qing Jiao, Mian Lu, Huynh Phung Huynh, and Tulika Mitra. Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 1–11, Washington, DC, USA, 2015. IEEE Computer Society.
- [45] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 223–234, New York, NY, USA, 2015. ACM.
- [46] Eric Michael Jonas. *Stochastic architectures for probabilistic computation*. PhD thesis, Massachusetts Institute of Technology, 2014.
- [47] Stephen Jones. Introduction to dynamic parallelism. In *Nvidia GPU Technology Conference*. NVIDIA, 2012.
- [48] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, Sept 2013.
- [49] Minje Kim, Paris Smaragdis, Glenn G Ko, and Rob A Rutenbar. Stereophonic spectrogram segmentation using markov random fields. In *2012 IEEE International Workshop on Machine Learning for Signal Processing*, pages 1–6. IEEE, 2012.
- [50] Y. Kim, M. Imani, and T. Rosing. Orchard: Visual object recognition accelerator based on approximate in-memory processing. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 25–32, 2017.
- [51] G. G. Ko, Y. Chai, R. A. Rutenbar, D. Brooks, and G. Wei. Accelerating bayesian inference on structured graphs using parallel gibbs sampling. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 159–165, 2019.
- [52] G. G. Ko, Y. Chai, R. A. Rutenbar, D. Brooks, and G. Wei. Flexgibbs: Reconfigurable parallel gibbs sampling accelerator for structured graphs. In *2019*

IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 334–334, 2019.

- [53] J. Konrad and E. Dubois. Bayesian estimation of motion vector fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(9):910–927, Sept 1992.
- [54] Stan Z Li. *Markov random field modeling in computer vision*. Springer Science & Business Media, 2012.
- [55] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. Efficient gpu spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):748–760, March 2015.
- [56] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.
- [57] A. L’Heureux, K. Grolinger, H. F. Elyamany, and M. A. M. Capretz. Machine learning with big data: Challenges and approaches. *IEEE Access*, 5:7776–7797, 2017.
- [58] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.
- [59] David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 416–423. IEEE, 2001.
- [60] Patrick McClure, Nao Rho, John A. Lee, Jakub R. Kaczmarzyk, Charles Y. Zheng, Satrajit S. Ghosh, Dylan M. Nielson, Adam G. Thomas, Peter Bandettini, and Francisco Pereira. Knowing what you know in brain segmentation using bayesian deep neural networks. *Frontiers in Neuroinformatics*, 13:67, 2019.
- [61] Mentor. Algorithmic c datatypes, 2020.
- [62] Microsoft. Microsoft azure, 2016.
- [63] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

- [64] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.
- [65] NanGate. Nangate freepdk15 generic open cell library, 2017.
- [66] Nvidia. CUDA programming guide, 2008.
- [67] Nvidia. Next generation CUDA computer architecture Kepler GK110, 2012.
- [68] NVIDIA. Pascal architecture whitepaper, June 2015.
- [69] NVIDIA. Multi-process service, 2017.
- [70] NVIDIA. Volta architecture whitepaper, June 2017.
- [71] NVIDIA. Turing architecture whitepaper, 2020.
- [72] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 407–418, New York, NY, USA, 2013. ACM.
- [73] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 593–606, New York, NY, USA, 2015. ACM.
- [74] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, pages 527–540, New York, NY, USA, 2017. ACM.
- [75] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July 2003.
- [76] Matias Quiroz, Robert Kohn, Mattias Villani, and Minh-Ngoc Tran. Speeding up mcmc by efficient data subsampling. *Journal of the American Statistical Association*, 114(526):831–843, 2019.
- [77] M. Randles, D. Lamb, and A. Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 551–556, April 2010.

- [78] Christian P Robert, Víctor Elvira, Nick Tawn, and Changye Wu. Accelerating mcmc algorithms. *Wiley Interdisciplinary Reviews: Computational Statistics*, 10(5):e1435, 2018.
- [79] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*, pages 131–140, Dec 2001.
- [80] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: A case for software address translation on gpus. In *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16*, pages 596–608, Piscataway, NJ, USA, 2016. IEEE Press.
- [81] Alistair Sinclair and Mark Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Information and Computation*, 82(1):93–133, 1989.
- [82] B. So, M. W. Hall, and H. E. Ziegler. Custom data layout for memory parallelism. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 291–302, March 2004.
- [83] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454, 2017.
- [84] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Lie, and Wen-mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing, 2012.
- [85] Marc A. Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. Understanding gpu programming for statistical computation: Studies in massively parallel massive mixtures. *Journal of Computational and Graphical Statistics*, 19(2):419–438, 2010. PMID: 20877443.
- [86] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.
- [87] Synopsys. Design compiler, 2019.
- [88] Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen, and Carsten Rother. A comparative study of energy minimization methods for markov random fields with

- smoothness-based priors. *IEEE transactions on pattern analysis and machine intelligence*, 30(6):1068–1080, 2008.
- [89] Tamás Szirányi, Josiane Zerubia, László Czúni, David Geldreich, and Zoltán Kato. Image segmentation using markov random field model in fully parallel cellular network architectures. *Real-Time Imaging*, 6(3):195–211, 2000.
- [90] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecture, ISCA '14*, pages 193–204, Piscataway, NJ, USA, 2014. IEEE Press.
- [91] Alexander Terenin, Shawfeng Dong, and David Draper. Gpu-accelerated gibbs sampling: a case study of the horseshoe probit model. *Statistics and Computing*, 29(2):301–310, 2019.
- [92] Thomas N Theis and H-S Philip Wong. The end of Moore’s law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.
- [93] S Thoziyoor, N Muralimanohar, JH Ahn, and N Jouppi. Cacti 5.3. *HP Laboratories, Palo Alto, CA*, 2008.
- [94] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. Efficient and fair multiprogramming in gpus via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 247–258, 2018.
- [95] S. Wang, X. Zhang, Y. Li, R. Bashizade, S. Yang, C. Dwyer, and A. R. Lebeck. Accelerating markov random field inference using molecular optical gibbs sampling units. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 558–569, June 2016.
- [96] Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, FPGA '14*, pages 199–208, New York, NY, USA, 2014. ACM.
- [97] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369, March 2016.
- [98] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of service support for fine-grained sharing on gpus. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17*, pages 269–281, New York, NY, USA, 2017. ACM.

- [99] Neil HE Weste and David Harris. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.
- [100] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and Exploiting Flexible Task Assignment on GPU Through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15*, pages 119–130, New York, NY, USA, 2015. ACM.
- [101] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, pages 483–496, New York, NY, USA, 2017. ACM.
- [102] Z. Wu, Q. Wang, A. Plaza, J. Li, L. Sun, and Z. Wei. Parallel spatial–spectral hyperspectral image classification with sparse representation and markov random fields on gpus. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8(6):2926–2938, 2015.
- [103] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multi-programming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242, June 2016.
- [104] Allen Y Yang, John Wright, Yi Ma, and S Shankar Sastry. Unsupervised segmentation of natural images via lossy data compression. *Computer Vision and Image Understanding*, 110(2):212–225, 2008.
- [105] X. Zhang, R. Bashizade, C. LaBoda, C. Dwyer, and A. R. Lebeck. Architecting a stochastic computing unit with molecular optical devices. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 301–314, June 2018.
- [106] Xiangyu Zhang, Ramin Bashizade, Yicheng Wang, Cheng Lyu, Sayan Mukherjee, and Alvin R Lebeck. Beyond application end-point results: Quantifying statistical robustness of mcmc accelerators. *arXiv preprint arXiv:2003.04223*, 2020.
- [107] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1522–1532, June 2014.
- [108] Stephanie Zierke and Jason D Bakos. Fpga acceleration of the phylogenetic likelihood function for bayesian mcmc inference methods. *BMC bioinformatics*, 11(1):1–12, 2010.

Biography

Ramin Bashizade received his B.Sc. in computer engineering in 2011 from Shahed University in Tehran, Iran, and he earned his M.Sc. in computer engineering in 2013 from Sharif University of Technology in Tehran, Iran. He then worked for two years as a software developer at Amin Software. During his time there, he worked on the graphical user interface for a soccer analysis software. In 2015, he started his Ph.D. journey in computer science at Duke University, where he conducted research on parallel computer architectures and systems. During this time, he contributed to several projects [95, 105, 106], and his work on GPU multi-tenancy was published in JSSPP 2018 [11]. He also interned as a systems and infrastructure engineer at LinkedIn, where he worked on improving resource utilization and automation in computing clusters. After his Ph.D., he will join LinkedIn in January 2021.