gcn.MOPS: Accelerating cn.MOPS with GPU

by

Mohammad Alkhamis
BASc, University of Ottawa, 2012

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

© Mohammad Alkhamis, 2017
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

gcn.MOPS: Accelerating cn.MOPS with GPU

by

Mohammad Alkhamis

BASc, University of Ottawa, 2012

Supervisory Committee

Dr. Amirali Baniasadi, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Nikitas Dimopoulos, Departmental Member

(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Amirali Baniasadi, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Nikitas Dimopoulos, Departmental Member

(Department of Electrical and Computer Engineering)

ABSTRACT

cn.MOPS is a model-based algorithm used to quantitatively detect copy-number variations in next-generation, DNA-sequencing data. The algorithm is implemented as an R package and can speed up processing with multi-CPU parallelism. However, the maximum achievable speedup is limited by the overhead of multi-CPU parallelism, which increases with the number of CPU cores used. In this thesis, an alternative mechanism of process acceleration is proposed. Using one CPU core and a GPU device, the proposed solution, gcn.MOPS, achieved a speedup factor of 159× and decreased memory usage by more than half. This speedup was substantially higher than the maximum achievable speedup in cn.MOPS, which was ∼20×.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **ix**

Acknowledgements **xi**

Dedication **xii**

1 Introduction **1**

2 Background **4**

2.1 Detection of Copy-number Variations 5

2.2 cn.MOPS: a Brief Abstract 7

2.3 GPU and CUDA Programming Model 10

2.4 R and Its C/C++ Interfaces 13

2.5 Software Aspects of cn.MOPS 16

3 Execution Time of cn.MOPS **21**

4 gcn.MOPS: Accelerating cn.MOPS with GPU **28**

4.1 Staging the Code for Execution on GPU 28

4.1.1 Defragmenting the Design 29

4.1.2 Eliminating Functional R Codes 30

4.2 From cn.MOPS to Naïve gcn.MOPS 31

4.2.1 Setting up User Arguments 32

4.2.2 Input and Output Buffers 34

4.2.3 Mapping Threads to Memory Space 38

4.2.4 Partitioning Large Data Sets 42

4.3 Optimizing gcn.MOPS 45

4.3.1 Altering Code Lines 48

4.3.2 Using Constant Memory 51

4.3.3 Coalescing Memory Accesses 56

4.3.4 Changing Data Layout of Results 62

4.3.5 Eliminating Branch Divergence 63

4.3.6 Overlapping Host/Device Execution 68

4.3.7 Disabling GPU's L1-Cache Memory 69

5 Experimental Results and Performance Analysis **73**

5.1 Platform 73

5.2 The Build 74

5.3 Benchmarks 75

5.4 Methodology 77

5.5 Comparison with the Original cn.MOPS 79

6 Discussion **81**

6.1 Numeric Accuracy 81

6.2 Sensitivity Analysis 83
6.3 Multi-GPU Support 89

7 Conclusion and Future Work 91

A The Impact of Branch #1 on Coalescing Memory Requests 93

B Relevant Source Codes 96

B.1 Contents of File “Makevar” 96
B.2 R Script for Verifying gcn.MOPS Results 98
B.3 R Script for Processing 2 Data Sets on 2 GPUs Concurrently 99

Bibliography 101

List of Tables

Table 2.1 An Example of a Read-count (RC) Matrix 8

Table 2.2 an Example of Matrix $\hat{\alpha}_{ik}$ for a Single Genomic Region (GR) . . 9

Table 3.1 Execution Time of cn.MOPS Pipeline (Benchmark BM-A) . . . 21

Table 4.1 Illustration of the Grid-strided Loop for Branch #2 39

Table 4.2 Mapping Threads to Draft Spaces 41

Table 4.3 Calculating the Size of the Draft Space in Bytes 43

Table 4.4 Calculating the Size of the Argument Space in Bytes 43

Table 4.5 Calculating the Size of the Argument Space in Bytes 44

Table 4.6 The Impact of Changing `meanx` into `sumx` 49

Table 4.7 The Impact of Code Reordering (Listings 4.14-4.16) 49

Table 4.8 The Impact of Using Constant Memory 53

Table 4.9 Data Access Patterns for the Input Matrix 58

Table 4.10 Memory Bandwidth Efficiencies for BM-B 62

Table 4.11 Kernel Performance for Different Optimizations (BM-B) 65

Table 4.12 Performance Metrics of Kernel with Branch #1 Always `FALSE` . 67

Table 4.13 Performance Metrics after Disabling L1 Cache (BM-B) 70

Table 5.1 Technical Details of Used Benchmarks 76

Table 6.1 Mean Relative Difference of CPU and GPU Results 82

Table 6.2 Impact of Grid/Block Size on Kernel's Execution Time (BM-A) 87

Table A.1 Mem-store:Warp Ratios of Various Grid/Block Sizes (Listing A.3) 95

List of Figures

Figure 2.1 Illustration of CNVs 5
Figure 2.2 Short-read Alignment Against a Reference Genome 6
Figure 2.3 Counting Reads in Fixed-width Genomic Regions 7
Figure 2.4 Simplified GPU Architecture 11
Figure 2.5 Simplified Example of 32-thread Warp Execution 12
Figure 2.6 Software View of cn.MOPS's Processing Pipeline [7] 17
Figure 2.7 Program Flow of cn.MOPS in Stage 4.1 19
Figure 2.8 Output from Stage 4.1 of cn.MOPS's Pipeline 20
Figure 2.9 Output Matrices from Stage 4.2 of cn.MOPS Pipeline 20

Figure 3.1 Execution Time Breakdown of BM-A (1× CPU Core) 22
Figure 3.2 Speedup Curve for Multi-CPU Parallelism of Stage 4.1 23
Figure 3.3 Efficiency Curve for Multi-CPU Parallelism of Stage 4.1 25
Figure 3.4 The Experimentally Determined Serial Fraction of Stage 4.1 . . 26

Figure 4.1 The Program Flow of gcn.MOPS in Stage 4.1 (Phase 1) 30
Figure 4.2 The Logical Organization of the Result Buffer (Phase 2) 36
Figure 4.3 Coalesced Memory Access with Consecutive Addressing 56
Figure 4.4 Coalesced Memory Access with Random Addressing 56
Figure 4.5 Uncoalesced Memory Access with Random Addressing 57
Figure 4.6 Uncoalesced Memory Access with Strided Addressing 57
Figure 4.7 Memory Access Patterns for the Result Buffer in the Naïve Kernel 59

Figure 4.8 Visualization of Result Buffer α_ik in the Naïve Kernel 60

Figure 4.9 The Optimized Layout of the Result Buffer (Phase 3) 61

Figure 4.10 Illustration of Branch Divergence 64

Figure 4.11 % of Warps with T Diverging Threads (BM-B) 66

Figure 4.12 Progressive Speedup for all Optimization Techniques (BM-B) . 72

Figure 5.1 gcn.MOPS vs cn.MOPS: Execution Time in Stage 4 79

Figure 6.1 Processing Timeline (Stage 4.1/BM-A/`minReadCount=3525`) . . 83

Figure 6.2 Impact of $\texttt{minReadCount}_{\mathrm{R}}$ on Execution Time (Stage 4.1/BM-A) 84

Figure 6.3 Impact of $\texttt{cyc}_{\mathrm{R}}$ on Execution Time (Stage 4.1/BM-B) 85

Figure 6.4 Slowdown Factors–Increasing $\texttt{cyc}_{\mathrm{R}}$ from 20 to 200 (Stage 4.1) . 86

Figure 6.5 Available DRAM vs Execution Time (Stage 4.1/BM-A) 88

Figure 6.6 Multi-GPU Processing Timeline (Stage 4.1/2×BM-A) 89

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincerest gratitude to my supervisor, **Dr. Amirali Baniasadi**, for providing me with the opportunity to continue my graduate studies and trusting me to navigate through disciplines I was unfamiliar with. Under his supervision, I gained so many skills that positively impacted me on the personal level. The debt of gratitude I owe him could never be lower than deep.

I would also like to thank:

Dr. Günter Klambauer , for responding to my questions regarding cn.MOPS. His attention and advice helped me define the scope of this work;

Dr. Nikitas Dimopoulos , for examining my thesis and giving me invaluable notes;

Dr. Ehsan Atoofian , for providing an early feedback on my initial work;

Burair Alsaihati , for helping me understand the genetics part of this thesis;

Gloria Forbes , for her indispensable English tips;

Ahmad Lashgar , for answering my technical questions and for his lab support;

Eyad Alhakeem , for sharing his feedback on my scientific writing;

and many others who indirectly supported me in completing this work.

Thank you so much.

DEDICATION

To my children and my wife who has been my backend throughout the ups and downs of my journey. Samar was more patient with me than I was with myself. Without her, this work would have never been completed.

To my parents who always did every possible thing to set me up for success.

To my brother who inspired me to learn software development at a young age.

To Tracy Gour who unconditionally loves us.

Chapter 1

Introduction

The introduction of next-generation sequencing (NGS) technologies in 2005 has enabled scientists and researchers to sequence DNA samples at a relatively very low cost. Since 2007, the sequencing cost per genome has dropped at a significantly steeper rate than that projected by Moore's law [8]. This made DNA sequencing (DNA-seq) more accessible, which subsequently led to an explosive growth in the amount of DNA-seq data [17]. One major area of study which benefits from such growth in data is the detection of copy-number variations (CNV) from NGS data using statistical and quantitative methods. Many software tools and algorithms were pioneered [22] to tackle the problem of CNV detection and one of these tools is "Copy Number estimation by a Mixture Of PoissonS" (cn.MOPS) [7]. However, one factor that is limiting cn.MOPS, as well as other software tools in this field, is the computation time of analyzing massive NGS data sets.

According to a Sboner et al [21], breakthroughs in NGS technology has significantly decreased the proportion of the time spent on generating sequence data in a sequencing project. Around year 2000, almost 70% of a sequencing project's live-time was dedicated to the phase of sequence data generation. This percentage stood

at nearly 30% in year 2010 and the same percentage was projected by the study to be a mere 5% in year 2020. On the other hand, the phase of downstream analysis (e.g. cn.MOPS), in which knowledge is generated, constituted only 15% of a project's live-time in year 2000. This percentage increased to 35% in year 2010 and was projected to stand at around 55% in year 2020. Sboner et al concluded in their study that tools and algorithms delivered by computational biologists for the phase of downstream analyses are slow at generating enough knowledge or, in other words, processing. This phase, which can take months in a project, is extremely critical for advancements in personalized medicine and genomics.

While cn.MOPS can accelerate data processing using multiple central processing units (CPU), there is a limit to the maximum achievable speedup with this acceleration paradigm due to parallelism overhead. To achieve a speedup that is considerably higher than the maximum offered by multi-CPU parallelism, an alternative approach of acceleration is proposed. Our approach stemmed from the notion that cn.MOPS applies the same algorithm on a large amount of independent data elements. Therefore, it is well-suited for execution on a processor that implements the single-instruction-multiple-data architecture (SIMD) such as graphical processing units (GPUs). Thus, instead of executing all steps of cn.MOPS's processing pipeline using CPU, the modelling step was excluded from CPU execution and was offloaded to GPU. This step is very compute-intensive and is the core of cn.MOPS. Additionally, the impact of the memory-intensive part of the pipeline, which is the result postprocessing step, was minimized by making its execution time negligible.

In this thesis, the GPU-accelerated cn.MOPS (gcn.MOPS) is introduced. The proposed solution achieved a speedup factor of 159× in the modelling step. This achieved speedup is substantially higher than the maximum achievable speedup using cn.MOPS, which was found to be ∼20×. Moreover, gcn.MOPS minimized the

execution time of the result postprocessing step by 97%. Lastly, gcn.MOPS had less than half the memory footprint of cn.MOPS in the modeling and the postprocessing steps combined. The rest of this thesis is organized as follows. In Chapter 2, a relevant domain background is presented and is specifically focused on DNA copy-number variations, CUDA Programming Model & GPUs, and the R programming language. The software organization of cn.MOPS is also presented. In Chapter 3, the execution time of cn.MOPS's pipeline is analyzed to determine bottlenecks. Then, the steps taken to restructure cn.MOPS into gcn.MOPS are detailed in Chapter 4. Finally, experimental results of achieved speedups are presented in Chapter 5 and are discussed in Chapter 6.

Chapter 2

Background

The scope of this thesis is concerned with how GPU was used to accelerate the processing pipeline of cn.MOPS. In order to comprehend how data processing was accelerated in gcn.MOPS, a background in both the field and the involved tools is presented. While the concept of detecting copy-number variations is not very crucial to understand this work, it is introduced in the next section to help the reader conceive the broader impact of both cn.MOPS and gcn.MOPS. Additionally, cn.MOPS is an R package and its core algorithm was written in C/C++; thus, a brief background about the R environment as well as its C/C++ interface is given. In § 2.3, the CUDA programming model is concisely explained since gcn.MOPS utilizes CUDA to accelerate processing. Finally, the software organization and other technical aspects of cn.MOPS are presented to help the reader understand how cn.MOPS was restructured into gcn.MOPS. To avoid ambiguity and if applicable, R and C/C++ symbol names will be proceeded with a small character which indicates the relevant language, e.g. `function(..)`$_{\mathrm{R}}$ and `function(..)`$_{\mathrm{C}}$.

2.1 Detection of Copy-number Variations

Numerous studies showed that Copy-number variations (CNVs) play a major role in complex diseases as well as neurological and developmental disorders. For example, Gonzalez et al showed that CNVs in the *CCL3L1* gene play a role in the susceptibility to the human immunodeficiency virus-1 (HIV-1) [5]. In another study, Weiss et al showed that CNV at 16p11.2[1] of size 593 kb[2] results in a greater risk of autism [24]. In addition, CNVs in the *SNCA* and *PARK2* genes are known to cause Mendelian forms of Parkinson disease [18]. These studies among many others have drawn scientists' attention to detecting CNVs as a means to evaluate drug response and treat complex diseases [25].

CNV is a form of genetic structural variation (SV) in which a large segement of DNA ($>$ 1000 bases) is altered [23]. This alteration may take a form of duplication (gain/amplification) or deletion (loss) relative to a reference genome [4]. In other words, a segment of DNA could be duplicated multiple times consecutively or missing as a whole block [27]. Figure 2.1 shows examples of CNVs; these examples are for illustration only and are not realistic.

Reference Sequence: ACCAGGATTTCGAAAAACCGGACTCCTTTATCGH

CNV duplication (CN2): ACCAGGATTTCGAAAAACCGGAAACCGGACTCCTTTATCGH
CNV duplication (CN3): ACCAGGATTTCGAAAAACCGGAAACCGGAAACCGGACTCCTTTATCGH
CNV deletion (CN0): ACCAGGATTTCGAAACTCCTTTATCGH

Figure 2.1: Illustration of CNVs

CNVs can be detected either by using array-based technologies or by statistically analyzing NGS data. The latter technique has the advantage of higher resolution as it has the potential to detect many forms of SVs including single-nucleotide poly-

[1] Chromosome 16, location p11.2
[2] Kilo-bases: a base is the character representation of an amino acid i.e. A, T, C, G.

morphism (SNP) at the base pair level [4]. In comparison, the resolution of array-based technologies, such as array-Comparative Genomic Hybridization (array-CGH), is greater than 10 kb [26] and, thus, they cannot detect CNVs in the size range [1 kb, 10 kb). This advantage as well as others mentioned in [7] makes the analysis of NGS data an important method for detecting CNVs. There exist many quantitative techniques and algorithms to detect CNVs in NGS data [22], but this thesis is only focused on cn.MOPS. To help the reader form a complete picture of this work, a basic idea about the general concept of statistically analyzing NGS data to detect CNVs is firstly given.

In a typical DNA-seq experiment, data is generated by one of the DNA-seq technologies like Illumina NextSeq System series. At this stage, a DNA sample is sheared into fragments and these fragments are then amplified (ie replicated). Next, the sequencing machine generates a large amount of data called *short reads*. These short reads are the digital representation of the amino-acid sequences of the DNA fragments. Finally, computer tools are used to assemble the DNA sequence of the sample by aligning the short reads to a reference genome as shown in Figure 2.2 [6].

```
A C G G T T C G A A A T T A C G T T G C A T C C G T A G G C    Ref. Genome

A C G G   C G A A     A C G T       T C C  T A G G
  C G G T   G A A A  T A C G          C C G T          Short-read
    G T T     A A T T    G T T     T C C G T           Alignment
      T C G A  A T T A                C G T A

A C G G T T C G A A A T T A C G T T - - - T C C G T A G G -  Sample Sequence
```

Figure 2.2: Short-read Alignment Against a Reference Genome

Besides the assembled DNA sequence, the short-read alignment results can be used to calculate read counts (RCs), which are beneficial to CNV analysis. RC is defined

as the number of reads that are mapped to a specific range/region in the reference genome. Figure 2.3 shows a simple example of how RCs are calculated for genomic regions of size 3 bases. RC per genomic region can be analyzed to detect CNVs because, in DNA-seq experiment data, RC is proportional to the number of times that region appears in a given DNA sample [9]. Thus, if RC is below an established threshold for a specific genomic region, then there could be a CNV (loss) in that region. In other words, this means that there might be fewer copies of that region in the given DNA sample than there are in the reference genome. Similarly, if RC is above an established threshold for another genomic region, then there could be a CNV (gain).

A C G	G T T	C G A	A A T	T A C	G T T	G C A	T C C	G T A	G G C	**Ref. Genome**
A C G	G	C G A	A	A C	G T		T C C	T A	G G	**Short-read Alignment**
C G	G T	G A	A A	T A C	G		C C	G T		
	G T T		A A T	T	G T T		T C C	G T		
	T	C G A	A T	T A			C	G T A		
2	4	3	4	4	3	0	4	4	1	**Read Counts**

Figure 2.3: Counting Reads in Fixed-width Genomic Regions

2.2 cn.MOPS: a Brief Abstract

The basic idea behind cn.MOPS is that a CNV is detected in a given chromosomal segment for a given DNA-seq sample if the following conditions hold:

1. variation in RCs is detected across samples for a given chromosomal segment;
2. variation in RCs is detected across chromosomal segments for a given sample.

Analysis for the second condition happens in the segmentation step, which is beyond

the scope of this thesis. Meanwhile, analysis for the first condition takes place in the modelling step, which is the focus of this thesis, and, hence, it is briefly explained.

Analysis for the second condition begins by forming a read-count matrix of multiple DNA-seq samples as shown in Table 2.1. Values for each sample are obtained in a roughly similar fashion to that shown in Figure 2.3. For each row in the RC matrix,

Genomic	RCs per DNA-seq Sample			
Region #	Sample 1	Sample 2	Sample 3	Sample k
1	1625	15	1670	...
2	1391	1379	1383	...
3	935	901	921	...
4	1457	1438	102	...
5	250	261	1709	...
6	35	1802	1790	...
...	...	...	...	...

Table 2.1: An Example of a Read-count (RC) Matrix

cn.MOPS computes multiple values as follows:

- *"information gain of posterior over prior"* (I/NI): an indicator for the existence of a CNV in the genomic region (GR) across all samples. This value is calculated as shown in Equation 2.1.

$$\mathrm{I/NI} = \frac{1}{N} \sum_{k=1}^{N} \sum_{i=0}^{n} \hat{\alpha}_{ik} \times |\log(i/2)| \tag{2.1}$$

- The posterior matrix ($\hat{\boldsymbol{\alpha}}$): the probability distribution of RCs x_k for samples $1..N$ corresponding to copy number (CN) i. Matrix values are calculated as shown in Equation 2.2, where: P is the probability density of Poisson distribution; and the denominator is a conditional probability. Table 2.2 shows and example of matrix $\hat{\alpha}_{ik}$ where the sum of each probability-distribution column

$= 1$.

$$\hat{\alpha}_{ik} = \frac{\alpha_i^{old} \times P(x_k; \frac{i}{2}\lambda^{old})}{p(x_k; \alpha^{old}, \lambda^{old})} \tag{2.2}$$

		Probability Distributions			
		Sample 1	Sample 2	Sample 3	Sample N
Copy Number	CN0	1.77812026e-99	2.07524141e-97	1.74556206e-99	...
	CN1	6.21988278e-06	1.90708336e-31	2.78450008e-06	...
	CN2	9.99993776e-01	2.64784252e-09	9.99997206e-01	...
	CN3	2.53419292e-37	5.41800306e-36	4.05521981e-37	...
	CN4	4.37318414e-09	9.99999997e-01	9.76867608e-09	...
	CN5	3.55394735e-47	2.30242978e-33	1.02828518e-46	...
	CN6	1.77812026e-99	2.07524141e-97	1.74556206e-99	...
	CN7	1.77812026e-99	2.07524141e-97	1.74556206e-99	...
	CN8	1.77812026e-99	2.07524141e-97	1.74556206e-99	...

Table 2.2: an Example of Matrix $\hat{\alpha}_{ik}$ for a Single Genomic Region (GR)

- Vector $\boldsymbol{\alpha}^{new}$: a model parameter which is calculated using Equation 2.3, where: initial values α_i^{old} are set to 0.05 except α_2^{old} which is set to 0.6; γ_i is related to the Drichet prior on α_i; and $\gamma_s = \sum_{i=0}^{n} \gamma_i$.

$$\alpha_i^{new} = \frac{\frac{1}{N}\sum_{k=1}^{N} \hat{\alpha}_{ik} + \frac{1}{N}(\gamma_i - 1)}{1 + \frac{1}{N}(\gamma_s - n)} \tag{2.3}$$

- λ^{new}: the expected read count for CN2 which is calculated using Equation 2.4.

$$\lambda^{new} = \frac{\frac{1}{N}\sum_{k=1}^{N} x_k}{\sum_{i=0}^{n}\left(\frac{i}{2N}\sum_{k=1}^{N} \hat{\alpha}_{ik}\right)} \tag{2.4}$$

- *"The signed individual I/NI"* (**sI/NI**): measures the contribution of each sample to the I/NI call and whether that contribution is a gain or a loss. This vector is used by the segmentation algorithm to join consecutive, I/NI-calling, GRs

into a CNV chromosomal segment. Values in this vector are calculated using Equation 2.5, which is similar to the inner summation of Equation 2.1 without taking the absolute value of the log function. Alternatively, **sI/NI** can be seen as the result of multiplying a vector of constants, $\tilde{\boldsymbol{I}}$, with a matrix similar to that shown in Table 2.2.

$$\text{sI/NI} = \sum_{i=0}^{n} \hat{\alpha}_{ik} \times \log(i/2) \tag{2.5}$$

- The expected copy number (**CN**): each component in this vector corresponds to a sample such that components are row-labels of the maximum value in each column of $\hat{\boldsymbol{\alpha}}$. For instance, **CN** of Table 2.2 is $\{CN2, CN4, CN2\}$.

Once all these values are calculated, **sI/NI** is passed to the segmentation step for further analysis.

2.3 GPU and CUDA Programming Model

GPU, as the name suggests, is known for its powerful graphic processing. Because it has so many cores, it is able to simultaneously process a lot of pixels. Many problems share the same properties of graphic processing. That is, solving the problem requires applying the same instructions on multiple data elements. This makes GPU the ideal processor for said problems since it is based on the SIMD architecture. It must be noted that nVidia GPUs[3] are based on the single-instruction-multiple-threads (SIMT) architecture. Execution resources, or simply CUDA cores, are organized in many stream-multiprocessors (SM) as shown in Figure 2.4. SIMT and SIMD are conceptually similar except that in SIMT, multiple and lightweight threads occupy

[3] For simplicity, subsequent mentionings of "GPU" would mean "nVidia GPU."

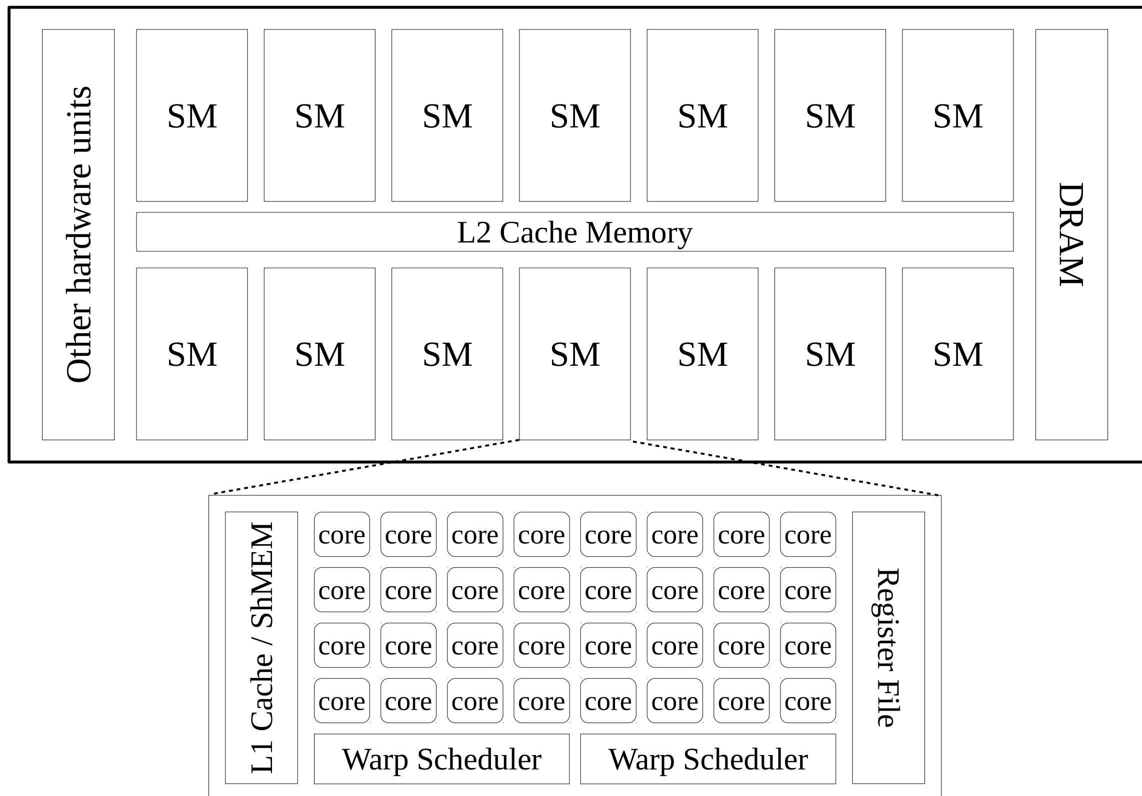


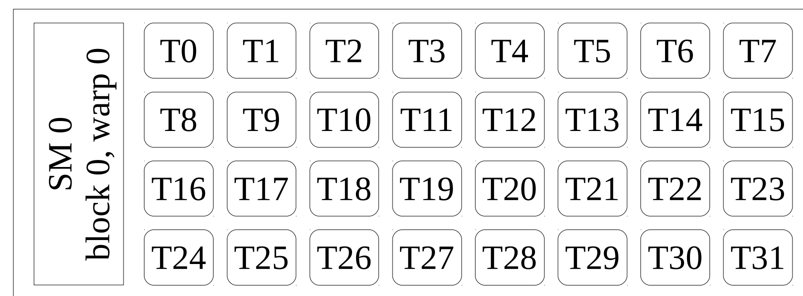
Figure 2.4: Simplified GPU Architecture

the same GPU resources concurrently to maximize hardware utilization. Unlike CPU threads, GPU threads are grouped in what is called *warps*. In all nVidia's GPU architectures released [2][14], a warp is a group of 32 threads which execute the same instruction on different data elements in a lock-step fashions. Consequently, if a memory transaction is issued and some threads are not served due to cache miss, the other threads in this warp are forced to wait until all threads are ready to proceed. This is very inefficient as processing resources would be underutilized. Thus, the warp scheduler stalls this warp and schedules another one that is ready for execution. Like CPU threads can be created with OpenMP (Open Multi-Processing), GPU threads can be created with CUDA.

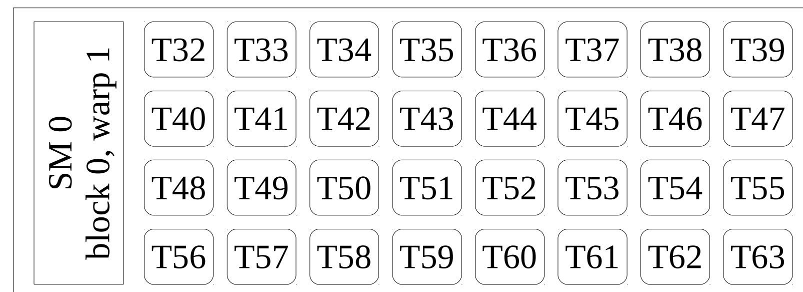
CUDA is a programming environment which was invented by nVidia Corporation [12]. It extends C/C++ with directives and keywords that enable the use of supported

GPU devices for general-purpose computing. It also provides a set of application programmer interfaces (APIs) to manage GPU devices, GPU memory, host-device synchronization etc. Further, nVidia provides a proprietary compiler called *nvcc* which processes CUDA sources, by separating GPU codes from host codes (ie CPU), and compiles them.

In order to harness the power of GPU using CUDA, the problem must fit the SIMT/SIMD architecture. CUDA provides the means to map threads to different



(a) time = t_0 , instruction X , data[0..31]



(b) time = t_{0+c} , instruction X , data[32..63]

Figure 2.5: Simplified Example of 32-thread Warp Execution

data elements. From the programmer perspective, threads are organized in blocks and each thread can be identified using its local ID and its block ID. At hardware level, blocks are assigned to available SMs until all instructions are executed. These blocks are executed warp by warp as illustrated in Figure 2.5. All threads execute the same program which is called the *kernel*. Kernels are launched and configured from host while data and results are copied to and from device using CUDA APIs such as

`cudaMemcpy(..)`$_{\mathrm{C}}$.

2.4 R and Its C/C++ Interfaces

R is a high-level, multi-paradigm, interpreted programming language and environment for statistical computing. It offers various statistical methods such as linear regression and time-series analysis. It also facilitates generating publication-quality charts and plots with an ease. Further, R provides the means to easily run a function in parallel while abstracting the underlying infrastructure [20]. R is very versatile and it can be extended with user-defined packages. An obvious example of an R package is cn.MOPS. It might be tempting to think that an R package is written purely in R, but that is not the case. In fact, an R package may contain compiled codes written in C/C++ or other languages.

There are three native interfaces for calling a C/C++ function from within an R script: `.C`$_{\mathrm{R}}$, `.Call`$_{\mathrm{R}}$, or `.External`$_{\mathrm{R}}$. The first method requires the arguments of the C/C++ function to be passed by reference (ie pointers). Furthermore, the function does not explicitly `return`$_{\mathrm{C}}$ any value and it is expected to write results in the arguments themselves. Listing 2.1 shows an example of a C function, which multiplies two integers, that is intended to be invoked via interface `.C`$_{\mathrm{R}}$. The source is then compiled into a shared object (.so file) as shown in Listing 2.2; line 1 is the compilation command while lines 2 and 3 are the compiler output. In Listing 2.3, `mul(..)`$_{\mathrm{C}}$ is invoked via interface `.C`$_{\mathrm{R}}$. The following is a line-by-line explanation of Listing 2.3:

- line 1: starts an R session in the shell command;
- line 2: loads the shared object dynamically in the current R session;

- line 3: invokes mul(..)$_{C}$ by its symbol name and specifies the arguments. Values of the arguments, after invocation, are stored in m$_{R}$;
- line 4: shows the value of ans$_{R}$ that is stored in m$_{R}$ (line 5).

Listing 2.1: mul(..)$_{C}$ is to be Invoked via Interface .C$_{R}$

```
1 /* function is defined in ‘func.c’ */
2 void mul(int *a, int *b, int *res) { (*res) = (*a) * (*b); }
```

Listing 2.2: Steps to Compile a C Function into Shared Object

```
1 (shell)$ R CMD SHLIB func.c
2 gcc -std=gnu99 -I/usr/share/R/include -DNDEBUG -fpic -g -O2 -fstack-protector-strong
      -Wformat -Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -g -c func.c -o
      func.o
3 gcc -std=gnu99 -shared -L/usr/lib/R/lib -Wl,-Bsymbolic-functions -Wl,-z,relro -o func
      .so func.o -L/usr/lib/R/lib -lR
```

Listing 2.3: Calling mul(..)$_{C}$ via Interface .C$_{R}$

```
1 (shell)$ R
2 > dyn.load("~/path/to/shared/object/func.so")
3 > m <- .C("mul", n1=as.integer(10), n2=as.integer(5), ans=as.integer(ans))
4 > m$ans
5 [1] 50
```

The second method (.Call$_{R}$) requires the C/C++ function to receive arguments and return a single result object that are of type SEXP$_{C}$[4]. This datatype is an R's internal data structure that encompasses a data pointer and a header. Depending on the type of data, R's internal macros such as INTEGER(..)$_{C}$, REAL(..)$_{C}$, and LOGICAL(..)$_{C}$ are used to access the data pointer within a SEXP$_{C}$ object. Attributes of a SEXP$_{C}$ object like matrix dimensions or vector length are stored in the header and can be accessed using macros like GET_DIM(..)$_{C}$ and length(..)$_{C}$. In addition, special memory allocators such as Calloc(..)$_{C}$, allocVector(..)$_{C}$, and allocMatrix(..)$_{C}$ are used to dynamically allocate memory. Listing 2.4 shows a C

[4]Simple EXPression.

function similar to that shown in Listing 2.1 except that it is intended to be invoked via interface `.Call`$_R$. Arguments passed from R to C/C++ are individually packed in `SEXP`$_C$ structures without intervention form developer. The following is a line-by-line explanation of Listing 2.4:

- line 2: includes the header containing R's internal macros;
- line 4: gets pointers to the data portion of the `SEXP`$_C$ arguments and typecasts them to integer pointers using macro `INTEGER(..)`$_C$;
- line 7: allocates memory for a vector of length 1 whose data type is integer. `ret`$_C$ is protected from garbage collection.
- line 8: gets a pointer to `ret`$_C$'s data and typecasts it to integer pointer;
- line 9: performs the multiplication;
- line 11: allows garbage collection of the last protected n item(s).

Listing 2.4: `mul(..)`$_C$ is to be Invoked via Interface `.Call`$_R$

```
1  /* function is defined in 'func.c' */
2  #include <Rdefines.h>
3  SEXP mul(SEXP a, SEXP b) {
4    int *a_data = INTEGER(a), *b_data = INTEGER(b);
5
6    SEXP ret;
7    PROTECT( ret = allocVector(INTSXP, 1) );
8    int *ret_data = INTEGER(ret);
9    (*ret_data) = (*a_data) * (*b_data);
10
11   UNPROTECT(1);
12   return ret;
13 }
```

The source is then compiled into a shared object (.so file) as shown in Listing 2.2. In Listing 2.5, `mul(..)`$_C$ is invoked via interface `.Call`$_R$. In line 3, the returned value is an R object, which is internally a `SEXP`$_C$ object. Thus, `m`$_R$ is now accessible as if it was created in R.

Listing 2.5: Calling `mul(..)`$_C$ via interface `.Call`$_R$

```
(shell)$ R
> dyn.load("func.so")
> m <- .Call("mul", n1=as.integer(10), n2=as.integer(5))
> m
[1] 50
```

The last method (`.External`$_R$) is similar to the second method. The only difference is that interface `.External`$_R$ packs all arguments in a single `SEXP`$_C$ object. Then, the C/C++ function uses appropriate macros to extract all arguments from the `SEXP`$_C$ structure.

2.5 Software Aspects of cn.MOPS

As stated earlier, cn.MOPS is a processing pipeline. Figure 2.6 shows the stages in which input is processed before output is produced. Normally, processing starts at stage 2 of the pipeline provided the input is a group of BAM files (Binary Alignment/Map). In stage 2, RCs per DNA-seq sample for reference sequences are extracted and organized in matrix `X`$_R$. An example of `X`$_R$ was shown in Table 2.1. Then, `X`$_R$ is normalized in stage 3 and a new matrix `X.norm`$_R$ is generated and passed as input to stage 4.1. In R, matrix data is stored in a column-major fashion and, thus, data for each sample is stored contiguously in memory. Listing 2.6 shows an R script that runs cn.MOPS's processing pipeline for a list of 15 BAM files stored in a folder. The following is an explanation of the script:

- `line 1:` loads cn.MOPS package in current R session;
- `line 2:` gets the full path of all ".bam" files in the given path;
- `line 4-6:` creates `GRanges`$_R$ object which stores RCs of given ".bam" files;
- `line 8-13:` starts stages 4, 5, and 6 in Figure 2.6;

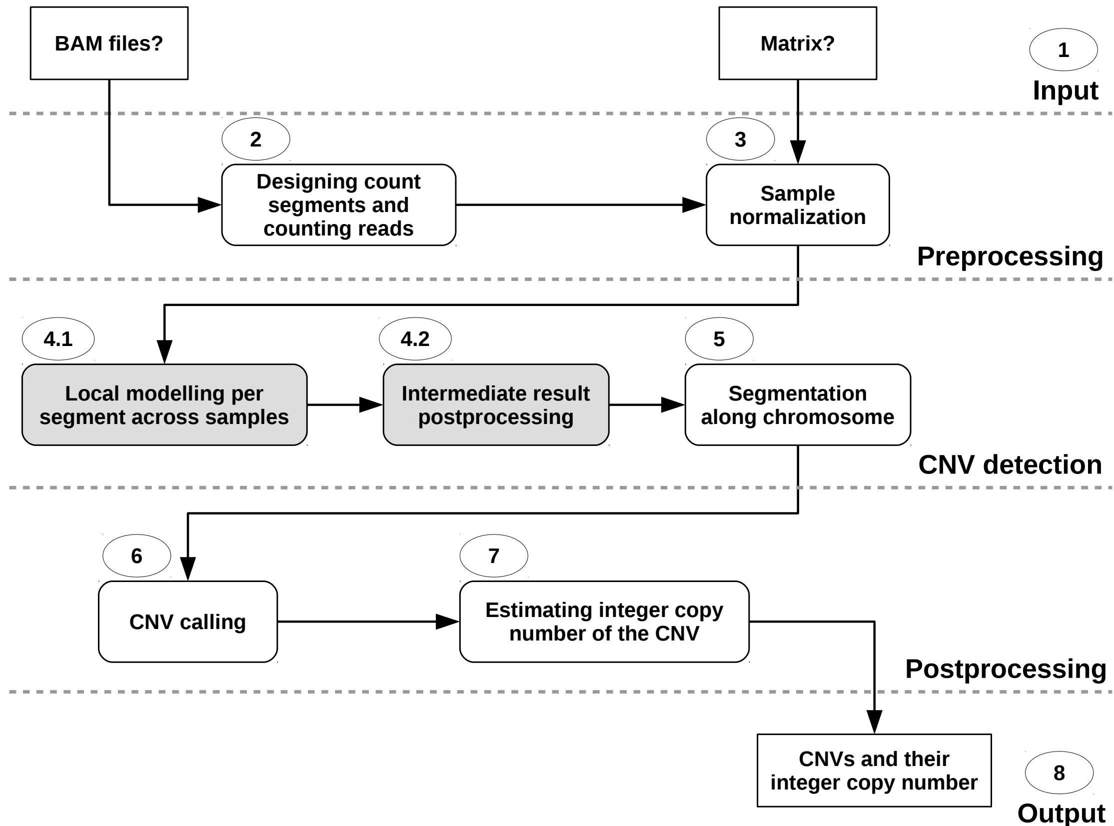


Figure 2.6: Software View of cn.MOPS's Processing Pipeline [7]

- `line 15:` starts stage 7, 8, and 9 in Figure 2.6.

For the most part, this work is focused on accelerating stages 4, which is highlighted in Figure 2.6. Hence, technical details about these stages are presented.

In stage 4.1, `X.norm`$_{\mathrm{R}}$ is processed by calling `.cn.mopsCE(..)`$_{\mathrm{R}}$ as many times as there are rows. This function is one of other internal functions and, essentially, is cn.MOPS's core algorithm. It is also the wrapper for cn.MOPS's core mathematical model, which is written in C/C++. This software organization is illustrated in Figure 2.7. Branch #1 is evaluated at R and if it is not taken, the core mathematical model, `cnmops(..)`$_{\mathrm{C}}$, is invoked via interface `.Call`$_{\mathrm{R}}$. Branch #2 is inherently a loop and it is implemented using `apply(..)`$_{\mathrm{R}}$ as shown in Listing 2.7. Line 1 instructs R to iterate through the rows of `X.norm`$_{\mathrm{R}}$; in each iteration, the function in line 2 is invoked

Listing 2.6: An R Script for Running cn.MOPS

```
1  library(cn.mops)
2
3  # if data is not saved on disk
4  BAMFiles <- list.files(path="/path/to/samples",pattern=".bam$", full.names=TRUE)
5  bamDataRanges <- getReadCountsFromBAM(
6      BAMFiles[1:15], sampleNames=paste("Sample",1:15),
7      refSeqName= c("1", "2","3"), WL=100, mode="paired", parallel=12)
8
9  # if data is saved on disk:
10 #load("bamDataRanges.RData")
11
12 res <- cn.mops(input=bamDataRanges, I=c(0.025, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4),
13     classes=c("CN0", "CN1", "CN2", "CN3", "CN4", "CN5", "CN6", "CN7", "CN8"),
14     priorImpact=10, cyc=20, parallel=12, norm=1, normType="poisson",
15     sizeFactor="mean", normQu=0.25, quSizeFactor=0.75, upperThreshold=0.5,
16     lowerThreshold=-0.8, minWidth=5, segAlgorithm="fast", minReadCount=1,
17     useMedian=FALSE, returnPosterior=FALSE)
18
19 res <- calcIntegerCopyNumbers(res)
```

and the row is passed to the function as input data; arguments, other than input data, for the invoked function are in line 3. Returned values from all invocations are then grouped together in a list and this list is stored in intermediate variable $\texttt{res}_{\text{R}}$ as shown in Figure 2.8. Each returned value in the list is itself a list of: two numeric vectors of length n (λ and α); a numeric vector of length N ($sini$); a vector of N strings (CN), a numeric value (ini), and optionally a numeric matrix of size $N \times n$ (α_ik); where N is the number of samples and n is the number of classes as shown in lines 6 and 13 of Listing 2.6 respectively. These symbols and their meaning were introduced in § 2.2. It must be noted that in Figure 2.8, each vector is allocated independently and, hence, vectors are not contiguous in memory. For instance, λ for genomic region (GR) 0 is not contiguous with λ for GR 1. That is also the case with λ and α for GR 0 and other GRs. Colors only show the logical organization of the result object returned from stage 4.1.

Next, list $\texttt{res}_{\text{R}}$ is passed as an input to stage 4.2. Strictly speaking, stage 4.2 is not part of cn.MOPS's logical pipeline. Though, it is shown in Figure 2.6 as a separate stage because it is single-threaded and very time-consuming[5]. In this

[5] in [7], stages 4.1 and 4.2 are simply called "local modelling per segment across samples."

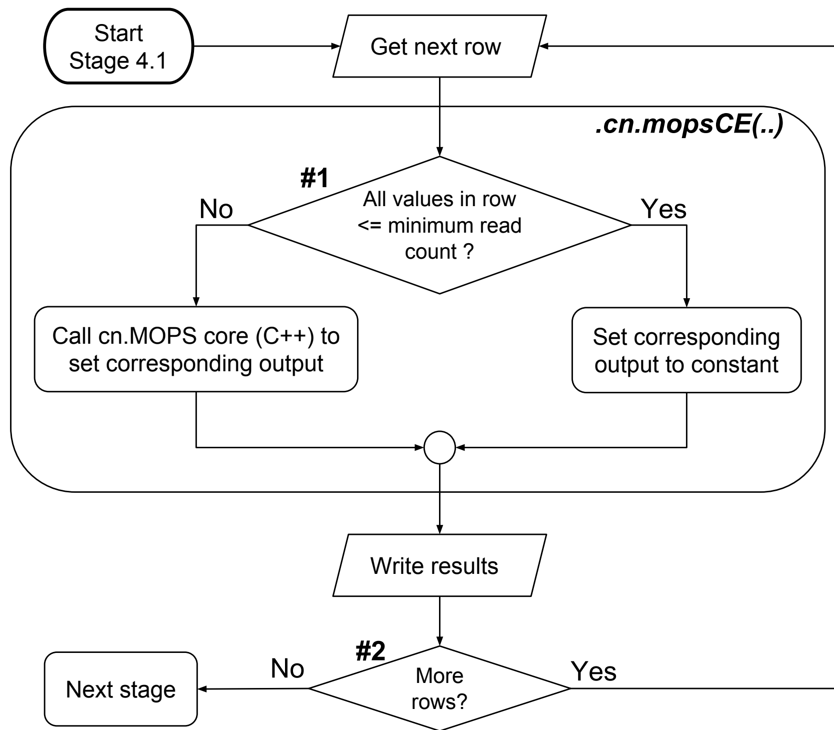


Figure 2.7: Program Flow of cn.MOPS in Stage 4.1

Listing 2.7: The Implementation of Branch #2 Using `apply(..)`$_{\mathrm{R}}$

```
1 resChr <-apply(X.norm, 1,   # 1 indicates rows and 2 indicates columns
2     .cn.mopsCE,
3     arg1, arg2, (...), minReadCount, (...), argN) # minReadCount is user arg
```

memory-intensive stage, four matrices, one vector, and one 3D-matrix are constructed from list `res`$_{\mathrm{R}}$. This is done by simply concatenating results for all GRs per symbol. Figure 2.9 shows the four output matrices after concatenating relevant vectors; vector ini and 3D-matrix α_ik are omitted. As previously stated, matrices in R are stored in a column-major fashion. Thus, each matrix shown in Figure 2.9 is stored contiguously in memory. After stage 4.2, the constructed vector and matrices are passed to the next stage and list `res`$_{\mathrm{R}}$ is deleted from memory.

res

genomic region [0]						
	λ	α	CN	sini	ini	α_ik
λ	[0]	[1]	[2]	[3]	...	[n]
α	[0]	[1]	[2]	[3]	...	[n]
sini	[0]	[1]	[2]	[3]	...	[N]
CN	[0]	[1]	[2]	[3]	...	[N]
ini	c					
α_ik	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]
	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]
	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]
	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]
	:	:	:	:	⋱	:
	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]

genomic region [1]						
	λ	α	CN	sini	ini	α_ik
λ	[0]	[1]	[2]	[3]	...	[n]
α	[0]	[1]	[2]	[3]	...	[n]
sini	[0]	[1]	[2]	[3]	...	[N]
CN	[0]	[1]	[2]	[3]	...	[N]
ini	c					
α_ik	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]
	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]
	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]
	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]
	:	:	:	:	⋱	:
	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]

genomic region [2]						
	λ	α	CN	sini	ini	α_ik
λ	[0]	[1]	[2]	[3]	...	[n]
α	[0]	[1]	[2]	[3]	...	[n]
sini	[0]	[1]	[2]	[3]	...	[N]
CN	[0]	[1]	[2]	[3]	...	[N]
ini	c					
α_ik	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]
	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]
	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]
	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]
	:	:	:	:	⋱	:
	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]

genomic region [R]						
	λ	α	CN	sini	ini	α_ik
λ	[0]	[1]	[2]	[3]	...	[n]
α	[0]	[1]	[2]	[3]	...	[n]
sini	[0]	[1]	[2]	[3]	...	[N]
CN	[0]	[1]	[2]	[3]	...	[N]
ini	c					
α_ik	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]
	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]
	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]
	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]
	:	:	:	:	⋱	:
	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]

Figure 2.8: Output from Stage 4.1 of cn.MOPS's Pipeline

λ matrix	CN1	CN2	CN3	CN4	...	CN_n
GR [0]	λ[0]	λ[1]	λ[2]	λ[3]	...	λ[n]
GR [1]	λ[0]	λ[1]	λ[2]	λ[3]	...	λ[n]
GR [2]	λ[0]	λ[1]	λ[2]	λ[3]	...	λ[n]
GR [R]	λ[0]	λ[1]	λ[2]	λ[3]	...	λ[n]

sini matrix	S1	S2	S3	S4	...	S_N
GR [0]	sini[0]	sini[1]	sini[2]	sini[3]	...	sini[N]
GR [1]	sini[0]	sini[1]	sini[2]	sini[3]	...	sini[N]
GR [2]	sini[0]	sini[1]	sini[2]	sini[3]	...	sini[N]
GR [R]	sini[0]	sini[1]	sini[2]	sini[3]	...	sini[N]

α matrix	CN1	CN2	CN3	CN4	...	CN_n
GR [0]	α[0]	α[1]	α[2]	α[3]	...	α[n]
GR [1]	α[0]	α[1]	α[2]	α[3]	...	α[n]
GR [2]	α[0]	α[1]	α[2]	α[3]	...	α[n]
GR [R]	α[0]	α[1]	α[2]	α[3]	...	α[n]

CN matrix	S1	S2	S3	S4	...	S_N
GR [0]	CN[0]	CN[1]	CN[2]	CN[3]	...	CN[N]
GR [1]	CN[0]	CN[1]	CN[2]	CN[3]	...	CN[N]
GR [2]	CN[0]	CN[1]	CN[2]	CN[3]	...	CN[N]
GR [R]	CN[0]	CN[1]	CN[2]	CN[3]	...	CN[N]

Figure 2.9: Output Matrices from Stage 4.2 of cn.MOPS Pipeline

Chapter 3

Execution Time of cn.MOPS

In order to demonstrate how execution time is limiting cn.MOPS, a sufficiently large, yet relatively small, benchmark was run and profiled. Details about this benchmark which was called BM-A, as well as the used platform, are elaborated in chapter 5. Table 3.1 presents the execution time of each pipeline stage using a single CPU core while Figure 3.1 shows a visual representation of Table 3.1 as a percentage of total. Apparently, stage 4, i.e. stages 4.1 and 4.2 combined, is the bottleneck.

Pipeline Stage	**Time (min)**
(Figure 2.6)	×1 CPU core
#2 counting reads	16.5
#3 normalization	9.1
#4.1 modelling	57.3
#4.2 postprocessing	8.3
#5 segmentation	13.7
#6 CNV Calling	0.1
#7 est. integer CNV	6.7
all stages (total)	111.7

Table 3.1: Execution Time of cn.MOPS Pipeline (Benchmark BM-A)

There were many factors that affected the execution time of the bottleneck stage,

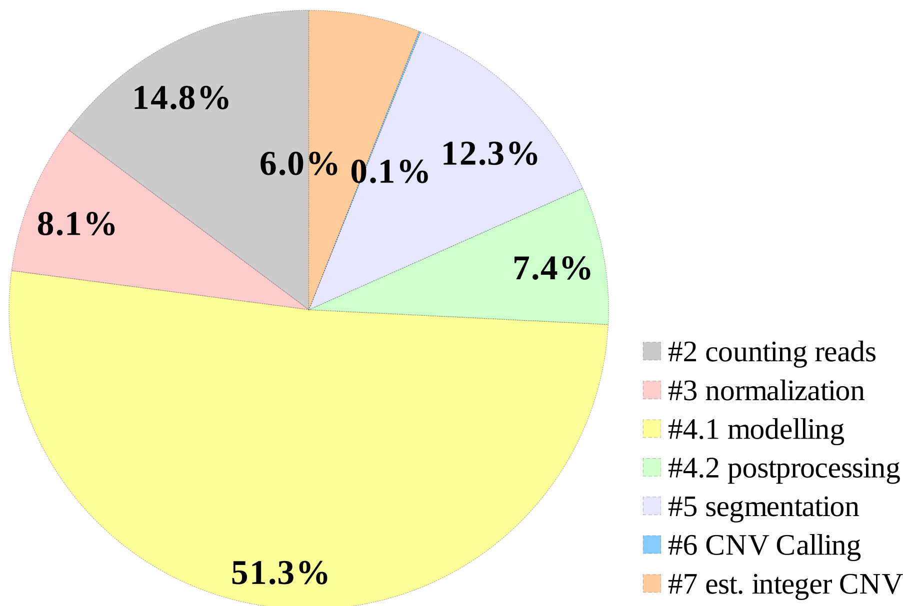


Figure 3.1: Execution Time Breakdown of BM-A (1× CPU Core)

but the most important one was the number of genomic regions/ranges (GRs) and samples in the experiment (Table 2.1). Perhaps, the original cn.MOPS package provides the means for parallel processing which can reduce the execution time of stage 4.1. As shown in Listing 2.6, this can be done by specifying argument `parallel=x` in line 14, where `x` is the number of CPU cores to be used. It should be noted that there is no implemented parallelism for stages 3, 6, and 7.

However, multi-CPU parallelism for stage 4.1 would be limited by Amdahl's Law, shown in Equation 3.1 [19]. That is, the maximum speedup ψ is bound by serial fraction f of program as shown in Equation 3.2, where p is the number processors.

$$\psi \leq \frac{1}{f + \frac{1-f}{p}} \tag{3.1}$$

$$\lim_{p \to \infty} \psi \leq \frac{1}{f} \tag{3.2}$$

Despite that stage 4.1 was embarrassingly parallel (i.e. $f = 0$), using more CPU

cores increased the overhead due to parallelism ($\kappa(p)$). Thus, with parallel execution, the execution time T would involve component $\kappa(p)$, which is usually an increasing function, as shown in Equation 3.3; where T_φ is the time taken in the parallel fraction and $T_f = 0$.

$$T(p) = T_f + \frac{T_\varphi}{p} + \kappa(p) \tag{3.3}$$

Therefore, Equation 3.1 would be rewritten as shown in Equation 3.4, where $T_f = 0$.

$$\psi \leq \frac{T_\varphi}{\frac{T_\varphi}{p} + \kappa(p)} \tag{3.4}$$

Experimentally, stage 4.1 executed in accordance with Amdahl's Law or, more precisely, Equation 3.3. This was demonstrated by measuring speedup, with respect to the execution time of one CPU core, versus the number of used CPU cores as shown in Figure 3.2. Speedup values were obtained using Equation 3.5.

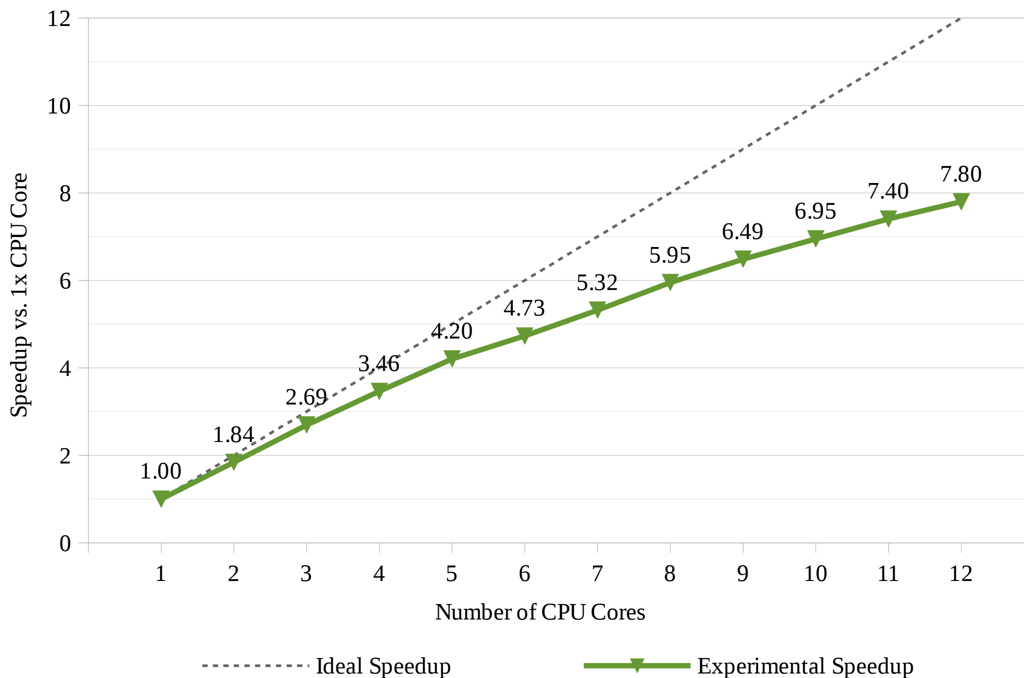


Figure 3.2: Speedup Curve for Multi-CPU Parallelism of Stage 4.1

$$\psi(p) = \frac{T(1)}{T(p)} \tag{3.5}$$

Evidently, the increasing gap between ideal speedup and experimental speedup indicated that $\kappa(p)$ was increasing with the number of CPU cores used. Therefore, there existed a theoretical limit for the maximum achievable speedup with multi-CPU parallelism, ψ_{max}^{CPU}, which was due to parallelism overhead $\kappa(p)$.

To determine ψ_{max}^{CPU}, the performance was modelled as a polynomial by extrapolating data from Figure 3.2 using LibreOffice Calc. The obtained speedup formula, $g(p)$, is shown in Equation 3.6.

$$g(p) = -0.022695p^2 + 0.908861p + 0.141837 \tag{3.6}$$

Next, the global maxima for $g(p)$ was found by deriving $g(p)$ and solving the derivative for p such that $g'(p)$=0. This yielded p=20, which meant that the projected maximum speedup would be achieved using 20× CPU cores. Substituting p=20 in Equation 3.6 yielded $g(20)$=9.24, which is ψ_{max}^{CPU}.

Though, using 20× CPU cores to achieve $\psi_{max}^{CPU}(20) \lesssim 9.24\times$ is very inefficient. In this scenario, the efficiency, which is calculated as per Equation 3.7, would be ∼39.3%.

$$\text{Efficiency}(p) = \frac{\psi(p)}{p} \tag{3.7}$$

Experimentally, the projected efficiency for 20× CPU cores is also similar. A plot of experimental efficiency versus the number of used CPU cores is shown in Figure 3.3. Assuming linearity, a line function that estimated efficiency was formulated by extrapolating data from Figure 3.3 using LibreOffice Calc. The obtained efficiency

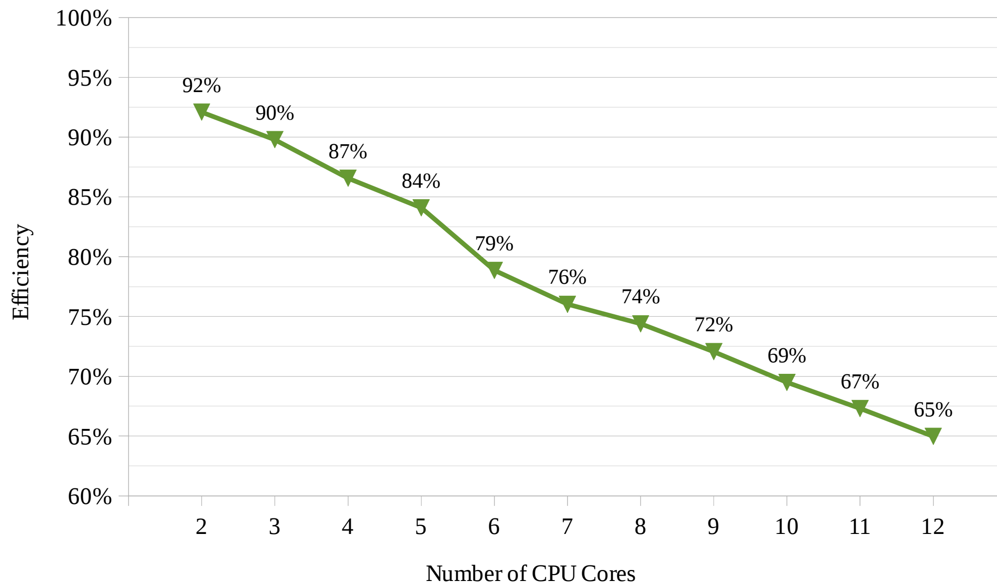


Figure 3.3: Efficiency Curve for Multi-CPU Parallelism of Stage 4.1

formula, $h(p)$, is shown in Equation 3.8.

$$h(p) = -0.027752p + 0.944356 \tag{3.8}$$

Substituting p=20 in $h(p)$ yielded a projected efficiency of 38.9% ($\approx$ 39.3%). This efficiency is considered very low and impractical. Perhaps, it is desirable to keep efficiency $\geq 50\%$, which could be achieved using 16$\times$ CPU cores as per Equation 3.8. This translates to a speedup of $\psi^{CPU}_{max}(20) \lesssim 8.87\times$ as per Equation 3.6.

Alternatively, it is possible to determine ψ^{CPU}_{max} using Karp-Flatt metric. This metric, shown in Equation 3.9, determines the serial fraction of program experimentally [19]. It was previously assumed that serial fraction $f = 0$. However, with Karp-Flatt metric, f would be set to $f_e(p)$, which would essentially be composed of serial codes, parallelism overhead, and any other source of inefficiency, given p.

$$f_e(p) = \left(\frac{1}{\psi(p)} - \frac{1}{p}\right) \div \left(1 - \frac{1}{p}\right) \tag{3.9}$$

Figure 3.4 shows Karp-Flatt metric versus the number of CPU cores used; values

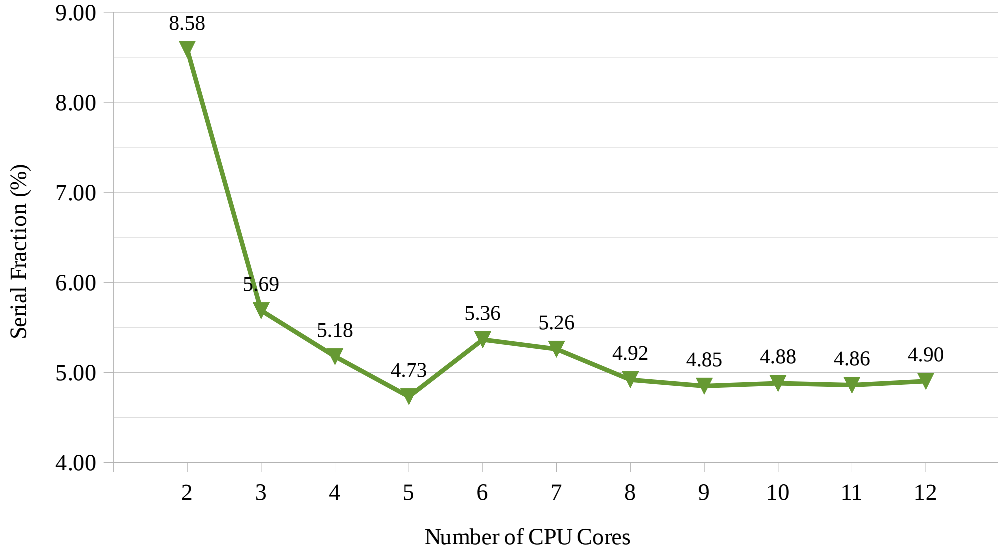


Figure 3.4: The Experimentally Determined Serial Fraction of Stage 4.1

for $\psi(p)$ are from Figure 3.2. Apparently, Karp-Flatt metric stabilized at an average value of ~4.88% after using 8× CPU cores. Assuming this steady-state would persist as $p \rightarrow \infty$, which is unlikely given inter-node communication overhead in a computer cluster environment, then according to Equation 3.2, $\psi^{CPU}_{max}(\infty) \lesssim 20.49\times$. Perhaps, $p = \infty$ is unrealistic and if one considers a desired speedup of 20×[1], then at least 793× CPU cores would be required to achieve $\psi^{CPU} \lesssim 20\times$. However, such a marginal projected speedup using a large number of CPU cores would result in an efficiency of ~2.5%, which is unacceptable in large computer-cluster environments.

To find ψ^{CPU}_{max} using Karp-Flatt metric and under the constraint that efficiency ≥ 50%, Equation 3.10 was first solved for p. This equation was obtained by substituting Equation 3.1 (Amdahl's Law) in Equation 3.7 (efficiency). Given $f = f_e(\infty) \approx 0.0488$ as previously assumed, the solution to Equation 3.10 was p=21. Finally, substituting

[1] Arbitrary number to give the reader a realistic example such that $\psi \approx 20.49$.

p=21 and f=0.0488 in Equation 3.1 yielded speedup $\psi_{max}^{CPU}(21) \lesssim 10.63\times$.

$$\left(\frac{1}{f + \frac{1-f}{p}}\right) \div p \geq 0.5 \tag{3.10}$$

The maximum achievable speedup obtained using Karp-Flatt metric was more optimistic than that obtained using Equation 3.6. This was due to Equation 3.6 implicitly assumed that parallelism overhead necessarily increased with the number of CPU cores used. However, in the case of using Karp-Flatt metric, this assumption was relaxed as parallelism overhead and other sources of inefficiency were set to an experimentally-determined constant ($f = f_e(p) \approx 4.88\%$). In short, it would be fair to state that the maximum achievable speedup in stage 4.1 for BM-A, using multi-CPU parallelism and under the constraint that efficiency is around 50%, would approximately be in the range [8.87×, 10.63×]. This could be achieved using roughly 16-21× CPU cores. This conclusion suggests that multi-CPU parallelism might not be the right paradigm of acceleration in stage 4.1 for fixed-size problems. This is especially the case given that stage 4.1 of cn.MOPS pipeline is embarrassingly parallel yet it did not exhibit high scalability with increased CPU cores. In subsequent chapters, gcn.MOPS is introduced which showed significantly higher speedups and better scalability with increased processing power compared to cn.MOPS.

Chapter 4

gcn.MOPS: Accelerating cn.MOPS with GPU

Like any project that is ought to be accelerated with GPU and CUDA, fundamental changes to the existing code and algorithms are often required. In this chapter, the technical details about how the modelling step of cn.MOPS, namely stage 4, was modified for GPU acceleration. Restructuring cn.MOPS into gcn.MOPS was carried out in three phases. In phase 1, which is described in § 4.1, the necessary code restructuring that was needed before designing a naïve kernel is presented. The naïve kernel (phase 2) and the optimized kernel (phase 3) are described in § 4.2 and § 4.3 respectively.

4.1 Staging the Code for Execution on GPU

The three basic steps of executing a program on GPU using CUDA could be summarized as follows:

1. copy input data from host memory to device memory (H2D);

2. launch the kernel to process the data on GPU;

3. copy results from device memory to host memory (D2H).

The question is how could stage 4.1 of cn.MOPS pipeline, shown in Figure 2.7, fit the GPU execution framework shown above? There were two issues that must be addressed to make this happen. The two issues were related to branch #1 and branch #2 in Figure 2.7 and are discussed below.

4.1.1 Defragmenting the Design

The problem with branch #1, which is executed in R, is that if it evaluates to `TRUE`, an R script is executed; and if it evaluates to `FALSE`, the core mathematical model is executed by invoking `cnmops(..)`$_{\mathrm{C}}$. In both branches, there were other R statements and calculations related to the core algorithm. This is inconvenient from a technical point of view because there is no direct CUDA support for R. Since the core mathematical model of cn.MOPS was already written in a language that is supported by CUDA, it made sense to rewrite the R portion of branch #1 in C/C++ as well. This made the original core algorithm of cn.MOPS, which was implemented in `.cn.mopsCE(..)`$_{\mathrm{R}}$, fully written in C/C++. Thus, `.cn.mopsCE(..)`$_{\mathrm{R}}$ was eliminated and `gcnmops(..)`$_{\mathrm{C}}$ was introduced. This new function was exactly like `.cn.mopsCE(..)`$_{\mathrm{R}}$ shown in Figure 2.7 except it was fully written in C/C++. Therefore, it became possible to later use CUDA C/C++ keywords and directives to turn the core algorithm into a GPU kernel. That is, it became possible to carry out the second step of the GPU execution framework shown at the beginning of this chapter. Besides, `gcnmops(..)`$_{\mathrm{C}}$ made the design coherent and unfragmented as all the logic pertained to the algorithm was kept in one place.

4.1.2 Eliminating Functional R Codes

As stated earlier, branch #2 is a loop which is originally implemented using `apply(..)`$_{\mathrm{R}}$ as shown in Listing 2.7. Assuming `gcnmops(..)`$_{\mathrm{C}}$ is a functioning kernel, there would be as many kernel launches as there are rows. This also implies that there would be

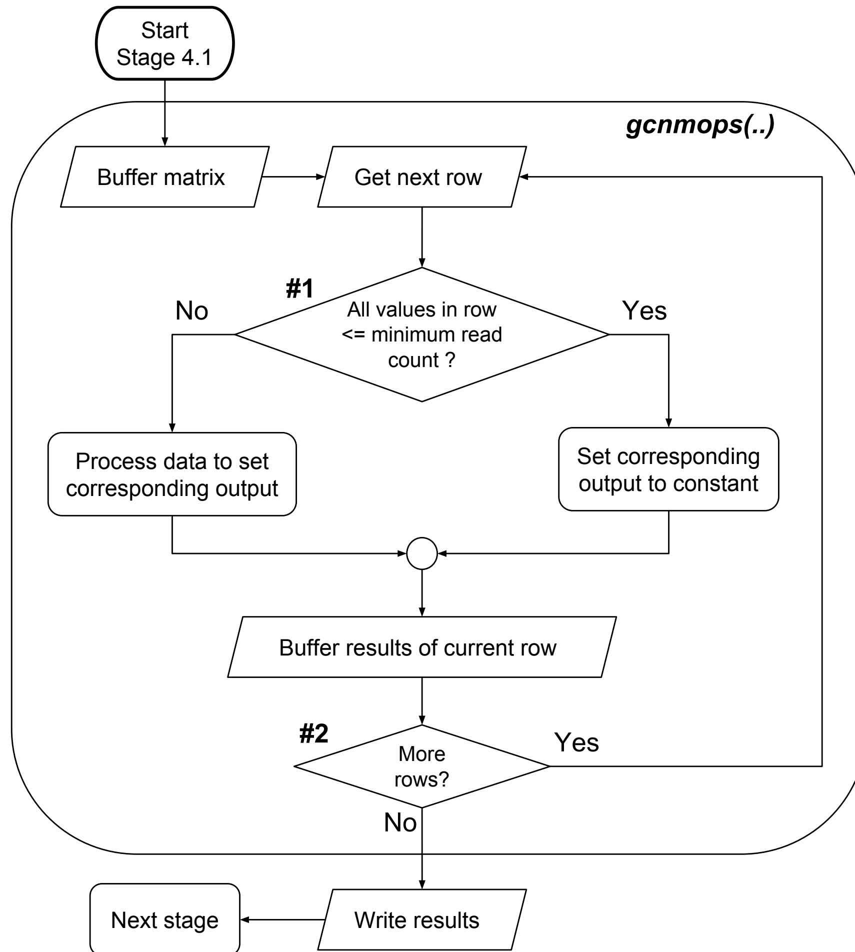


Figure 4.1: The Program Flow of gcn.MOPS in Stage 4.1 (Phase 1)

as many H2D and D2H memory-copy operations as launched kernels. Although this might seem to fit the GPU execution framework, there are two problematic issues. First, a kernel that is launched to process a single row is essentially single-threaded. This is extremely inefficient as GPU is able to concurrently handle hundreds to thousands of threads. Second, H2D and D2H memory-copy operations are expensive and

should be minimized if possible. In order to solve these two issues, all rows were passed to `gcnmops(..)`$_{\text{C}}$ as a whole matrix. In addition, `apply(..)`$_{\text{R}}$ was replaced with a conventional for-loop that took place inside `gcnmops(..)`$_{\text{C}}$. Figure 4.1 shows the program flow in stage 4.1 after modifying branch #1 and #2 to stage the code for GPU execution.

4.2 From cn.MOPS to Naïve gcn.MOPS

In phase 1, `gcnmops(..)`$_{\text{C}}$, which still executed on CPU, was introduced in § 4.1. In this section, which details phase 2, this function is turned into a naïve kernel. Further, a new interface, named `gcn.mops(..)`$_{\text{R}}$, was provided; it took an extra integer argument called `gpu`$_{\text{R}}$ which specified the device to be used. Earlier, it was noted that there is no direct CUDA support in R. Therefore, it was not possible to directly launch a kernel from within R using interface `.Call`$_{\text{R}}$ as shown in Listing 2.5. Listing 4.1 shows how a simple kernel is launched in a C++ application. To launch this kernel from within R, the arguments of the kernel can be passed via `.Call`$_{\text{R}}$. Though, the kernel configuration parameters, which are inside the `<<<...>>>`$_{\text{C}}$ notation, are not directly accommodated in `.Call`$_{\text{R}}$. To circumvent this issue, a C/C++ wrapper called `gcnmops_w(..)`$_{\text{C}}$ was written. From within R, the wrapper was invoked via `.Call`$_{\text{R}}$ and the kernel was launched from within `gcnmops_w(..)`$_{\text{C}}$.

Listing 4.1: Launching a Simple Kernel

```
1 #include <stdio.h>
2 __global__ void helloGPU() { printf("Hello from GPU\n"); }
3
4 int main(int argc, char **argv) {
5   dim3 grid(atoi(argv[1]));
6   dim3 block(atoi(argv[2]));
7   helloGPU <<<grid,block>>> (/*args*/);
8   return 0;
9 }
```

The rest of this section details the following points in order:

- how user arguments for the modelling step were passed to the kernel.
- how input and output were transferred between an R session and GPU memory.
- how GPU threads were mapped to input and output spaces.
- how large data sets which did not fit in GPU memory were processed.

4.2.1 Setting up User Arguments

In Listing 2.6, processing starts by calling `cn.mops(..)`$_{\mathrm{R}}$ with user arguments. The following user arguments propagate to stage 4.1:

- `input`: a numeric/integer matrix or a `GRanges` object, which is normalized in stage 3 and passed to stage 4.1 as numeric matrix `X.norm`;
- `I`: a vector of length `n` which contains numeric values;
- `classes`: a vector of strings, whose length must be equal to the length of `I`, i.e. `n`. The string "CN2" must also exist in this vector;
- `priorImpact`: a single numeric value;
- `cyc`: a single integer value;
- `parallel`: a single integer value that specifies the number of cores to be used;
- `minReadCount`: a single integer value that is used in branch #1 (Figure 4.1);
- `returnPosterior`: a single logical value that indicates whether α_{ik} matrices are returned or discarded (Figure 2.8).

In addition, the core algorithm of cn.MOPS, `.cn.mopsCE(..)`$_{\mathrm{R}}$, also requires the following arguments which are computed from user arguments:

- `cov`: a single numeric value that is calculated according to user argument `norm` (line 10 of Listing 2.6);
- `idxCN2`: the index of string "CN2" in user argument `classes`;
- `alphaInit`: a numeric vector of length `n`. All values in this vector are set to the same constant except for the one at index `idxCN2`, which is set to a different constant. These constants are all hard-coded.
- `alphaPrior`: a numeric vector of length `n`. All values in this vector are set to zero except for the one at index `idxCN2`, which is set to `priorImpact`;
- `n`: the number of classes, i.e. the length of `classes`;
- `N`: the number of samples.

In the original design, `.cn.mopsCE(..)`$_{\mathrm{R}}$ is internally called with the following arguments: `X.norm, I, classes, cov, cyc, N, n, idxCN2, alphaInit, alphaPrior, minReadCount,returnPosterior`. With the new wrapper, `gcnmops_w(..)`$_{\mathrm{C}}$, the same arguments were also passed except for `N`$_{\mathrm{R}}$ and `n`$_{\mathrm{R}}$. These two arguments were instead extracted inside the wrapper using `GET_DIM(..)`$_{\mathrm{C}}$. If `gcnmops_w(..)`$_{\mathrm{C}}$ is invoked via `.Call`$_{\mathrm{R}}$, all arguments are individually packaged in `SEXP`$_{\mathrm{C}}$ objects and passed to `gcnmops_w(..)`$_{\mathrm{C}}$. Listing 4.2 shows the header of `gcnmops_w(..)`$_{\mathrm{C}}$. It should be noted that argument `parallel`$_{\mathrm{R}}$ no longer affects the performance of stage 4.1 in gcn.MOPS, so it was not passed.

Listing 4.2: The Header of `gcnmops_w(..)`$_{\mathrm{C}}$

```
1  extern "C" SEXP gcnmops_w(SEXP gpuS, SEXP x_normS, SEXP IS, SEXP classesS,
2    SEXP covS, SEXP cycS, SEXP idxCN2S, SEXP alphaInitS, SEXP alphaPriorS,
3    SEXP minReadCountS, SEXP returnPosteriorS);
```

Next, these `SEXP`$_{\mathrm{C}}$ objects were internally passed to a helper function called `setup_gcnmops_args(..)`$_{\mathrm{C}}$. This function extracted arguments relevant to GPU ex-

ecution from the $\texttt{SEXP}_\text{C}$ objects. Then, it repackaged these arguments in a C-struct, named $\texttt{gcnmops_args}_\text{C}$, which was returned and, later, passed to the kernel as an argument. Listing 4.3 shows the header of $\texttt{setup_gcnmops_args(..)}_\text{C}$ as well as $\texttt{gcnmops_args}_\text{C}$. In line 18, possible values for argument `d_or_h` are 'd'/'D' for "device" or 'h'/'H' for host. If the setup is meant for device memory, all pointers in the initialized C-struct are device pointers and, thus, cannot be dereferenced from host. Pointers `log2_I` and `abs_log2_I` are for arrays both of length `n` which were needed to compute $\texttt{ini}_\text{C}$ and $\texttt{sini}_\text{C}$. In cn.MOPS, values in these arrays were calculated in $\texttt{cn.mopsCE(..)}_\text{R}$, which meant they were redundantly recalculated for every row. In gcn.MOPS, values were precalculated once and were considered part of $\texttt{gcnmops_args}_\text{C}$ because they were derived from user arguments.

Listing 4.3: The Header of $\texttt{setup_gcnmops_args(..)}_\text{C}$

```
1   extern "C" struct gcnmops_args {
2     double *mem_handle; // internal use
3     // single-value constants
4     unsigned int nRanges_tot;
5     unsigned int nSamples;
6     unsigned int nClasses;
7     unsigned int minReadCount;
8     unsigned int cyc;
9     unsigned int idxCN2;
10    // array constants
11    double *I;
12    double *cov;
13    double *alphaInit;
14    double *alphaPrior;
15    double *log2_I;
16    double *abs_log2_I; };
17
18  static struct gcnmops_args setup_gcnmops_args(char d_or_h, SEXP x_normS, SEXP IS,
19    SEXP classesS, SEXP covS, SEXP cycS, SEXP idxCN2S, SEXP alphaInitS,
20    SEXP alphaPriorS, SEXP minReadCountS);
```

4.2.2 Input and Output Buffers

To process data on GPU, said data must exist in GPU's physical memory while results must be copied back to host's physical memory. As shown Figure 4.1, the input matrix is first buffered in GPU memory before processing starts. Then, the

results of each processed row are also buffered in GPU memory before writing them to the return objects. Buffering input in GPU memory was done as shown in Listing 4.4. In line 10, input matrix `x_normS`$_C$ is passed from R to `gcnmops_w(..)`$_C$ as a `SEXP`$_C$ object. Then, in line 13, a pointer to matrix data is obtained using `REAL(..)`$_C$. The number of rows and the number columns are obtained as shown in lines 14 and 15 respectively. In line 19, device memory is allocated and in line 21, data is copied from host to device. To avoid confusing the reader, details related to the size of GPU memory allocation in line 19 are not shown. These details will be explained in § 4.2.4.

Listing 4.4: Setting up Input Buffer on GPU

```
#include <Rdefines.h>
#include <cuda_runtime.h>

extern "C" struct X_norm {
  unsigned int nSamples;
  unsigned int nRanges;
  double *data;
};

extern "C" SEXP gcnmops_w(SEXP gpuS, SEXP x_normS, /* .. and other args .. */) {
  ...
  struct X_norm x_norm;
  x_norm.data = REAL(x_normS);
  x_norm.nRanges  = INTEGER(GET_DIM(x_normS))[0]; // n (number of rows)
  x_norm.nSamples = INTEGER(GET_DIM(x_normS))[1]; // N (number or cols)
  ...
  struct X_norm d_Xchunk; // to be passed to kernel
  ...
  CHK_CU( cudaMalloc((double**)&(d_Xchunk.data), /* nBytes */) );
  ...
  cpyDataToDev( /* args */ );
  ...
}
```

For the results, two contiguous linear memory buffers were allocated: one for host and another for device. The memory space of both buffers was logically divided into six regions such that the 1st region was for α, the 2nd was for λ, the 3rd was for α_ik, and so on. Further, each region was logically subdivided such that each sub-region was associated with a single genomic region. This logical organization is illustrated in Figure 4.2. To conveniently access these regions, a structure named `gcnmops_buf`$_C$

was introduced and is shown in Listing 4.5. Within `gcnmops_w(..)`$_{\mathrm{C}}$, a helper function was called to: 1) dynamically allocate memory; 2) initialize a `gcnmops_buf`$_{\mathrm{C}}$ structures; 3) map pointers to their respective logical region; 4) and finally return the C-struct by value. The function call was made once for host and once for device.

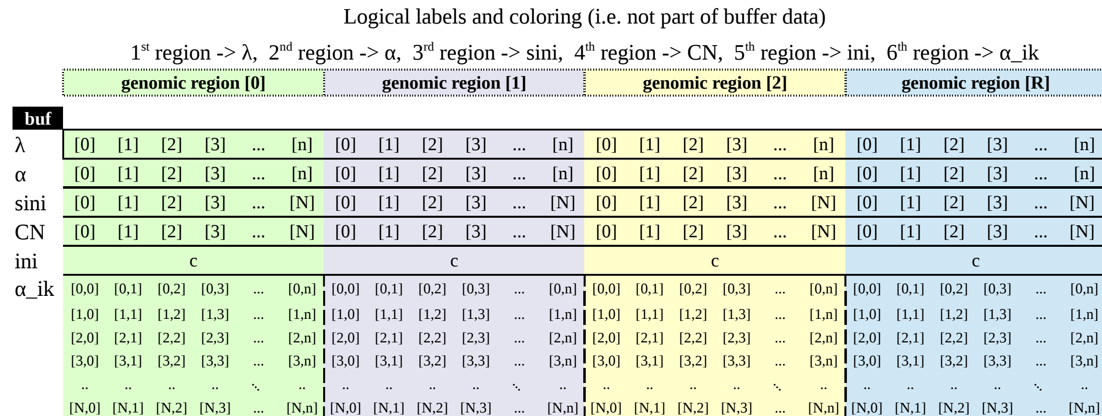


Figure 4.2: The Logical Organization of the Result Buffer (Phase 2)

Listing 4.5: Organizing the Result Buffer in struct `gcnmops_buf`$_{\mathrm{C}}$

```
1  extern "C" struct gcnmops_buf {
2    // metadata
3    unsigned int nSlices;
4    unsigned int nBytes_slice;
5    double *mem_handle; // internal use
6
7    double  *alpha_est;
8    double  *lambda_est;
9    double  *alpha_ik;
10   double  *ini;
11   double  *ExpLogFoldChange; //sini
12   double  *expCN_idx;
13 };
```

Once processing finished in GPU, results were copied from GPU buffer to host buffer. These results were, again, copied from host buffer into pre-allocated `SEXP`$_{\mathrm{C}}$ objects. These objects were bound together in a single `SEXP`$_{\mathrm{C}}$ object and this object was returned by `gcnmops_w(..)`$_{\mathrm{C}}$ to the calling R function. In Figure 2.8, each vector/matrix/constant was stored in a separate `SEXP`$_{\mathrm{C}}$ object, but both host and GPU result buffers, as shown in Figure 4.2, stored all vectors contiguously. Accordingly,

after copying results from GPU buffer to host buffer as a single memory chunk, each logical sub-region in the host buffer was then individually copied into its respective `SEXP`$_{\mathrm{C}}$ object. While performing two memory-copy operations might sound expensive, this was not the case as it will be shown in § 4.3.6. In either case, direct scatter-copy operation from GPU buffer to `SEXP`$_{\mathrm{C}}$ objects using `cudaMemcpy2D(..)`$_{\mathrm{C}}$ was extremely slow.

The size of the result buffer depended on the number of samples (N), the number of classes (n), and the number of genomic regions/ranges (*nRanges*). Thus, for each range, the required number of storage elements (*nElem_GR*) was calculated using Equation 4.1. Meanwhile, Equation 4.2 was used to calculate the total size (in elements) of the result buffer.

$$\text{nElem_GR} = (2 \times n) + (2 \times N) + (n \times N) + 1 \tag{4.1}$$

$$\text{nElem_buf} = \text{nElem_GR} \times \text{nRanges} \tag{4.2}$$

In the result buffer, all elements were of type `double` even `expCN_idx`$_{\mathrm{C}}$. It was mentioned in § 2.5 that CN was a vector of strings. For each genomic region, cn.MOPS core algorithm calculated N indexes, corresponding to elements in CN, whose values were between zero and n. Next, these indexes were used to fetch the corresponding strings from `classesS`$_{\mathrm{C}}$. Then, the fetched strings were copied to the N elements of CN, which was a `SEXP`$_{\mathrm{C}}$ object. In gcn.MOPS, strings were not directly set in the result buffer within the kernel. Instead, the indexes were calculated in the kernel while the strings were copied on host within `gcnmops_w(..)`$_{\mathrm{C}}$. It was done this way because copying a string into a `SEXP`$_{\mathrm{C}}$ object is not done using `memcpy(..)`$_{\mathrm{C}}$, but involves using macro `SET_STRING_ELT(..)`$_{\mathrm{C}}$ and function `mkChar(..)`$_{\mathrm{C}}$. Thus, there was no perceived benefits of string copying for CN within the kernel. That is, it was

not possible to simply copy string bytes from host buffer into $\texttt{SEXP}_\mathrm{C}$ object as it was the case with other result vectors.

4.2.3 Mapping Threads to Memory Space

Threads were mapped to three, separately allocated, GPU memory spaces: the input matrix, the result buffer, and the draft space. Since there was no dependency between rows in an input matrix, each GPU thread might independently handle a single row. A kernel could be configured to create thousands of threads. Thus, each thread could be assigned a row using its unique global ID, which could be calculated as shown in Equation 4.3. In CUDA, this could be done for a column-major matrix as shown in Listing 4.6.

$$\text{global thread ID} = (\text{block index} \times \text{block size}) + \text{thread index} \tag{4.3}$$

Listing 4.6: Associating Threads with Rows in a Column-major Matrix

```
1 __global__ void some_kernel(double *matrix, int rows, int cols /*, other args */) {
2   unsigned int g_tid = blockIdx.x * blockDim.x + threadIdx.x;
3   if(!(g_tid < rows)) return;
4   double *x = matrix;
5   x += g_tid;
6   ...
7   // process x
8   ...
9 };
```

However, the number of rows can be greater than the maximum allowed number of threads in a kernel launch. Consequently, the loop shown in Figure 4.1, which is branch #2, was not substituted for the indexing scheme shown in Listing 4.6. Instead, it was turned into a grid-strided loop. In other words, threads were mapped to multiple rows which were separated by a stride of length g, i.e. the number of threads in the grid. Table 4.1 illustrates how work was distributed in a grid of g

threads given an input matrix, similar to that shown in Table 2.1, with $(i \times g)$ rows. If the number of rows was not a multiple of g, some higher-order threads in the last

Thread ID	**iteration 0**	**iteration 1**	**iteration 2**	**...**	**iteration *i***
Thread 0	row 0	row g	row 2g	...	row i×g
Thread 1	row 1	row g+1	row 2g+1	...	row i×g + 1
Thread 2	row 2	row g+2	row 2g+2	...	row i×g + 2
Thread 3	row 3	row g+3	row 2g+3	...	row i×g + 3
⋮	⋮	⋮	⋮	⋱	⋮
Thread g-1	row g-1	row 2g -1	row 3g-1	...	row (i+1)×g -1

Table 4.1: Illustration of the Grid-strided Loop for Branch #2

iteration would be idle. On the same token, if the number of rows was less than the total number of threads, only one iteration would be needed and some higher-order threads would be idle.

The result buffer was mapped to threads in a broadly similar way to that described for the input matrix. If each memory region in Figure 4.2 was thought of as a C-like, row-major matrix, then result and input buffer would be similar. With the grid-strided loop design, multiple rows, separated by a stride of size g, of each buffer memory region were mapped to a thread. Though, there were two minor differences between mapping the result buffer and the input buffer. First, the result buffer consisted of six pointers as shown in Listing 4.5. Second, each memory region in the result buffer was perceived as a row-major matrix while the input matrix was a column-major matrix. Thus, mapping threads started by creating a local copy of the buffer C-struct for each thread. Next, each thread initially incremented the internal pointers of its local copy to map these pointers to its corresponding sub-region within each memory region. The initial incrementation size was `g_tid` $\times\, c(r)$, where $c(r)$ was the constant number of elements per vector per memory region r. After data processing, each thread wrote its results in its mapped sub-region. If the

same thread would perform another iteration to process another row, the local copy of the buffer would be incremented by $g \times c(r)$. To ease the process of incrementing these pointers, a C macro was written which is presented in Listing 4.7. If there were more threads than rows in an iteration, then some buffer regions associated with higher-order threads would be unused.

Listing 4.7: Mapping Threads to the Result Buffer

```
// this macro is defined in a separate file
#define INC_GCNMOPS_BUF(inc_val, numClasses, numSamples, buffer) { \
  unsigned int MACROVAR__n_inc_val = inc_val * numClasses; \
  unsigned int MACROVAR__N_inc_val = inc_val * numSamples; \
  unsigned int MACROVAR__nN_inc_val = inc_val * numClasses * numSamples; \
  \
  buffer.lambda_est += MACROVAR__n_inc_val; \
  buffer.alpha_est += MACROVAR__n_inc_val; \
  buffer.expCN_idx += MACROVAR__N_inc_val; \
  buffer.ExpLogFoldChange += MACROVAR__N_inc_val; \
  buffer.ini += inc_val; \
  buffer.alpha_ik += MACROVAR__nN_inc_val; \
}
...
// inside the kernel
  unsigned int g_tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(!(g_tid < rows)) return;
  ...
  struct gcnmops_buf local_buf = res_buf;  //copy internal pointers of buffer
  INC_GCNMOPS_BUF(g_tid, n, N, local_buf); //point buffer to this thread's region
...
}
```

As for the draft space, it was used for intermediate calculations. In cn.MOPS, there is a temporary array of size N named `lg`$_{\text{C}}$, which is allocated and freed for every `gcnmops(..)`$_{\text{C}}$ invocation. Beside this temporary array, other ones were introduced in gcn.MOPS. For convenience, all temporary arrays were packed in a C-struct shown in Listing 4.8. For a grid with a total of g threads, g draft spaces of type

Listing 4.8: Intermediate Variables Used by gcn.MOPS

```
extern "C" struct gcnmops_tmpVars {
  double *mem_handle; //internal use
  double *x_over_cov_sorted;
  double *lambdaInit;
  double *lg;
  double *alpha_i;
  double *draftspace_ini;
};
```

`gcnmops_tmpVars`$_C$ were allocated. Then, a big chunk of GPU memory was allocated and internal pointers in each draft space were mapped to a region within this memory chunk. Table 4.2 illustrates how the draft space of each thread was wired to said memory chunk. Alternatively, Table 4.2 can be viewed as a C-like, row-major matrix,

Draft for..	`x_over_cov`	`lambdaInit`	`lg`	`alpha_i`	`draftspace_ini`
Thread 0	[0..N]	[0..n]	[0..N]	[0..n]	[0..N]
Thread 1	[0..N]	[0..n]	[0..N]	[0..n]	[0..N]
Thread 2	[0..N]	[0..n]	[0..N]	[0..n]	[0..N]
Thread 3	[0..N]	[0..n]	[0..N]	[0..n]	[0..N]
⋮	⋮	⋮	⋮	⋮	⋮
Thread g-1	[0..N]	[0..n]	[0..N]	[0..n]	[0..N]

Table 4.2: Mapping Threads to Draft Spaces

in which data in a single row is contiguous and rows are placed besides each other contiguously as well. It should be noted that the draft space per thread was reused if a thread process more than a single row. Again, if there were more threads than rows in a single iteration, then some draft spaces associated with higher-order threads would be unused. The following is a description, that is relevant for this thesis, of C-struct `gcnmops_tmpVars`$_C$:

- `x_over_cov_sorted`: used in calculating `lambdaInit`. As the name suggests, it stored the sorted values of RCs of a genomic region/range divided by `cov`. In cn.MOPS, this is done in R to determine the mean and the median values. In gcn.MOPS, this operation was done in the kernel and, hence, this temporary array was needed. Values were sorted using the in-place bubble-sort algorithm;

- `lambdaInit`: used in calculating λ values. In cn.MOPS, This is calculated in R. Since the R portion of the core algorithm was written in C++, this temporary

variable was introduced.

- `alpha_i`: used in calculating α values. Originally, this array is allocated and returned to R. However, it is never used and, therefore, it was considered a temporary array.

- `draftspace_ini`: used in calculating *ini* value. In cn.MOPS, *ini* is calculated in R and this temporary array was introduced in gcn.MOPS due to rewriting the R portion of the core algorithm in C++.

4.2.4 Partitioning Large Data Sets

In the previous subsection, the problem of having more rows than threads was described and solved. But what about the case of having more rows than what GPU memory can accommodate? As it was mentioned before, there was no dependency between rows of an input matrix. Accordingly, the input matrix was partitioned into smaller chunks such that each chunk fitted in GPU memory. Then, the kernel was launched multiple times to process each chunk independently.

Given a GPU memory of size M bytes, the size of the largest chunk, in rows, was determined with the following steps:

1. the total size of all draft spaces, in bytes, was independent of the total number of rows in an input matrix because threads reused their draft spaces to process multiple rows. Table 4.3 presents the steps to calculate the total size of all draft spaces D. For simplicity, the total size of the `gcnmops_tmpVars` C-structs is not shown in the table as they were eliminated in the optimized kernel;
2. similarly, the size of dynamically allocated space for arguments was independent from the number of rows. It was also independent from the total number of

line	draft array	length per row	total size (bytes)
1	`x_over_cov`	N	$8 \times N$
2	`lambdaInit`	n	$8 \times n$
3	`lg`	N	$8 \times N$
4	`alpha_i`	n	$8 \times n$
5	`draftspace_ini`	N	$8 \times N$
6	total per thread	$3 \times N + 2 \times n$	$24 \times N + 16 \times n$
7	total for g threads	$g \times (3 \times N + 2 \times n)$	$g \times (24 \times N + 16 \times n) = D$

Table 4.3: Calculating the Size of the Draft Space in Bytes

threads in a gird. Table 4.4 presents the steps to calculate the total size of arguments A;

line	**argument array**	**length**	**size (bytes)**
1	`I`	n	$8 \times n$
2	`cov`	N	$8 \times N$
3	`alphaInit`	n	$8 \times n$
4	`alphaPrior`	n	$8 \times n$
5	`log2_I`	n	$8 \times n$
6	`abs_log2_I`	n	$8 \times n$
7	total	$5 \times n + N$	$40 \times n + 8 \times N$=A

Table 4.4: Calculating the Size of the Argument Space in Bytes

3. Then, the free GPU memory, in bytes, was calculated as per Equation 4.4;

$$F = M - D - A \tag{4.4}$$

4. to process a single row, result space and memory space for the row itself were needed. The total memory space per row, RB, was calculated in line 8.2 of Table 4.5;
5. finally, the maximum number of rows in a single chunk, C, was calculated as

line	**buffer array**	**length**	**size (bytes)**
1	row (genomic region)	N	$8 \times N$
2	`alpha_est`	n	$8 \times n$
3	`lambda_est`	n	$8 \times n$
4	`alpha_ik`	n×N	$8 \times n \times N$
5	`ini`	1	8
6	`ExpLogFold..`	N	$8 \times N$
7	`expCN_idx`	N	$8 \times N$
8.1	total per row (length)	$(2 \times n) + (2 \times N) + (n \times N) + 1$	
8.2	total per row (bytes)	$(16 \times n) + (16 \times N) + (n \times N) + 8 = RB$	

Table 4.5: Calculating the Size of the Argument Space in Bytes

shown in Equation 4.5.

$$C = \lfloor F \div RB \rfloor \tag{4.5}$$

In practice, `cudaMalloc(..)` allocated memory in blocks whose sizes were a constant multiple of bytes MBS. For instance, allocating 1 byte on the GPU memory of nVidia Tesla C2050 reserved a 1-MB memory block. Additionally, if GPU memory space of this device was divided into 1-MB chunks, then the amount of memory in the last chunk LCS would not be allocatable if its size was less than 1 MB. In gcn.MOPS, there was a total of three separate calls to `cudaMalloc(..)`$_{\mathrm{C}}$: the first was for the input matrix; the second was for arguments; and the third was for the result buffer. To minimize software complexity, it was assumed that these three allocations always reserved extra unused space of size MBS. Thus, Equation 4.4 was adjusted to account for these technicalities and Equation 4.6 was used instead.

$$F = \begin{cases} M - D - A - (3 \times MBS) - LCS, & \text{if LCS} < \text{MBS} \\ M - D - A - (3 \times MBS), & \text{if LCS} = \text{MBS} \end{cases} \tag{4.6}$$

Now that the maximum number of rows that could reside in GPU memory, C, is

known, a *partition table* for the input matrix could be created. The partition table held an array of pointers to dynamically-allocated C-structs of type `X_norm` (chunks). In lines 15-18 of Listing 4.9, `x_norm` held information about the whole input matrix and a data pointer to the beginning of the matrix. In the partition table, each `X_norm` C-struct held information about a unique sub-matrix within the input matrix. The data pointer of each said C-struct pointed to the beginning of the associated sub-matrix. The length (i.e. `nRanges`) of each sub-matrix/chunk would be $\leq C$. If the total number of rows was not a multiple of the number of chunks, the last chunk would have less rows than the rest. Listing 4.9 shows how a partition table was created for a given input matrix. In lines 28-34, a for-loop is used to launch as many kernels as there are chunks.

The header of the naïve kernel, shown in Listing 4.10, was deliberately not shown throughout this section to avoid confusing the reader. By now, the relevant information was already presented and the header would make more sense. `x_chunk` was discussed in § 4.2.4; `args` were discussed in § 4.2.1; `draftspace` was discussed in § 4.2.3; and `res_buf` was discussed in § 4.2.2.

4.3 Optimizing gcn.MOPS

The naïve kernel of gcn.MOPS yielded a performance improvement in stage 4.1 compared to the performance of cn.MOPS using a single CPU core as shown in Table 3.1. With the naïve kernel, the execution time of running BM-A became ∼2.2 minutes, which is a speedup of $\psi^{GPU} \approx 26\times$. In chapter 3, it was projected that the most optimistic speedup for cn.MOPS was $\psi^{CPU}_{max}(p = \infty) \lesssim 20.49\times$, which is less than what naïve gcn.MOPS offered.

To reveal potential opportunities of performance improvements, nVidia's perfor-

Listing 4.9: Creating a Partition Table for an Input Matrix

```
extern "C" struct X_norm {
  unsigned int nSamples;
  unsigned int nRanges;
  double *data; };

extern "C" struct X_partition_table {
  unsigned int nPartitions;
  struct X_norm *chunk; };

struct X_partition_table _create_partition_table( double *matrix,
  unsigned int x_nRanges, unsigned int nSamples, unsigned int chunk_len);

extern "C" SEXP gcnmops_w(SEXP gpuS, SEXP x_normS, /* .. and other args .. */) {
  ...
  struct X_norm x_norm;
  x_norm.data = REAL(x_normS);
  x_norm.nRanges  = INTEGER(GET_DIM(x_normS))[0]; // n (number of rows)
  x_norm.nSamples = INTEGER(GET_DIM(x_normS))[1]; // N (number or cols)
  ...
  struct X_norm d_Xchunk; // to be passed to kernel
  ...
  unsigned int x_partn_len = calc_prtn_len(
    x_norm.nRanges, x_norm.nSamples, n, GRID_SIZE, BLOCK_SIZE);
  struct X_partition_table x_norm_prtnTable =
    _create_partition_table(
      x_norm.data, x_norm.nRanges, x_norm.nSamples, x_partn_len);
  ...
  for(int itr = 0; itr < x_norm_prtnTable.nPartitions; itr++) {
    ...
    cpyDataToDev(d_Xchunk, x_norm_prtnTable.chunk[itr], nRanges_tot);
    ...
    // invoke the kernel
    ...
  }
  ...
}
```

mance profiler, *nvprof*, was utilized. This tool can collect run-time performance metrics of a launched kernel such as cache hit rate, the number of executed instructions, execution time of the kernel and CUDA APIs, etc.[16]. The collected metrics were then used to guide optimization efforts. To profile a kernel, the application which launches the kernel is run under *nvprof* as shown in Listing 4.11. If no `--metrics` are specified, the profiler provides execution time statistics.

In gcn.MOPS, the kernel was launched from within R instead of as a standalone application. Thus, profiling the kernel was not as straightforward as how it is done in Listing 4.11. Fortunately, an interactive R session can be started in shell under a debugger specified by the user. To profile a kernel launched from within R, the

Listing 4.10: Creating a Partition Table for an Input Matrix

```
__global__ void _gcnmops(
    struct X_norm x_chunk, struct gcnmops_args args,
    struct gcnmops_tmpVars *draftspace, struct gcnmops_buf res_buf);
```

Listing 4.11: Profiling an Application Using *nvprof*

```
(shell)$ nvprof --metrics inst_executed,l2_utilization ./application
==45808== NVPROF is profiling process 45808, command: ./application
...
==41575== Metric result:
Invocations   Metric Name      Metric Description       Min       Max        Avg
...
2            inst_executed   Instructions Executed   6291467   12058624    9175045
2            l2_utilization  L2 Cache Utilization    Low (1)   Mid (4)     Low (2)
```

R session is started in the shell such that *nvprof* is specified as a debugger (Listing 4.12). In order to collect performance metrics for the naïve kernel, benchmark BM-B was run and profiled under *nvprof*. At this point, the execution time of the naïve kernel was 4.75738 seconds. In the next subsections, which detail phase 3, various metrics were obtained, where applicable, and the necessary optimizations to enhance the kernel are presented.

Listing 4.12: Profiling a Kernel in an R Session Using *nvprof*

```
(shell)$ R -d "nvprof --metrics ins_executed,l2_utilization,other_metrics"
> source("cnmops_script.R")
...
==5941== NVPROF is profiling process 5941, command: /usr/lib/R/bin/exec/R
...
> q()
Save workspace image? [y/n/c]: n
==5941== Profiling application: /usr/lib/R/bin/exec/R
...
==5941== Metric result:
...
(shell)$
```

Before presenting the introduced optimizations, it should be noted that from now on, the naïve kernel was configured, at run time, to prefer L1 cache over shared memory. This preference instructs the device to assign more space of the fast, on-chip memory for L1 cache by assigning less space for shared memory. This configuration was chosen because optimization techniques introduced throughout this section never

utilized shared memory. Such configuration already reduced execution time from 4.75738s down to 4.05843s.

4.3.1 Altering Code Lines

Performance improvements can sometimes be achieved by altering program statements without changing the final answer. Generally, determining which statements to alter is speculative because *nvcc* is a proprietary compiler. Hence, to know whether some code alterations resulted in performance improvement, both tools *nvprof* and *cuobjdump* were used. The latter tool provides information about generated GPU assembly codes after a CUDA source is compiled with *nvcc*. To reduce the execution time of the kernel, multiple code lines were altered such that the number of issued instructions, obtained with metric `inst_issued` [16], was reduced without changing final answers. Determining which line to alter was done by trial and error as well as by understanding the code.

One of the introduced code alterations was related to variable `meanx`. Originally, this variable is used to calculate the average of all RCs in a row (genomic range/region) by summing these values and then dividing them by N. Later in the code, `meanx` is used to compute the values of λ, but before the computed result is stored, it is multiplied by N. This implied that instead of computing `meanx`, it was possible to simply compute `sumx` as shown in Listing 4.13. The effect of this code alteration is shown in Table 4.6, where the total of `MUL` instructions was obtained with the following shell command: ‘`cuobjdump -ptx -sass cn.mops.so | grep -c "MUL" > MULcount.txt`’

Another introduced code alteration was reordering independent statements. Listings 4.14 through 4.16 show portions of the core algorithm's code where statements were reordered without affecting the final answer. Together, these changes resulted in

Listing 4.13: Changing `meanx` into `sumx` in gcn.MOPS

```
...
  ...
  double meanx=0.0;
  for(k = 0; k < N; k++) {
    ...
    //original code: meanx += x[k*x_chunk.nRanges];
    /*new code:*/    sumx  += x[k*x_chunk.nRanges];
  }
  //original code: meanx=meanx/N;
  /*new code:*/  //discarded original code
  ...
    ...
    for(i=0; i<n; i++) {
      //original code: local_buf.lambda_est[i] = N*meanx/sumIalpha_i*I[i];
      /*new code:*/    local_buf.lambda_est[i] = sumx   /sumIalpha_i*I[i];
      ...
    }
  ...
...
```

Optimization:	naïve kernel	`meanx -> sumx`
total `MUL` instructions	116	114
issued instructions	4.2550×10^{10}	4.2549×10^{10}
execution time (s)	4.05843	4.05571

Table 4.6: The Impact of Changing `meanx` into `sumx`

a performance improvement as presented in Table 4.7. It was not entirely clear why

Optimization:	`meanx -> sumx`	after code reordering
issued instructions	4.2549×10^{10}	4.1675×10^{10}
execution time (s)	4.05571	3.92473

Table 4.7: The Impact of Code Reordering (Listings 4.14-4.16)

this particular order of statements reduced the number of instructions issued and enhanced performance. One possible explanation is that this particular order decreased the average number of replays per instruction executed (`inst_replay_overhead`). This metric remained constant at 0.098530 for the naïve kernel with L1 cache preferred and for the `sumx`-optimized one. After introducing the new order of statements, it decreased to 0.089098. That is, having less instruction replays resulted

Listing 4.14: Separating In-loop Statements into Independent Loops (1)

```
1  ...
2    // original code
3    /*
4    for(i = 0; i < n; i++) {
5      local_buf.alpha_est[i]=alphaInit[i];
6      local_buf.lambda_est[i]=lambdaInit[i]*I[i]; }
7    */
8
9    // new code
10   for(i = 0; i < n; i++)
11     local_buf.alpha_est[i]=alphaInit[i];
12   for(i = 0; i < n; i++)
13     local_buf.lambda_est[i]=lambdaInit[i]*I[i];
14 ...
```

Listing 4.15: Separating In-loop Statements into Independent Loops (2)

```
1  ...
2  // original code
3    /*
4    for(i=0; i<n; i++) {
5      local_buf.lambda_est[i]=sumx/sumIalpha_i*I[i];
6      local_buf.alpha_est[i]=(alphaPrior[i]+alpha_i[i])/(1+sumAlphaPrior);}
7    */
8
9    // new code
10   for(i=0; i<n; i++)
11     local_buf.alpha_est[i]=(alphaPrior[i]+alpha_i[i])/(1+sumAlphaPrior);
12   for(i=0; i<n; i++)
13     local_buf.lambda_est[i]=sumx/sumIalpha_i*I[i];
14 ...
```

in less instructions issued. The improvement in metric `inst_replay_overhead` was likely due to *nvcc* discovering optimization opportunities after introducing the new order of statements. This is unlike when performance was enhanced for the naïve kernel after configuring it to prefer L1 cache over shared memory. In that case, metric `inst_replay_overhead` decreased from 0.11933 down to 0.098529 due to increase in cache hit rates. But for the case of statement reordering, the cache hit rate remained fairly stable throughout the various optimizations introduced above ($\sim 70\%$ for global hit rate and $\sim 53\%$ for local hit rate).

Listing 4.16: Reversing Independent Statements

```
...
// original code
  /*
  for (k=0; k<N; k++){
    alpha_i[i]=alpha_i[i]+local_buf.alpha_ik[k*n+i];
    sumIalpha_i=sumIalpha_i+I[i]*local_buf.alpha_ik[k*n+i]*cov[k];}
  */

  // new code
  for (k=0; k<N; k++){
    sumIalpha_i=sumIalpha_i+I[i]*local_buf.alpha_ik[k*n+i]*cov[k];
    alpha_i[i]=alpha_i[i]+local_buf.alpha_ik[k*n+i];}
...
```

4.3.2 Using Constant Memory

GPU constant memory is physically part of the high-latency global memory, but it has its own cache which is separate from other cache memories [11]. Thus, reading constant-cached values has the same latency as reading from on-chip memory such as L1 cache. Meanwhile, if the requested constant is not cached, then the read operation becomes as slow as reading from global memory. For devices of compute capabilities 2.x, the size of constant memory is 64KB and the size of constant cache is 8KB [11].

There are three restrictions to utilize constant memory. First, values intended to be stored in constant memory can only be initialized and written by host. Second, threads in a launched kernel cannot access constant memory for writing. Third, symbols pertaining to constant values must be decorated with keyword `__constant__` and declared in the global section of the CUDA compilation unit. Thus, to utilize constant memory, values must be declared and initialized on host, then they are copied to device's constant memory using `cudaMemcpyToSymbol(..)`$_C$. Listing 4.17 demonstrates how constant memory is used.

Constant memory is mainly intended for a specific memory-read access pattern. If all threads in a warp access the same memory address for reading, then constant memory becomes useful. In this scenario, constant memory broadcasts the requested value to all threads with a single memory-read request. However, if some threads

Listing 4.17: Demonstration of Using Constant Memory

```
1  // cuda_source.cpp
2  __constant__ double const_val;
3
4  __global__ void some_kernel(/*.. args ..*/){
5    unsigned int g_tid = blockIdx.x * blockDim.x + threadIdx.x;
6    printf("Thread %d prints const_val: %lf\n", g_tid, const_val);
7    //Illegal: const_val++;
8    ...
9  }
10
11 void kernel_wrapper(double some_val, /*.. other args ..*/) {
12   double val = some_val;
13   CHK_CU( cudaMemcpyToSymbol(const_val, &val, sizeof(double)) );
14   ...
15   some_kernel<<GRID_SIZE, BLOCK_SIZE>>(/*.. args ..*/);
16   ...
17     }
```

access different constant values, then memory requests are serialized per value requested. Therefore, storing values in constant memory merely because these values do not change throughout the life of the kernel may not justify the use of constant memory to improve performance.

In the naïve kernel, there were 14 variables that fit the criteria mentioned above: `N`, `n`, `minReadCount`, `cyc`, `indxCN2`, `nRanges`, `eps`, `magic_val`, `I[n]`, `alphaInit[n]`, `alphaPrior[n]` `log2_I[n]`, `abs_log2_I[n]`, and `cov[N]`. Variable `magic_val` had constant value 1.0E-10 which was originally hard-coded in cn.MOPS; it was not understood what this value was for in the context of CNV detection, hence the name. Except for the last two variables, the values of all other variables were determined or extracted at run-time from user arguments passed from R. Thus, it was possible to copy their values to constant memory prior to launching the kernel.

Table 4.8 presents the performance results of using constant memory on the top of optimizations discussed in § 4.3.1. The number of global load transactions was obtained with metric `gld_transactions` [16]. Further, information about register spills, which are explained shortly, was obtained with *nvcc* flag '`-Xptxas -v`'. The improvement in performance was mainly due to decreased global load transactions.

Optimization:	§ 4.3.1	using constant memory
register spills (load)	408 bytes	280 bytes
register spills (store)	468 bytes	296 bytes
global load transactions	5.9544×10^{10}	5.9449×10^{10}
execution time (s)	3.92473	3.88567

Table 4.8: The Impact of Using Constant Memory

However, this decrease was not only because of the direct effect of constant memory. Using constant memory indirectly minimized register spilling which, subsequently, minimized global load transactions.

Register spilling occurs when a kernel needs more registers per thread than what is available. In Fermi devices, each thread may use up to 63 registers for computations [2]. If the maximum limit is exceeded, the local, per-thread memory is used to store “spills” without intervention from user. In other words, local memory is solely managed by device, like L1 cache, and user cannot explicitly access it. For instance, if a thread separately declared 70 integers, then 63 integers might be stored in registers while the other 7 integers might be spilled to local memory. Perhaps, the amount of register spills also depends on the program structure and compiler optimizations. Register spilling has a negative impact on performance because the local memory is actually a per-thread, private region that is located on the global memory. That is, reading and writing to local memory has the same latency and performance implications as global memory’s. Therefore, minimizing register spilling will more likely yield improved performance [2].

As presented in Table 4.8, there was a substantial decrease in register spills after using constant memory. This implied that data stored in constant memory did not end up in registers if used, as is, for calculations. Consequently, this made more registers available for some variables that ended up in local memory before using

constant memory. To confirm this hypothesis, a test kernel was written which is shown in Listing 4.18. The kernel did not do anything useful in particular and it was not meant to be launched at all. The only purpose of this kernel was to investigate the generated GPU assembly code. Briefly, the kernel did the following:

1. received two integer arguments `i` and `j`;
2. used `j` and `__constant__` symbol `cons_val` to compute a value;
3. stored the computed value in an address;
4. stored argument `i`, as is, in another address.

Listing 4.18: Test Kernel to Detect How Constant Symbols were Fetched

```
1  __constant__ int const_val;
2
3  __global__ void kernel(int i, int j) {
4    // pointers with dummy values used to prevent nvcc from optimizing away our test
5    int *trick_ptr1 = (int*)0xaaaaaaaa;
6    int *trick_ptr2 = (int*)0xcccccccc;
7
8    int new_j = j + 0xabcd;
9    int k = new_j * const_val;
10
11   *trick_ptr1 = i;
12   *trick_ptr2 = k;
13
14   return;
15 }
```

Listing 4.19 shows the generated, device-specific assembly code after compiling the kernel. The following points were observed:

- Arguments `i` and `j` were initially stored in constant memory. The syntax `c[X][Y]` in lines 9 and 10 was interpreted as follows: '`c`' was for constant memory [13]; as for `X` and `Y`, it is mentioned in [15] that constant memory is organized in banks, so `X` likely indicated bank number while `Y` indicated offset.
- Argument `j` was loaded into register `R0` (line 9) before it was used for the addition instruction (line 11).

Listing 4.19: The Generated Assembly Code of Listing 4.18

```
(shell)$ nvcc -arch=sm_20 -c const.cu -o const.so  # compilation command
...
(shell)$ cuobjdump -ptx -sass const.so            # view GPU-specific assembly code
...
code for sm_20
  Function : _Z6kernelii
.headerflags    @"EF_CUDA_SM20 EF_CUDA_PTX_SM(EF_CUDA_SM20)"
/*0000*/         MOV R1, c[0x1][0x100];     /* 0x2800440400005de4 */
/*0008*/         MOV R0, c[0x0][0x24];      /* 0x2800400090001de4 */
/*0010*/         MOV R2, c[0x0][0x20];      /* 0x2800400080009de4 */
/*0018*/         IADD R0, R0, 0xabcd;       /* 0x4800c2af34001c03 */
/*0020*/         ST.E [0xaaaaaaaa], R2;     /* 0x96aaaaaaabf09c85 */
/*0028*/         IMUL R0, R0, c[0x2][0x0];  /* 0x5000480000001ca3 */
/*0030*/         ST.E [0xcccccccc], R0;     /* 0x9733333333f01c85 */
/*0038*/         EXIT;                      /* 0x8000000000001de7 */
..........................
...
```

- Argument `i` was never needed for any calculation yet it was loaded into register `R2` (line 10) before it was stored in the given memory address (line 12).

- `__constant__` symbol `const_val`, initially stored in `c[0x2][0x0]`, was never loaded into any register even though it was needed for multiplication (line 13).

- if line 9 of Listing 4.18 was changed to ‘‘`int k = new_j * (const_val+1);`’’, then symbol `const_val` was loaded into a register (line 6 of Listing 4.20).

Listing 4.20: The Generated Assembly Code of Listing 4.18

```
code for sm_20
  Function : _Z6kernelii
.headerflags    @"EF_CUDA_SM20 EF_CUDA_PTX_SM(EF_CUDA_SM20)"
    /*0000*/         MOV R1, c[0x1][0x100];  /* 0x2800440400005de4 */
    /*0008*/         MOV R2, c[0x0][0x24];   /* 0x2800400090009de4 */
    /*0010*/         MOV R0, c[0x2][0x0];    /* 0x2800480000001de4 */
    /*0018*/         MOV R3, c[0x0][0x20];   /* 0x280040008000dde4 */
    /*0020*/         IADD R2, R2, 0xabcd;    /* 0x4800c2af34209c03 */
    /*0028*/         IADD R0, R0, 0x1;       /* 0x4800c00004001c03 */
    /*0030*/         ST.E [0xaaaaaaaa], R3;  /* 0x96aaaaaaabf0dc85 */
    /*0038*/         IMUL R2, R0, R2;        /* 0x5000000008009ca3 */
    /*0040*/         ST.E [0xcccccccc], R2;  /* 0x9733333333f09c85 */
    /*0048*/         EXIT;                   /* 0x8000000000001de7 */
  ..........................
...
```

These observations together suggested that values read from symbols decorated with keyword `__constant__` might not necessarily be loaded into registers. In other

words, there likely existed a path for these symbols to reach execution units without having to store them in register file. This leads to the initial hypothesis that using constant memory reduced register spilling which reduced global load transactions and, hence, improved performance.

4.3.3 Coalescing Memory Accesses

Coalesced memory access is a pattern in which all memory requests of all threads in a warp fall within a contiguous region of memory. Accordingly, consecutive threads in a warp do not strictly need to make consecutively-addressed memory requests, as shown in Figure 4.3, to perform coalesced access. In Figure 4.4, consecutive threads

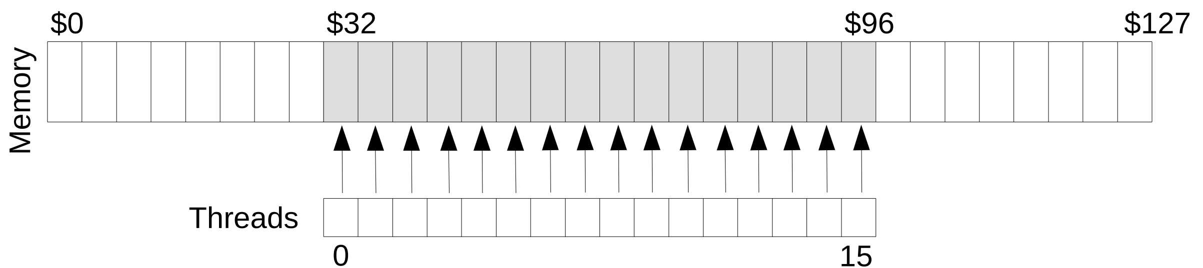


Figure 4.3: Coalesced Memory Access with Consecutive Addressing

in a warp make memory requests for random addresses, yet all requests are coalesced because they happen to fall within a contiguous region of memory. On the other

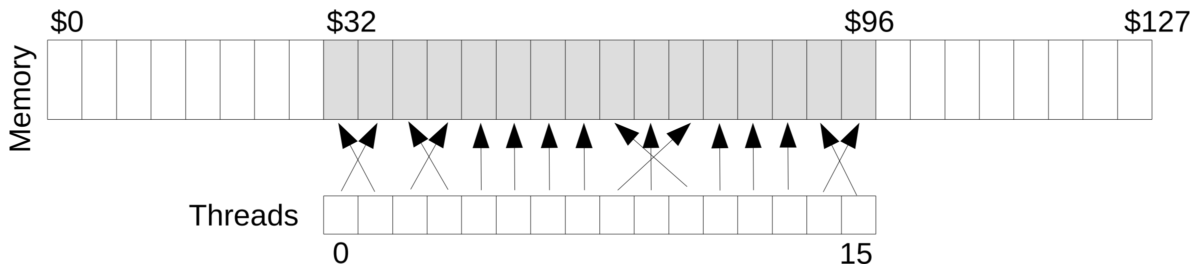


Figure 4.4: Coalesced Memory Access with Random Addressing

hand, uncoalesced memory access happens if the memory requests for all threads in a warp do not span a single contiguous region of memory. In Figure 4.5, random

threads make memory requests for random addresses, which results in an uncoalesced memory access. In Figure 4.6, consecutive threads in a warp make memory requests for consecutive addresses which are separated by a stride of a constant length. This also results in an uncoalesced memory access.

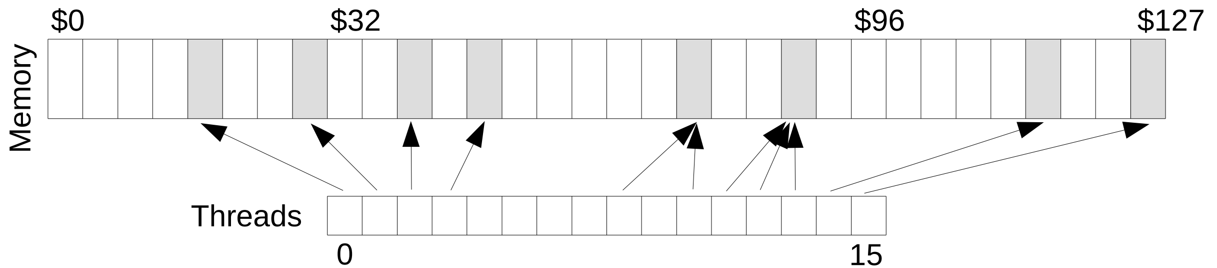


Figure 4.5: Uncoalesced Memory Access with Random Addressing

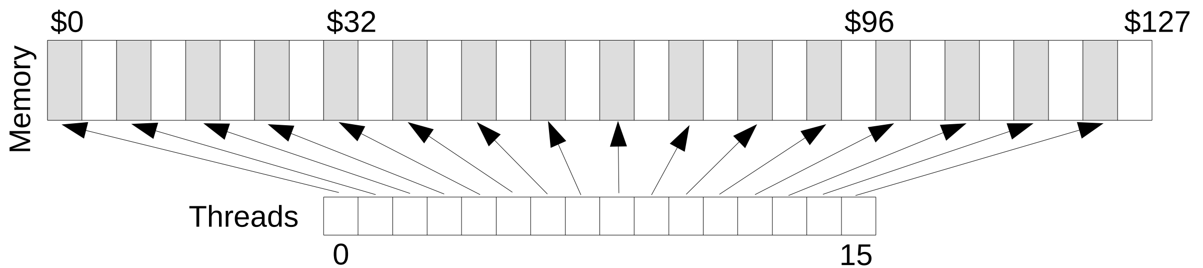


Figure 4.6: Uncoalesced Memory Access with Strided Addressing

Ensuring a software design, in which threads are arranged to make coalesced access to data, can positively impact execution performance. For instance, if thread T makes a memory-read request for address $\$addr$, the request is serviced by loading a contiguous, cache-line-sized chunk of memory. This chunk contains the requested data as well as neighbouring data. If neighbours of thread T also request neighbours of $\$addr$, then no further memory transactions are executed since data requested by neighbours of T were already loaded. That is, thread T and its neighbours made a coalesced memory access. This also applies for memory-write requests. The performance impact in this case is that less memory transactions are executed to service multiple threads. Since global memory transactions have high latency, minimizing

them results in a performance improvement.

In the naïve kernel, memory access patterns were inherited from cn.MOPS. For the input matrix, processing was performed by iterating over samples for every genomic region (GR), i.e. row, as explained in Table 4.9. This was the case for all other portions of the algorithm in which the input matrix was accessed for reading by all threads. This access pattern dictated that the input matrix was read column-by-column. Since the input matrix passed from R was stored in a column-major fashion, then all memory accesses to said matrix were coalesced. Therefore, no modifications were needed for this part of the core algorithm.

Thread ID	GR# (row)	RCs per DNA-seq Sample			
		Sample 1	Sample 2	Sample 3	Sample N
1	1	678	15	1670	...
2	2	594	7	1383	...
3	3	500	26	921	...
4	4	606	11	1457	...
5	5	604	11	1709	...
6	6	750	4	1790	...
T	T	...	...	...	⋱
data read in iteration i :		**i=0**	**i=1**	**i=2**	**i=N-1**

Table 4.9: Data Access Patterns for the Input Matrix

However, memory requests for elements in the result buffer as well as draft spaces were uncoalesced. This was confirmed by running BM-B to obtain metrics "global memory load efficiency" (`gld_efficiency`) and "global memory store efficiency" (`gst_efficiency`) [16]. The values for these two metrics were 9.23% and 25.00%, respectively. These considerably low percentages suggested that memory bandwidth was wasted as most of the elements loaded from global memory were unused. Figure 4.7 highlights the first element of each logical array in the result buffer as if memory access was requested by corresponding threads; there is no implied parallelism be-

tween vectors of different memory regions, e.g. λ and α. The memory access patterns for the result buffer are similar to that shown in Figure 4.6. For λ and α, the stride is of size n and for $sini$ and CN, the stride is of size N. Memory access requests for ini were coalesced like in Figure 4.3. For $alpha_ik$, the stride is of size $n \times N$ since matrices were stored besides each other.

Logical labels and coloring (i.e. not part of buffer data)

1st region -> λ, 2nd region -> α, 3rd region -> sini, 4th region -> CN, 5th region -> ini, 6th region -> α_ik

buf	Thread 0						Thread 1						Thread 2						Thread *T*					
	genomic region [0]						genomic region [1]						genomic region [2]						genomic region [R]					
λ	[0]	[1]	[2]	[3]	...	[n]	[0]	[1]	[2]	[3]	...	[n]	[0]	[1]	[2]	[3]	...	[n]	[0]	[1]	[2]	[3]	...	[n]
α	[0]	[1]	[2]	[3]	...	[n]	[0]	[1]	[2]	[3]	...	[n]	[0]	[1]	[2]	[3]	...	[n]	[0]	[1]	[2]	[3]	...	[n]
sini	[0]	[1]	[2]	[3]	...	[N]	[0]	[1]	[2]	[3]	...	[N]	[0]	[1]	[2]	[3]	...	[N]	[0]	[1]	[2]	[3]	...	[N]
CN	[0]	[1]	[2]	[3]	...	[N]	[0]	[1]	[2]	[3]	...	[N]	[0]	[1]	[2]	[3]	...	[N]	[0]	[1]	[2]	[3]	...	[N]
ini	c						c						c						c					
α_ik	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]	[0,0]	[0,1]	[0,2]	[0,3]	...	[0,n]
	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]	[1,0]	[1,1]	[1,2]	[1,3]	...	[1,n]
	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]	[2,0]	[2,1]	[2,2]	[2,3]	...	[2,n]
	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]	[3,0]	[3,1]	[3,2]	[3,3]	...	[3,n]
	..	..	..	..	⋱	..	..	..	..	..	⋱	..	..	..	..	..	⋱	..	..	..	..	..	⋱	..
	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]	[N,0]	[N,1]	[N,2]	[N,3]	...	[N,n]

Figure 4.7: Memory Access Patterns for the Result Buffer in the Naïve Kernel

In order to make all memory requests coalesced, each region in the buffer was reinterpreted. The 1st, the 2nd, the 3rd, and the 4th regions were alternatively viewed as separate, row-major matrices. It then became apparent that threads were accessing these matrices column by column, which was inefficient for row-major matrices. To eliminate this inefficiency, the four memory regions were individually treated as if they were transposed and, hence, consecutive threads made consecutively-addressed memory accesses. For a thread to read/write the next data element, the offset became equal to the total number of ranges in the chunk being processed. In other words, the logical sub-region of each thread was interleaved with the sub-regions of other threads as opposed to it being logically contiguous. This notion is visually presented in Figure 4.9. As for α_ik, it was viewed as a 3D array or a cuboid that was sliced into multiple matrices or XY-planes as in Figure 4.8. In the naïve kernel, threads

simultaneously accessed elements on the same XY-coordinates, but differed in the Z-coordinate. This was also inefficient since data was contiguous across the Y-axis for a given XZ-coordinate pair. Eliminating this inefficiency required more than treating the cuboid as transposed. First, the cuboid was sliced into multiple YZ-planes instead of multiple XY-planes. Then, each matrix, represented by a YZ-plane, was transposed such that the [X,0,0] elements were the beginning of each matrix. Next, the cuboid is glued back together and was sliced again into multiple XZ-planes. Each matrix, represented by an XZ-plane, was transposed such that the [0,Y,0] elements were the beginning of each matrix. Finally, the matrix was sliced into multiple XY-planes.

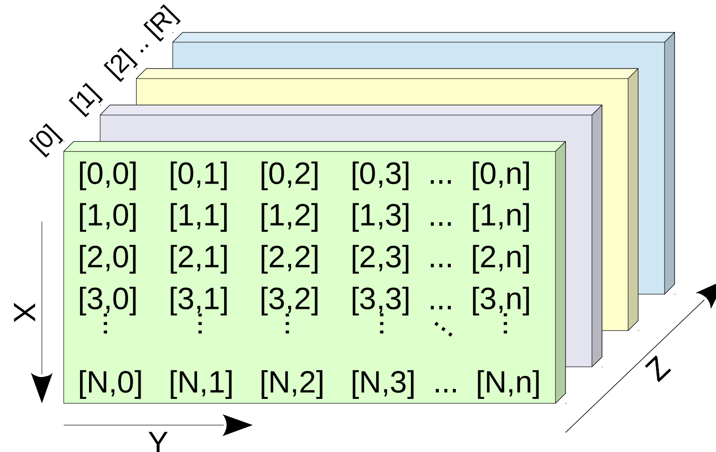


Figure 4.8: Visualization of Result Buffer α_ik in the Naïve Kernel

These series of transformations reinterpreted the cuboid such that consecutive threads simultaneously accessed a whole, contiguous row. Like the case of reading/writing elements for the first four regions in the result buffer, the offset to access the next element in α_ik was equal to the total number of ranges being processed. Processing started at the front-most matrix and went towards the X-axis direction. Once a whole matrix, represented by an XY-plane, was read/written, processing resumed for the next matrix in the cuboid. Figure 4.9 presents the new layout of the result buffer with which threads made coalesced memory requests. An alternative

view for the new arrangement of α_ik is also shown in Figure 4.9 in which data elements were interleaved and represented by a 2D, row-major matrix of arrays. As for the draft space, it was also reinterpreted in a fashion similar to that for the first four regions of the result buffer. With said reinterpretations, only one `gcnmops_tmpVars` C-struct was declared, initialized, and passed to the kernel. Then, each thread made a local copy of the C-struct and incremented its internal pointers to map them to the proper memory sub-region according to how each region was reinterpreted.

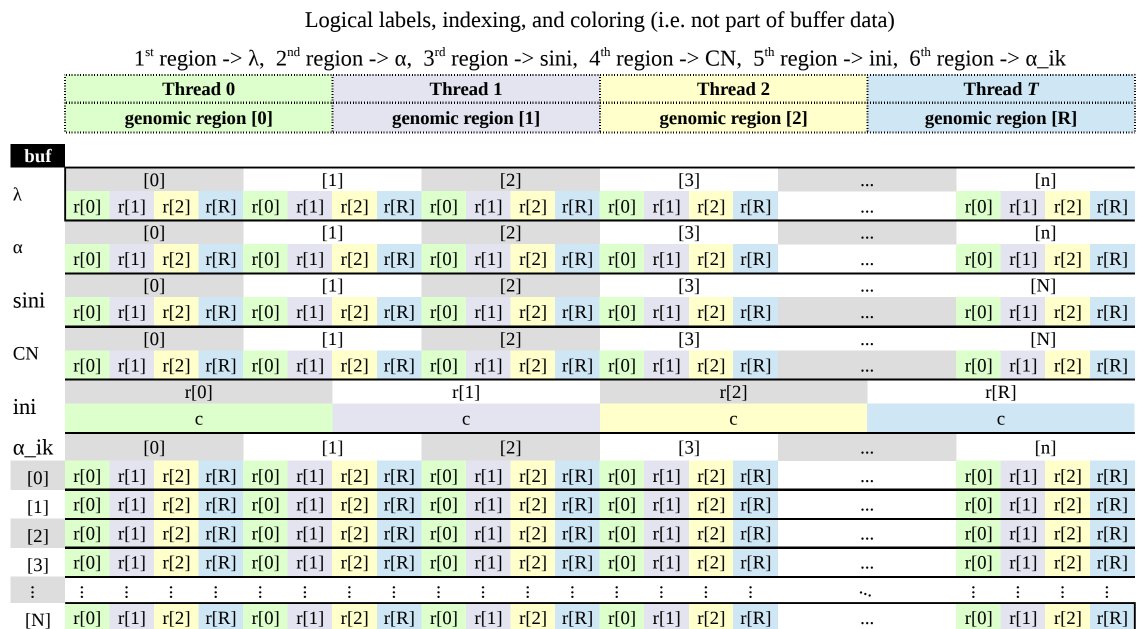


Figure 4.9: The Optimized Layout of the Result Buffer (Phase 3)

After these series of memory layout optimizations to ensure coalesced memory requests, one should expect a significant increase in metrics `gld_efficiency` and `gst_efficiency`. However, this was not the case as shown in the first two lines of Table 4.10. This indicated that there was another issue, other than the layout of the result buffer and the draft spaces themselves, which affected memory-request coalescing. The issue, which is discussed in § 4.3.5, was related to branch #1 in Figure 4.1. If gcn.MOPS was forced to always take the compute-intensive branch, then

metrics `gld_efficiency` and `gst_efficiency` substantially increased as presented in line 3 of Table 4.10.

line	Memory Layouts	gld_efficiency	gst_efficiency
1	Naïve buffer/drafspace	8.81%	25.00%
2	Optimized buffer/drafspace	15.62%	37.16%
3	Optimized buffer/drafspace (Branch #1 always `FALSE`)	57.30%	89.05%

Table 4.10: Memory Bandwidth Efficiencies for BM-B

4.3.4 Changing Data Layout of Results

The original layout of the result buffer was changed in order to ensure coalesced memory requests. Once results are ready and copied back to host, it might be tempting to think that the layout of results should be transformed back to the original layout shown in Figure 4.2. After all, the layout of the output from stage 4.1 of the pipeline, shown in Figure 2.8, looks logically similar to the layout of the naïve result buffer; one difference is that the memory space of the result buffer is contiguous while each vector in the output is allocated separately. Therefore, a scatter-copy operation from the result buffer to vectors in the return object will be required. This solution was counter-intuitive due to the overhead associated with both the process of reverse-transforming the result buffer and the scatter-copy operation. Such an overhead negated any performance improvement gained from coalescing memory requests in GPU.

Coincidentally, stage 4.2 merely transforms the output returned from stage 4.1 (Figure 2.8) such that it looks exactly like the optimized layout of the result buffer (Figure 4.9). The only difference is that in stage 4.2, new memory spaces for λ, α,

$sini$, CN, ini, and α_ik are allocated separately within R. That is, λ and α become separate matrices both of size $R \times n$ while $sini$ and CN become separate matrices both of size $R \times N$. ini becomes a separate vector of length R and α_ik becomes a separate, 3D array of size $R \times n \times N$. This knowledge was utilized to change the layout of the $\texttt{SEXP}_{\mathrm{C}}$ return-object and to skip stage 4.2. Accordingly, these separate allocations were, instead, made within $\texttt{gcnmops_w(..)}_{\mathrm{C}}$. Then, multiple memory operations were performed to copy each logical region in the result buffer to its proper space in the $\texttt{SEXP}_{\mathrm{C}}$ return-object. In this case, the the $\texttt{SEXP}_{\mathrm{C}}$ return-object held four matrices and two vectors. One of these vectors stored the values of α_ik even though it was reinterpreted as a transformed 3D array. The reason was there existed no mechanism to allocate 3D arrays for $\texttt{SEXP}_{\mathrm{C}}$ objects. To ensure that α_ik was interpreted correctly, its dimension attribute, $\texttt{dim}_{\mathrm{R}}$, was overwritten with the proper dimensions without data copying.

This new organization of the object returned from stage 4.1 was advantageous for the following reason. The execution time of stage 4.2 became negligible as the components of the object returned from stage 4.1 could be passed to stage 5 while bypassing all expensive memory operations in stage 4.2. Hence, only minor tasks were carried out in stage 4.2. An example of said minor tasks is assigning names to rows and columns of each matrix.

4.3.5 Eliminating Branch Divergence

Threads in a warp execute the same instruction for different data elements in a lock-step fashion. If a control-flow instruction, i.e. if-statement, is encountered, then different threads may take different execution paths. It is not possible for different threads in a warp to execute different instructions concurrently and, thus, the different paths are executed serially. This serialization of execution due to encountering

a control-flow statement in a warp is called *branch divergence*. The problem with branch divergence is that if instructions for path A are executed, threads that execute instructions for path B are forced to be idle. The idle thread may only resume execution after all instructions in path A are finished. After that, the threads which executed path A are forced to be idle until execution for path B is finished. Figure 4.10 illustrates how branch divergence is handled. Such an execution scheme hurts performance because of both serialization and resource underutilization due to idling threads. Ideally, it is desired that all threads are active all the time. It should be

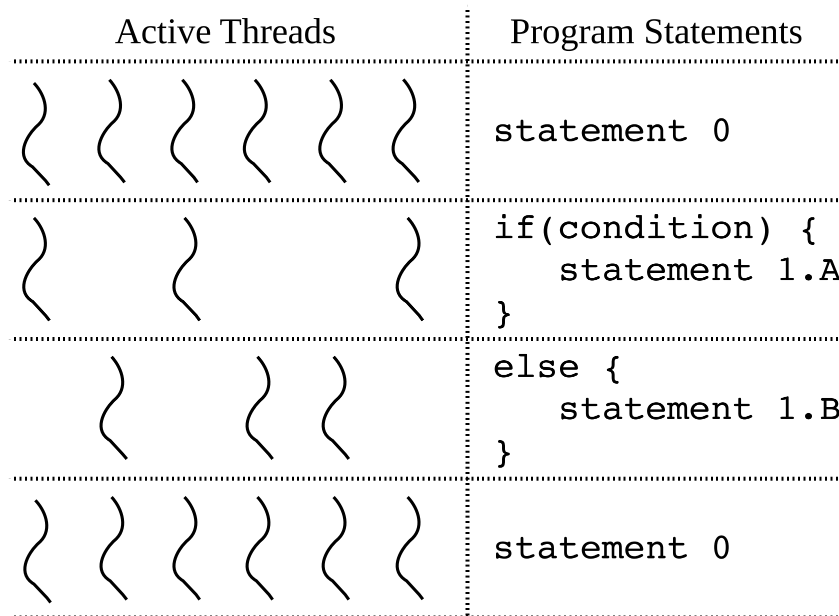


Figure 4.10: Illustration of Branch Divergence

noted that branch divergence is an inter-warp issue rather than intra-warp. In other words, different warps might take different paths without causing branch divergence as long as all threads in the same warp don't diverge.

After implementing the optimized buffer layout (§ 4.3.3), the ratio of non-diverging branches to total branches was obtained with metric `branch_efficiency` [16]. For BM-B, the ratio was 99.91%, which was considered a very good ratio. In § 4.3.3, it

was mentioned that branch divergence, which was just shown to be negligible, due to branch #1 in BM-B severely interfered with coalescing memory requests (Table 4.10). Experimentally, optimizing the kernel for branch efficiency alone did not yield a better performance. Likewise, optimizing the kernel for memory efficiency alone did not yield a better performance either. In Table 4.11, the execution times of the kernel for various optimizations are presented. Line 2 presents the execution time of the naïve kernel with optimizations described in § 4.3.1 and § 4.3.2. Lines 2 and 3 jointly convey the message that optimizing for either memory efficiency or branch efficiency alone hurt performance. Meanwhile, line 5 shows that a significant improvement in performance could only be achieved if the kernel was optimized for both branch efficiency and memory efficiency.

line	Optimization Technique	Execution Time (s)
1	naïve kernel	4.05843
2	§ 4.3.1 & § 4.3.2	3.88567
3	line 2 + § 4.3.3	3.98945
4	line 2 + branch #1 $\Rightarrow$ `FALSE`	7.41563
5	line 2 + line 3 + line 4	1.24644

Table 4.11: Kernel Performance for Different Optimizations (BM-B)

The question is what made such a tiny fraction of branch divergence so impacting? Why would eliminating it considerably improve performance after implementing the optimized buffer layout? To answer these questions, the input matrix (BM-B) was analyzed in the context of warp execution and branch #1 using R. In § 2.3, it was mentioned that warp size is 32 threads and that threads in a warp are consecutively indexed. Given that consecutive threads processed consecutive rows of the input matrix as previously shown in Table 4.1, then starting at the first row index of the input matrix, each group of 32 consecutive rows would map to a warp. That is, rows 0-31 would map to warp 0, rows 32-63 would map to warp 1, and so on. Since branch

#1 is evaluated based on `minReadCount=1`$_{\mathrm{R}}$, then it was possible to make a simple simulation of branch #1 to count diverging threads in each warp.

Figure 4.11 shows what percentage of all warps would contain what number of diverging threads if BM-B was executed on GPU. Around 18.6% of all warps contained

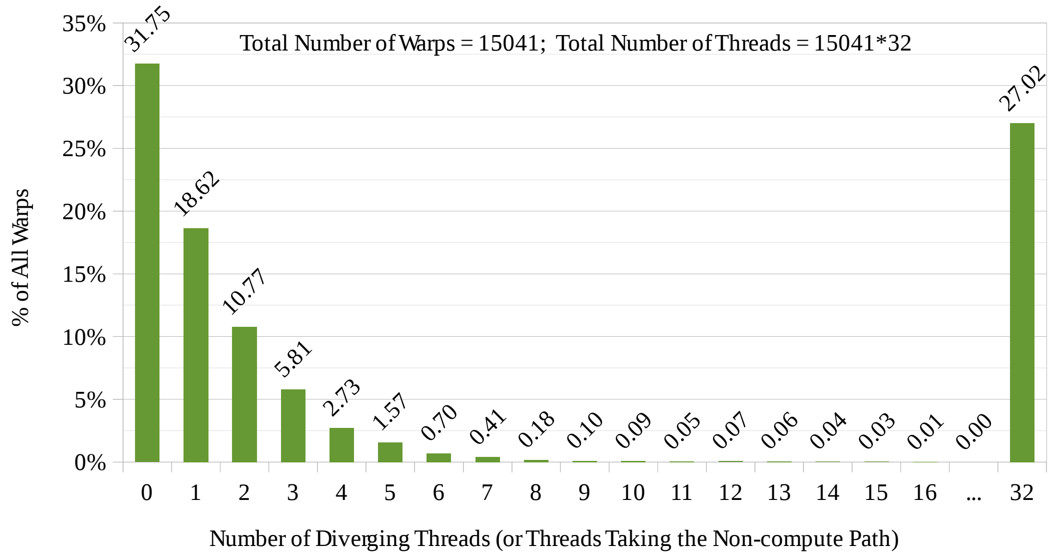


Figure 4.11: % of Warps with T Diverging Threads (BM-B)

a single diverging thread. The problem is that if a single thread in a warp diverged given the implementation of the optimized buffer layout, then separate memory instructions would be executed to serve the diverging thread. For example, if threads $T0$ through $T31$ in a warp requested consecutive integers $v[0]$ through $v[31]$, then their requests would be coalesced in a single memory transaction. However, if $T31$ diverged and requested $v[31+c]$ inside the `else` body, where $c > 0$, then another memory transaction would be issued to serve $T31$. This is also the case for warps with 2-16 diverging threads in which memory transactions might increase with the number of diverging threads depending on how close requested data element were to each other. This suggests that a single diverging thread in many warps could adversely affect performance if the not-taken-side of branch contained non-trivial amount of memory requests. The issue with metric `branch_efficiency` is it is blind to what happens

inside the `if-else` bodies [10]. In gcn.MOPS, the metric showed high branch efficiency while disregarding what happened inside branches. However, the non-compute path, which was executed by the diverging threads in BM-B, was composed of several statements of complexity $O(2n+3N+nN+2)$. Said statements were all independent, uncoalesced, memory operations that were not cheap in GPU terms. In short, due to the non-compute path being memory-intensive and given gcn.MOPS was optimized to coalesce memory accesses in the compute-intensive path, the existence of at least one diverging thread in a warp would increase memory requests for that warp by some factor (see appendix A). From Figure 4.11, ∼42% of all warps executing BM-B contained at least one diverging thread.

The above explanation about why the tiny fraction of branch divergence adversely affected performance was experimentally verified. This was done by obtaining the number of "executed load/store instructions" using metric `ldst_executed` [16]. As presented in Table 4.12, the number of LD/ST instructions substantially decreased if branch #1 was forced to always take the compute-intensive path. This also resulted in 3.2× speedup. The slight decrease in branch efficiency was due to the metric only accounting for control-flow statements that are part of the core mathematical model without involving branch #1. In other words, the presence of branch #1 slightly inflated branch efficiency. From Figure 4.11, this means that ∼42% of warps, or ∼30% of divergent rows, were responsible for generating ∼78% of all LD/ST instructions. Thus, eliminating branch divergence due to branch #1 in gcn.MOPS would lead

Optimization Technique	LD/ST Inst's	Branch Efficiency	Time (s)
§ 4.3.3 (coalescing mem-req)	2,152,885,240	99.91%	3.98945
§ 4.3.3 + branch #1 ⇒ `FALSE`	464,633,449	98.51%	1.24644
change% :	-78.41%	-1.40%	-68.76%

Table 4.12: Performance Metrics of Kernel with Branch #1 Always `FALSE`

to a better performance by decreasing LD/ST instructions and indirectly enhancing memory-request coalescing.

In order to eliminate branch divergence, branch #1 itself was eliminated from the kernel. This was done by decomposing `gcnmops(..)`$_{\mathrm{C}}$, which is the kernel, into two functions: `_gcnmops_gtMRC(..)`$_{\mathrm{C}}$ and `cnmops_leqMRC(..)`$_{\mathrm{C}}$. The former is a kernel function that handled the compute-intensive path while the latter is a host function that handled the other, none-compute path. In this setup, it was assumed that the compute-intensive path would always be taken and, thus, processing started by launching the kernel. Then, results were copied to host memory and, after that, `cnmops_leqMRC(..)`$_{\mathrm{C}}$ was invoked to overwrite the result for rows which should be processed in the non-compute path. The overhead of overwriting erroneous results was relatively insignificant due to the nature of the non-compute path. In this path, the core algorithm merely set erroneous results to constant values and, therefore, the whole setup achieved an overall improvement in gcn.MOPS performance.

4.3.6 Overlapping Host/Device Execution

Given an input matrix of length R, n classes, and N samples, the time complexity of `cnmops_leqMRC(..)`$_{\mathrm{C}}$ was $\mathrm{O}(R \times (2n + 3N + nN + 2))$. Normally, n and N are considerably smaller than R. For a very large R, the overhead of `cnmops_leqMRC(..)`$_{\mathrm{C}}$ might become noticeable. Such an overhead was hidden with host/device concurrency as follows. First, kernel launches are asynchronous with respect to host and, thus, it is possible to overlap host and device operations. Second, a large input was handled by partitioning it into multiple, smaller chunks; then, these chunks were processed independently (§ 4.2.4). These two pieces of information, together, implied that with proper synchronization, host operations acting on one chunk could be overlapped with device operations acting on another chunk.

Assuming a large input is partitioned into p chunks, each chunk C is identified by its serial number i, where $0 \leq i < p$. While a launched kernel is asynchronously processing C_{i+1}, host function `cnmops_leqMRC(..)`$_{\mathrm{C}}$ is invoked to concurrently overwrite erroneous results for C_i. By the time the last kernel is finished, $\frac{p-1}{p}\%$ of the overhead will have been absorbed by host/device concurrency. In gcn.MOPS, there were two other per-chunk, host operations whose associated overheads were also hidden in a similar fashion. In § 4.2.2, it was indicated that for technical reasons, strings for vector CN were made on host rather than device. This operation had an overhead which was hidden by host/device concurrency. In addition, copying large results from host buffer to the `SEXP`$_{\mathrm{C}}$ return-object had an associated overhead which was also hidden by host/device concurrency. In other words, results of C_i were copied to their corresponding spaces in the `SEXP`$_{\mathrm{C}}$ return-object while C_{i+1} was being processed on device.

Obviously, host became idle while device was processing C_0. To further decrease the overall execution time, there was a one-time, host operation which was overlapped with processing C_0 on device. The operation was allocating and setting up the `SEXP`$_{\mathrm{C}}$ return-object. The return object was not needed until some results were ready to be copied to it. Hence, this operation was overlapped with processing C_0 on device.

4.3.7 Disabling GPU's L1-Cache Memory

L1-cache global hit rate was not a major concern because L1-cache memory is mostly useful if data is reused after reading or writing it. Meanwhile, this was not the case with most parts of cn.MOPS's core algorithm. This also was the case after memory requests were coalesced as all data elements loaded into L1 cache were consumed by threads at once and never immediately needed again. Hence, optimizing for L1-cache global hit rate was counter-intuitive given it was foreseeable that L1-cache memory

might itself be a bottle neck.

Starting with the naïve kernel, L1-cache global hit rate was $\sim$ 70%, which was reasonably high. However, after all optimization techniques in this chapter were applied, the rate went down to $\sim$44% despite a considerable reduction in execution time. This indicated that L1 cache did not contribute to performance and that disabling it might improve performance. Normally, a global memory request goes to L1 cache first; if the request is a miss, then it goes to L2 cache and if it was another miss, the request goes to device's global memory (DRAM). If L1 cache is disabled, all global memory requests go to L2 cache directly without passing by L1 cache. For local memory requests, they always pass through L1 cache regardless of how it is configured. For nVidia's GPUs of compute capabilities 2.0 and higher, it is possible to disable L1 cache with compiler's flag `-Xptxas -dlcm=cg`.

After disabling L1 cache, various memory-related metrics, summarized in Table 4.13, improved. Further, there was a 3.3% reduction in execution time which was reasoned as follows. First of all, the fully optimized kernel still suffered from register

	naïve kernel	all optimizations	L1 cache disabled
register spills (load)	320 bytes	116 bytes	116 bytes
register spills (store)	500 bytes	108 bytes	108 bytes
L1 global hit rate	72.58%	44.59%	0.00%
L1 local hit rate	54.32%	57.08%	91.45%
L2 hit rate	72.29%	7.58%	35.68%
global load efficiency	8.79%	56.99%	86.88%
global store efficiency	25.00%	89.06%	89.06%
execution time (s)	4.05843	1.24644	1.20569

Table 4.13: Performance Metrics after Disabling L1 Cache (BM-B)

spilling, which meant local memory requests were made if spilled data was requested.

If L1 cache was enabled, both global and local memory requests competed for it. However, if it was disabled, local memory requests became the sole beneficiary of L1 cache. Consequently, L1 cache lines of spilled data, which were not accommodated in registers, were never invalidated because of global memory requests. This was confirmed by noticing a substantial increase in L1-cache local hit rate after disabling L1 cache. Technically, this implied that only 8.55% of all data spills, i.e. 20 instead of 224 bytes, ended up in DRAM. Meanwhile, the other 91.45% of spilled data became residents of L1 cache instead of DRAM. Moreover, global load efficiency was inline with global store efficiency only after L1 cache was disabled despite coalescing memory requests as discussed in § 4.3.3.

To summarize, Figure 4.12 shows how each suggested optimization technique progressively accelerated the kernel of gcn.MOPS running BM-B. Most of the speedup was the result of eliminating branch #1 from the kernel. In addition, it was shown how this major speedup would only be possible if memory requests were coalesced. The proposed buffer layout served the purpose of coalescing memory requests. In addition, it allowed gcn.MOPS to skip expensive memory operations in stage 4.2. Thus, these two optimization techniques were, essentially, mutually inclusive. The impact of host/device concurrency was not included in Figure 4.12 because BM-B was a small benchmark. That is, it had only a single chunk of data. Later, it will be clear how host/device concurrency enhanced the performance of gcn.MOPS for large data sets. Relatively, the other optimization techniques had minor impact on speedup, but implementing them did not affect software complexity and readability.

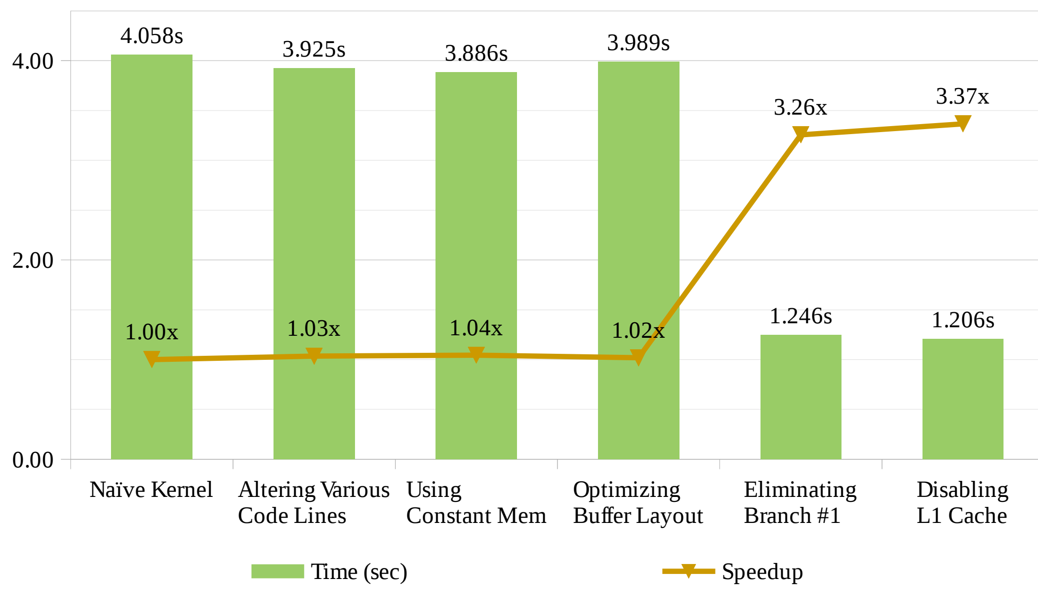


Figure 4.12: Progressive Speedup for all Optimization Techniques (BM-B)

Chapter 5

Experimental Results and Performance Analysis

In this chapter, a performance comparison between gcn.MOPS and cn.MOPS is presented. All optimization techniques discussed in chapter 4 were applied to gcn.MOPS.

5.1 Platform

The specifications of the used machine, Dell T7500, are detailed as follows:

- CPU: 2× Intel Xeon 6-core X5650 @ 2.67 GHz.
- RAM: 12× 4 GB, Multi-bit ECC, Registered (Buffered), DDR3 @ 1333 MHz.
- Storage: ATA Samsung SSD 840, Solid-state Drive with Capacity 500 GB.
- Graphic card <0>: nVidia Tesla C2050, Fermi architecture, Compute Capabilities 2.0, 14× stream multiprocessors (SMs), 448 CUDA cores, 2.6 GB DRAM.
- Graphic card <1>: nVidia Quadro 4000, Fermi architecture, Compute Capabilities 2.0, 8× stream multiprocessors (SMs), 256 CUDA cores, 2.0 GB DRAM.

In addition, the machine was configured from BIOS as follows:

- High I/O performance optimization was disabled, i.e. prioritized data flow between devices within the motherboard.
- Memory node interleaving was set to symmetric multiprocessing (SMP).
- Hardware prefetcher was enabled.
- Adjacent cache line prefeteching was enabled.
- Intel SpeedStep was disabled, i.e. system is put into highest performance state.
- Intel Turbo Boost Technology was disabled.
- Hyper-threading technology was disabled.

The following are relevant detailed about the system environment:

- Operating System: Linux Ubuntu 16.04.2 LTS, 64-bit.
- CUDA: version 8.0.
- R: version 3.2.3 (Wooden Christmas-Tree)
- Graphic driver: nVidia driver version 367.57.

5.2 The Build

The C/C++ sources were built and linked with *gcc/g++* version 5.4.0 20160609. For CUDA sources, *nvcc* which is part of CUDA compilation tools release 8.0, V8.0.44, and built on 04-SEP-2016 @ 22:14:01, was used. In addition R packge cn.MOPS version 1.18.0 was used. The bash script shown in Listing 5.1 was executed to: build

Listing 5.1: Bash Script for Building and Installing (g)cn.MOPS

```
#!/bin/bash
CURRENT_DIR=$(dirname $(readlink -f $0))
cd
cd $CURRENT_DIR/cn.mops
tar cvzf $CURRENT_DIR/cn.mops.tar.gz cn.mops
R CMD INSTALL $CURRENT_DIR/cn.mops.tar.gz
```

all (g)cn.MOPS source files into an R package; and install the package, as a library, in the R environment.

R does not automatically recognize CUDA sources and deal with them. Thus, a file named "Makevar" was placed in directory ../cn.mops/src. This file is like a makefile and is picked up by R's installation command to further guide the compilation process. The contents of Makevar are shown in appendix B.1.

5.3 Benchmarks

Two benchmarks based on real DNA samples from 1000 Genome Project were created. Each sample was whole-exome-alignment which was stored in BAM format (.bam) and accompanied by its indexing file (.bai). The samples were downloaded from the website of National Center for Biotechnology Information (NCBI) using IBM's plugin Aspera Connect. Below is the list of the used samples:

1. NA07048.mapped.ILLUMINA.bwa.CEU.exome.20120522.bam
2. NA07051.mapped.ILLUMINA.bwa.CEU.exome.20120522.bam
3. NA06984.mapped.ILLUMINA.bwa.CEU.exome.20120522.bam
4. NA06986.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
5. NA06989.mapped.ILLUMINA.bwa.CEU.exome.20120522.bam
6. NA07037.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam

7. NA11933.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
8. NA07347.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
9. NA10847.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
10. NA11843.mapped.ILLUMINA.bwa.CEU.exome.20120522.bam
11. NA11893.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
12. NA11894.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
13. NA11930.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
14. NA11931.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam
15. NA11932.mapped.ILLUMINA.bwa.CEU.exome.20121211.bam

For simplicity, the two benchmarks were named BM-A and BM-B. Table 5.1 presents the relevant details about both benchmarks. Specifically, BM-B was used

	BM-A	BM-B
number of samples	15	15
ref. chromosomes	1, 2, 3	21
genomic ranges (GRs)	6,904,726	481,299
branch #1 is `TRUE` for.. (`minReadCount`$_{\mathrm{R}}$`=1`)	9% of GRs	30% of GRs
size (as an array)	103,570,890	7,219,485
GRanges object size	447 MB	31 MB

Table 5.1: Technical Details of Used Benchmarks

to benchmark kernel optimizations, measure host memory footprints, and to conduct sensitivity analysis. Meanwhile, BM-A was used to benchmark both gcn.MOPS and cn.MOPS since its size was sufficiently larger than BM-B.

5.4 Methodology

To measure the execution time of each stage in the pipeline, `Sys.time()`$_{\mathrm{R}}$ was used [3]. System's time was captured before the start and after the end of each stage. Then, the difference in time was measured using `difftime(..)`$_{\mathrm{R}}$. For both gcn.MOPS and cn.MOPS, system's time was sampled at the same point of the program flow. On many occasions, it was noticed that R introduced a significant overhead when allocating a large memory space for the return object. However, this overhead disappeared after following these steps:

1. BM-A was run for the first time.

2. The return object was deleted.

3. R's garbage collector was never invoked with `gc()`$_{\mathrm{R}}$.

4. BM-A was run for the second time.

This was likely due to R needed to perform some initialization related to the creation of new objects. If newly created objects were deleted, R never released the memory space back to the operating system unless garbage collector was manually invoked. Thus, the disappearance of the overhead after step 4 might indicate that R was reusing the memory space it acquired from running BM-A the first time without the need to perform another initialization work.

The host memory footprint was measured using debugging tool *valgrind* and heap profiling tool *massif* as shown in Listing 5.2. Argument `parallel`$_{\mathrm{R}}$ in `cn.mops(..)`$_{\mathrm{R}}$ was set to zero because *massif* did not behave as expected and simply froze if multithreading was enabled. This adjustment was not expected to affect the results of memory-footprint profiling because `cn.mops(..)`$_{\mathrm{R}}$ always created the same number of objects regardless of argument `parallel`$_{\mathrm{R}}$.

To ensure that peak memory was not due to later pipeline stages, statement `return(NULL)`$_R$ was inserted before the beginning of stage 5 to halt further processing. For accuracy, memory footprints of the following were subtracted from peak memory

Listing 5.2: Measuring Host Memory Footprint of an R Session

```
1 (shell)$ R -d "valgrind --tool=massif \
2                        --massif-out-file=/home/[user]/heap.out"
```

usage of both gcn.MOPS and cn.MOPS:

- The initialization of the R session.
- The process of loading package cn.MOPS.
- The normalized input data loaded from disk.

This was done by running the shell command shown in Listing 5.2, then executing `‘‘library(cn.mops); load("norm_grangesData.RData"); q(save="no")’’` in the R interactive shell. These adjustments were needed because *valgrind* and *massif* profiled R as a whole process instead of just profiling the script. Therefore, host memory footprints associated with the list above did not contributed to the analysis and, accordingly, were not included. Experimentally, the sum of memory footprints of the processes/data above was found to be $\sim$ 164 MB. Hence, this value was subtracted from memory-footprint results.

In addition, stages 1, 2, and 3 were not modified in gcn.MOPS and, thus, they were never involved in benchmarking and memory-footprint profiling. They were excluded by running them in a separate R session and, then, saving the output on disk. During benchmarking and memory profiling, result file from stage 3 was simply loaded from disk and processing was started at stage 4.1.

Finally, both benchmarks BM-A and BM-B were run using the script shown in lines 12-17 of Listing 2.6 unless otherwise indicated. For gcn.MOPS, interface

`gcn.mops(..)`$_{\mathrm{R}}$ was used instead of `cn.mops(..)`$_{\mathrm{R}}$ and argument `gpu`$_{\mathrm{R}}$ was set to use nVidia Tesla C2050. Further, kernel's grid and block sizes were set to 112 and 512 respectively. All computing resources were made available while ensuring that the on-disk swap memory was never used by disabling it with shell command ``` ``sudo swapoff -a'' ```. The benchmarking script was run in the R interactive shell while system's graphical environment was disabled.

5.5 Comparison with the Original cn.MOPS

Figure 5.1 presents the execution time, on logarithmic scale, of gcn.MOPS and cn.MOPS in stage 4 for BM-A. The execution time of the original cn.MOPS run-

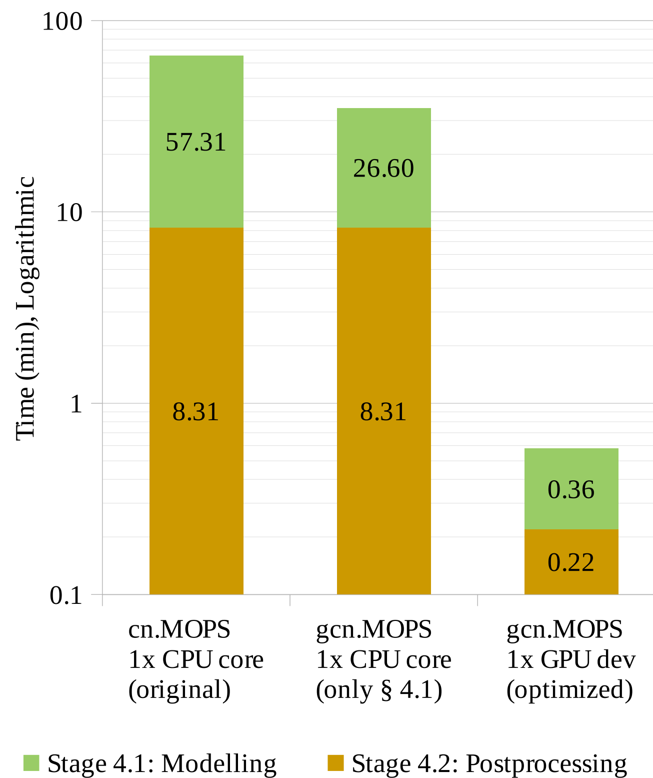


Figure 5.1: gcn.MOPS vs cn.MOPS: Execution Time in Stage 4

ning on 1× CPU core is shown in the left column. The right column shows the

execution time of the fully-optimized gcn.MOPS running on GPU. The middle column shows the execution time of gcn.MOPS given only modifications described in § 4.1 were implemented, i.e. writing branches #1 and #2 in C/C++ instead of R.

In stage 4.1, the fully-optimized gcn.MOPS achieved a speedup of 159.19× relative to the original cn.MOPS, which was substantially higher than the projected maximum achievable speedup with multi-CPU parallelism $\left(\psi_{max}^{CPU}\right)$. In chapter 3, it was shown that under the constraint that efficiency $\approx 50\%$, $\psi_{max}^{CPU} \in [8.87\times, 10.63\times]$ using $16-21\times$ CPU cores. If that constraint was relaxed by assuming parallelism overhead, as well as any other source of inefficiency, was constant ($f = f_e(p) \approx 4.88\%$), then $\psi_{max}^{CPU}(p = \infty) \lesssim 20.49\times$. For the middle column of Figure 5.1, a speedup factor of 2× was gained, relative to the original cn.MOPS, in the first phase of gcn.MOPS development merely by executing branches #1 and #2 in C/C++ instead of R. This design, described in § 4.1, would still be limited by Amdahl's Law like the original cn.MOPS as explained in chapter 3.

For stage 4.2, the fully-optimized gcn.MOPS decreased the execution time by 97% relative to the original cn.MOPS. This ratio was expected to remain fairly constant for larger problems. The reason is that in the fully-optimized gcn.MOPS, this stage was mostly limited to assigning names to rows and columns of various matrices, which could be seen as a relatively trivial operation.

In stages 4.1 and 4.2 combined, the fully-optimized gcn.MOPS had a lower host memory footprint running BM-B relative to the original cn.MOPS. The peak memory usage of the fully-optimized gcn.MOPS was 711.2 MB vs 1,726.5 MB for the original cn.MOPS. In other words, memory usage decreased by 58% or almost 1 GB in BM-B. This percentage of decrease in host memory usage was expected since results were not duplicated in stage 4.2 to change data organization like in the original cn.MOPS.

Chapter 6

Discussion

In this chapter, three technical aspects of gcn.MOPS which affected its utility and performance are discussed. First, there were numeric differences between CPU and GPU results and they are discussed in the first section. Next, variables, which gcn.MOPS was sensitive to in terms of performance, are analyzed. Finally, future implementation frameworks for extending gcn.MOPS with multi-GPU parallelism are presented.

6.1 Numeric Accuracy

To ensure that gcn.MOPS produced correct answers, results were compared to the output of cn.MOPS at the end of stage 4.2. In both `gcn.mops(..)`$_{\mathrm{R}}$ and `cn.mops(..)`$_{\mathrm{R}}$, matrices α, λ, $sini$, CN and α_{ik} and vector ini were all made globally visible using R's operator "$<<-$" as shown in Listing 6.1. In addition, both benchmarks were run using the script shown in lines 12-17 of Listing 2.6 with argument `returnPosterior`$_{\mathrm{R}}$ set to `TRUE`. Then, an R script was written which used `identical(..)`$_{\mathrm{R}}$ and `all.equal(..)`$_{\mathrm{R}}$ to verify results. The former function tests two objects for being "exactly equal" while the latter tests for "near equality" given a numeric tolerance [20]. The verification script is shown in appendix B.2.

Listing 6.1: Making Results from Stage 4.2 Globally Visible

```
# inside cn.mops(..) / gcn.mops(..)
...
# for cn.mops(..)   # for gcn.mops(..) #
########################################
cpu_L <<- L          # gpu_L <<- L
cpu_A <<- A          # gpu_A <<- A
cpu_CN <<- CN        # gpu_CN <<- CN
cpu_sINI <<- sINI    # gpu_sINI <<- sINI
cpu_INI <<- INI      # gpu_INI <<- INI
cpu_POST <<- post    # gpu_POST <<- post
return(NULL)         # return(NULL)

if (m>5){
  message("Starting segmentation algorithm...")
...
```

In both BM-A and BM-B, test for exact equality passed for CN, but failed for others because of slight numeric variations between CPU's and GPU's results. This was expected given architectural difference between CPU and GPU and due to different implementations of math libraries. Though, these numeric variations did not imply that CPU's results were more accurate than GPU's or vice versa [2]. Table 6.1 presents the numeric variations of results in stage 4.2, expressed as mean relative difference, between CPU's and GPU's results. When interpreting final results,

Benchmarks:	BM-A	BM-B
α	1.387175E−15	1.161012E−15
λ	1.395657E−15	8.426478E−16
ini	1.024378E−15	8.447883E−16
$sini$	2.630268E−15	1.909384E−15
α_{ik}	7.47812E−15	5.750321E−15
CN (strings)	identical	identical

Table 6.1: Mean Relative Difference of CPU and GPU Results

care must be taken as these numeric variations might affect the biological meaning of other model parameters at later pipeline stages. This aspect was not investigated or discussed because later pipeline stages were not within the scope of this thesis.

These numeric variations, however, did not have an impact on detected CNVs, which is matrix CN.

6.2 Sensitivity Analysis

The performance of gcn.MOPS was mostly sensitive to user argument `minReadCount`$_\mathrm{R}$. In § 4.3.5, it was assumed that the compute-intensive path of branch #1 (Figure 4.1) would always be taken so that GPU performance would improve; if the assumption turned wrong, then corrections would be made on host. Though, if `minReadCount`$_\mathrm{R}$ was set to a value such that the non-compute path was taken for all rows, then all processing done by GPU would not be useful. Hence, for a large input that was partitioned into multiple chunks and with host/device concurrency (§ 4.3.6), execution time would be bound either by CPU or GPU, whichever the slowest. Figure 6.1 shows the execution timeline of gcn.MOPS in stage 4.1 for BM-A given `minReadCount`$_\mathrm{R}$`=3525`, which was above the maximum value in BM-A after normalization. The timeline was obtained using nVidia's Visual Profiler, but it was manually redrawn to eliminate unnecessary details and to enhance visual resolution. Between the first and the last CUDA API call, processing took ∼17.5 seconds. The timeline did not include the R/C++ overhead, which was calculated as $21.6s - 17.5 = 4.1s$. Thus, if GPU was not involved and all green portions were concatenated, it would have potentially taken $10s + 4.1s = 14.1s$ for gcn.MOPS to finish stage 4.1. In cn.MOPS,

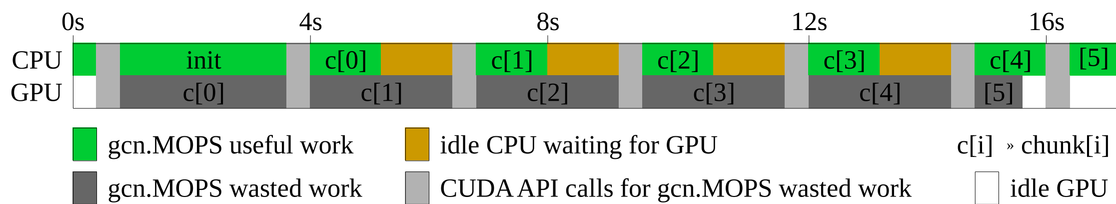


Figure 6.1: Processing Timeline (Stage 4.1/BM-A/`minReadCount=3525`)

setting $\texttt{minReadCount}_R$=3525 resulted in a ∼16.24× speedup, but in gcn.MOPS, however, the execution time remained constant as shown in Figure 6.2. Therefore, it would be fair to say that gcn.MOPS was insensitive to $\texttt{minReadCount}_R$ since it did not gain performance as $\texttt{minReadCount}_R \rightarrow \infty$ and assuming CPU was faster at doing its work than GPU.

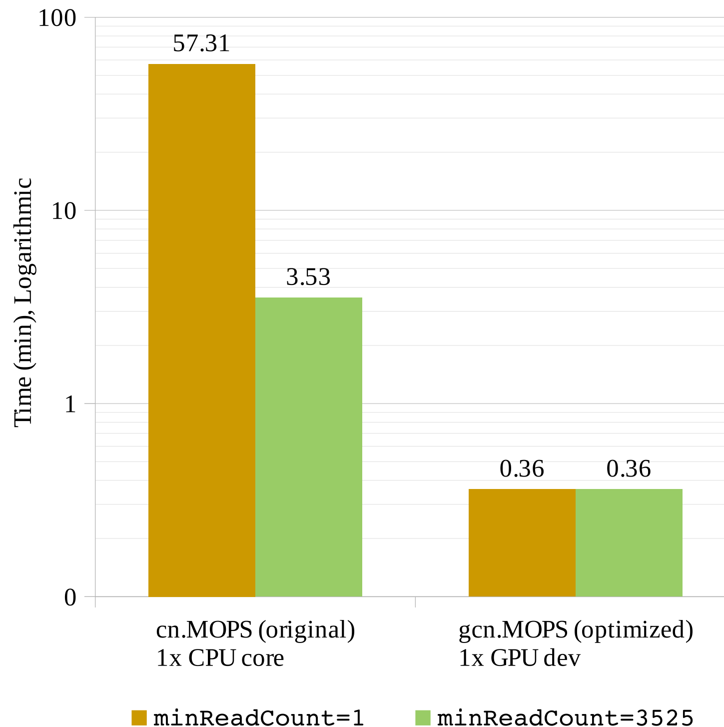


Figure 6.2: Impact of $\texttt{minReadCount}_R$ on Execution Time (Stage 4.1/BM-A)

In addition, gcn.MOPS exhibited similar sensitivity to user argument $\texttt{cyc}_R$ as cn.MOPS. This argument specifies how many times the computationally-intensive path is repeated for every row, i.e. number of cycles. In each cycle, the core mathematical model uses results from the previous cycle as initial values. Figure 6.3 shows the impact of $\texttt{cyc}_R$ if it was increased from 20 to 200 for BM-B. In cn.MOPS, the execution time was ∼5.19× slower while it was ∼5.16× slower in gcn.MOPS, which were almost similar.

However, cn.MOPS was supposed to be less sensitive to $\texttt{cyc}_R$ compared to gcn.MOPS.

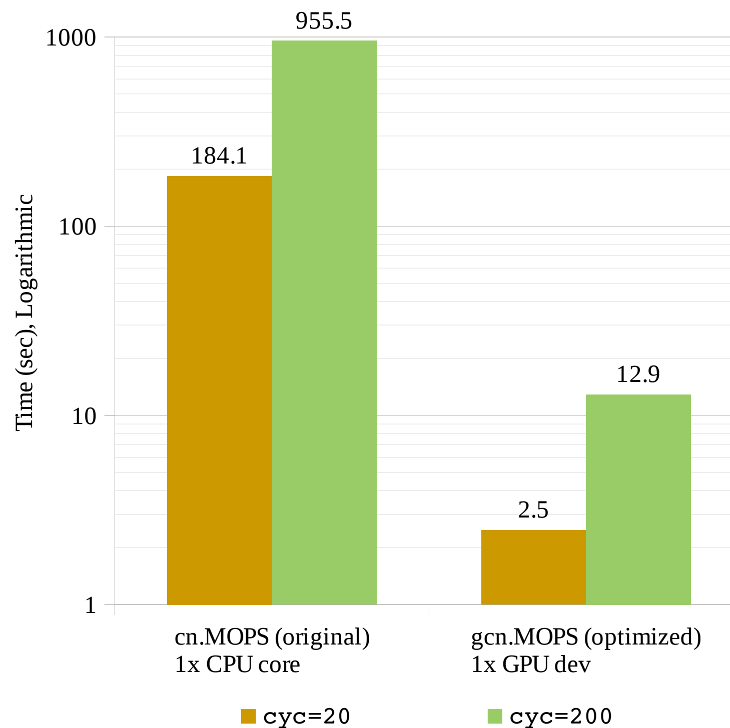


Figure 6.3: Impact of $\texttt{cyc}_{\mathrm{R}}$ on Execution Time (Stage 4.1/BM-B)

This is because unlike cn.MOPS, gcn.MOPS did not save time by taking the non-compute path more often as shown in Figure 6.2. As presented in Table 5.1, 30% of rows in BM-B caused the program to take the non-compute path given $\texttt{minReadCount}_{\mathrm{R}}\texttt{=1}$. If data in BM-B was overwritten with random numbers in the range $[2.1, 1000.1)$, then cn.MOPS, like gcn.MOPS, would always take the compute-intensive path given $\texttt{minReadCount}_{\mathrm{R}}\texttt{=1}$. In this case, the execution times of stage 4.1 for cn.MOPS became 293.9s for $\texttt{cyc}_{\mathrm{R}}\texttt{=20}$ and 1,777.7s for $\texttt{cyc}_{\mathrm{R}}\texttt{=200}$. This translated to a slowdown factor of $\sim$6.05× compared to $\sim$5.16× in gcn.MOPS. Consequently, gcn.MOPS was less sensitive to user argument $\texttt{cyc}_{\mathrm{R}}$ than cn.MOPS, which was advantageous to gcn.MOPS. Figure 6.4 summarizes sensitivity to $\texttt{cyc}_{\mathrm{R}}$ in terms of slowdown factors for both the original BM-B and BM-B filled with random numbers in the range [2.1, 1000.1).

Aside from gcn.MOPS's sensitivity to user arguments, varying grid/block sizes and DRAM were investigated. In § 4.3, grid size was arbitrarily set to 336 (24 × 14

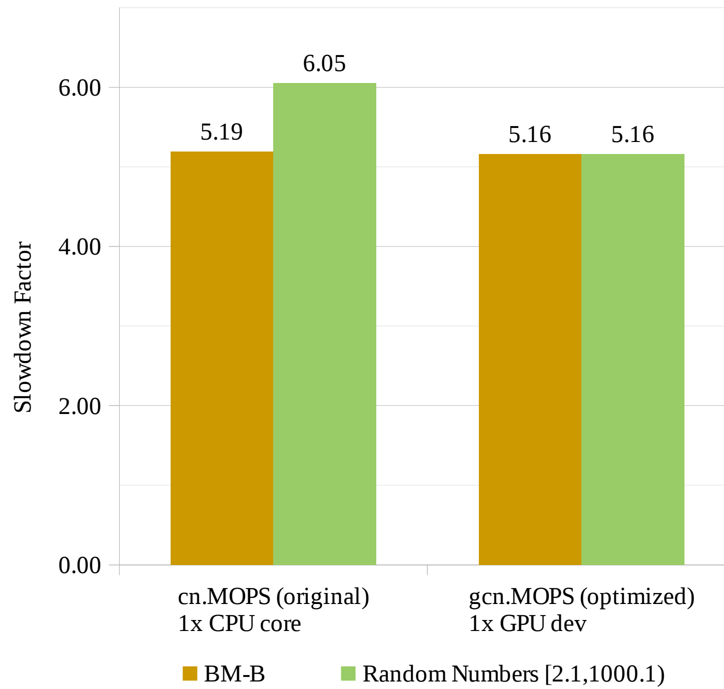


Figure 6.4: Slowdown Factors–Increasing $\mathtt{cyc}_{\mathrm{R}}$ from 20 to 200 (Stage 4.1)

SMs) while block size was set to 64 (2× warp size); these values were fixed throughout all optimization techniques. However, they were not optimal values because tuning these parameters only made sense if the kernel was fully optimized. Thus, mentioning these parameters in § 4.3 was avoided until chapter 5 in which results assumed a fully-optimized kernel. Table 6.2 presents the execution time of the kernel running BM-A for various grid/block sizes. Grid's headings are multiple of SMs while threads-per-block's headings are multiple of warp size. It was noticed that values are nearly equal within each right-to-left diagonal. What was common between these diagonals was the total number of threads in the kernel, which was calculated as in Equation 6.1.

$$\text{Total Threads} = \text{Grid Size} \times \text{Threads per Block} \tag{6.1}$$

Moreover, it was noticed that the two diagonals which started at [56,128] and [112,512] had similar average values; the latter diagonal had $2^3\times$ the number of threads in the

former. It is beyond the scope of this thesis as for what caused these patterns to appear. However, they showed that gcn.MOPS's kernel performance was sensitive to the total number of threads regardless of the individual values of grid/block sizes. For the fully optimized kernel, grid and block sizes were set to 112 and 512 respectively.

		Threads per Block			
		64	128	256	512
Grid	56	23.671s	13.566s	14.573s	17.163s
	112	13.559s	14.651s	17.206s	13.542s
	224	14.643s	17.254s	13.611s	16.466s
	448	17.253s	13.707s	16.486s	15.449s

Table 6.2: Impact of Grid/Block Size on Kernel's Execution Time (BM-A)

Finally, gcn.MOPS was not sensitive to the amount of available DRAM provided the minimum requirements were satisfied (§ 4.2.4) and that DRAM $\geq$ 256 MB. Testing for DRAM sensitivity was done by varying the amount of available DRAM for gcn.MOPS. In file Makevar, a macro named `MAX_DRAM=x` was defined, where `x` was in MB. This macro was used in function `calc_prtn_len(..)`$_C$ such that the required amount of DRAM to process a chunk of data did not exceed `MAX_DRAM`. To ensure no interference from other graphical processes, the graphics environment was disabled and testing was carried out in shell. Figure 6.5 shows a plot of execution time for various values of `MAX_DRAM`. There were three observations that are addressed as follows:

1. The performance was slightly better than average within range [256,512] MB.
2. There was no visually-noticeable trend in execution time vs available DRAM if extreme cases within range [64,128) MB were excluded.
3. The highest performance was achieved if exactly 1,792 MB of DRAM was used.

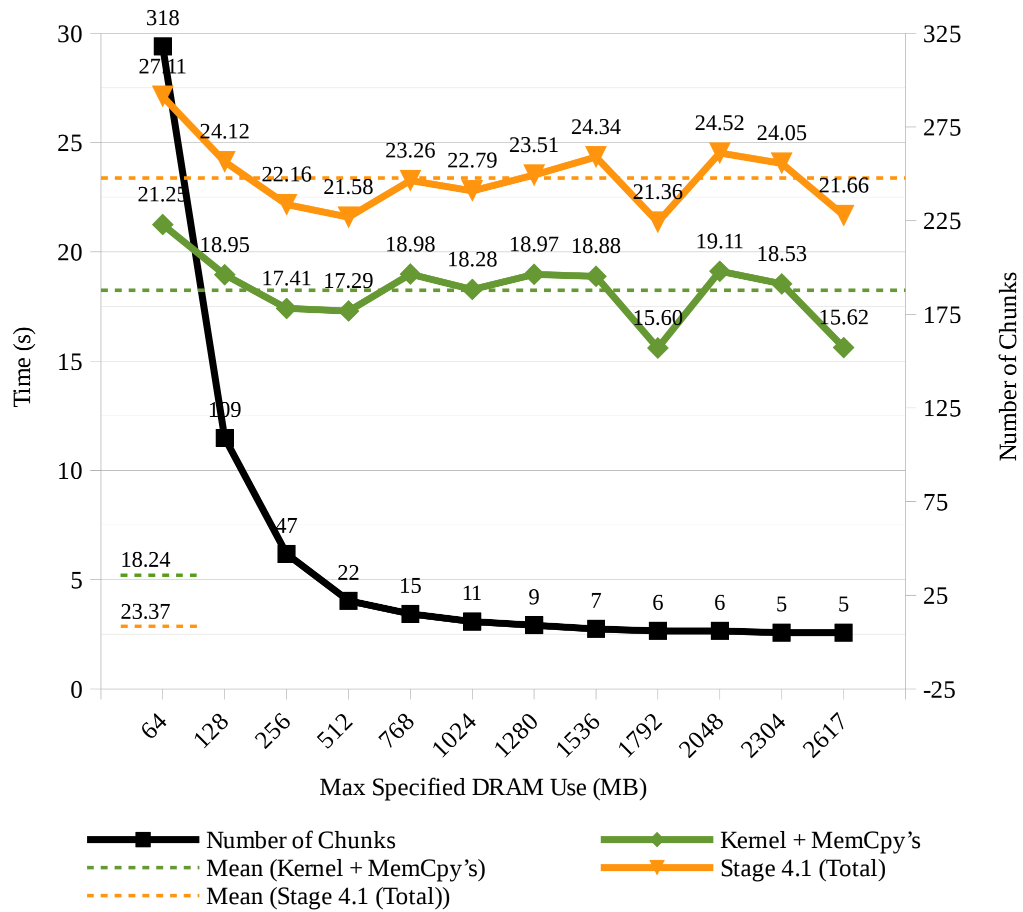


Figure 6.5: Available DRAM vs Execution Time (Stage 4.1/BM-A)

For the third point, it was not understood why limiting maximum DRAM usage to 1,792 MB resulted in a performance spike compared to other data points. The exact allocated DRAM space, obtained using `cudaMemGetInfo(..)`$_{\mathrm{C}}$, was 1,876,688,896 bytes (1,832,704 KB or 1789.75 MB). The experiment was repeated multiple times and `MAX_DRAM` was set manually for each data point, but the same behaviour reoccurred. Other than this special case, gcn.MOPS was insensitive to the amount of DRAM.

6.3 Multi-GPU Support

From a technical point of view, stage 4 of gcn.MOPS could be extended to support concurrent, multi-GPU execution. In this section, a framework that could guide future implementation is presented. In terms of work distribution, there are two scenarios that should be addressed differently:

1. Several, large and separate experiments.

2. A single, super-sized experiment.

For the first scenario, experiments are independent and, therefore, they could be run concurrently in separate R sessions/processes either locally or on a computer cluster. Assuming a set of experiments $P = \{p0, p1, p2, ..p_n\}$ and a set of GPUs $G = \{g0, g1, g2, ..g_n\}$, each experiment in P would map to one GPU in G. This solution was demonstrated with an R script, presented in appendix B.3, which used R-package *future* [1]. The package has many features which include running R statements asynchronously. Figure 6.6 shows the execution timeline of gcn.MOPS in which BM-A was concurrently run twice on two GPUs. The timeline was obtained using nVidia's Visual Profiler, but it was manually redrawn to eliminate unnecessary details and to enhance visual resolution. Additionally, the R/C++ overhead was not included in the timeline (∼4.1 seconds). It could be noticed that GPU<0> was twice

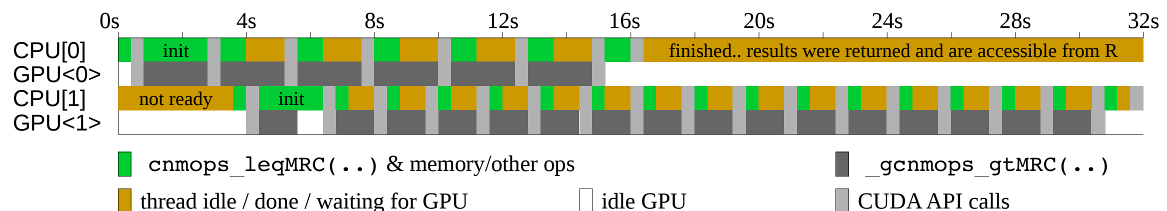


Figure 6.6: Multi-GPU Processing Timeline (Stage 4.1/2×BM-A)

as fast as GPU<1> since the former has twice as many CUDA cores as the latter. This

shows that gcn.MOPS exhibited a better scalability if given more processing power compared to cn.MOPS. Thus, execution resources per GPU should be accounted for to ensure larger experiments are dispatched to resourceful GPUs. Further, there was an overhead to bring all results into the main R session and this should also be taken into account.

In the second scenario, most of gcn.MOPS's internal C/C++ functions could be reused to achieve multi-GPU parallelism at the experiment level. Instead of having a single partition table, multiple ones could be created such that each GPU would handle one. From Figure 6.6, it is apparent that partition tables would be of different sizes depending on each GPU's resourcefulness. Then, multiple CPU threads could be spawned, e.g. `pthread_t`$_C$, such that each thread would control a GPU. Next, each thread would further partition its table into multiple chunks according to available DRAM on GPU. This could be done using functions `calc_prtn_len(..)`$_C$ and `_create_partition_table(..)`$_C$. This solution might be superior to queue-based mechanisms because it ensures that larger jobs would be assigned to GPUs with the most execution resources. Besides, it would account for available DRAM on each GPU individually.

Chapter 7

Conclusion and Future Work

In short, the modelling step of cn.MOPS, a CNV detection tool, was alternatively accelerated with GPU. The new solution, gcn.MOPS, achieved a speedup factor of 159× in the modelling step. Such a speedup factor was considerably higher than the maximum of what could be achieved with cn.MOPS, which was found to be $\lesssim 20.49\times$ given infinity CPU cores. Moreover, the execution time of the memory-intensive, data postprocessing step was reduced by 97% and was made negligible relative to other steps in cn.MOPS pipeline. In both steps combined, gcn.MOPS reduced memory usage by more than a half.

These performance levels were achieved by applying various optimization techniques to make the core algorithm fit GPU architecture. Data access patterns were changed to ensure coalesced memory accesses. This resulted in an efficient use of GPU memory and an almost-complete elimination of the data postprocessing steps. Additionally, branch divergence was made minimal by decoupling the non-compute path from the compute-intensive path of the algorithm. The former was executed on CPU and the associated execution time was hidden by host/device concurrency. Other applied optimization techniques included using constant memory and disabling

L1 cache memory. Potentially, more performance might be gained with for-loop unrolling, but this technique would impact software complexity and readability.

For the setup used, the modelling and the data postprocessing steps in gcn.MOPS accounted for $\sim$1.3% of the pipeline's total execution time as opposed to $\sim$58.7% in cn.MOPS using 1$\times$ CPU core. Thus, other pipeline stages should be investigated for GPU execution. Meanwhile, further improvements to gcn.MOPS should be limited to ensuring cross-device and cross-platform compatibility. If more performance is desired, then the suggested frameworks for multi-GPU parallelism might be considered.

Appendix A

The Impact of Branch #1 on Coalescing Memory Requests

In § 4.3.5, it was claimed that the existence of one branch-diverging thread in a warp would hurt memory-request coalescing and increase the number of executed LD/ST instructions by some factor. To demonstrate this notion, several test programs were written. The kernel and the driver program shown in Listing A.1 were used to detect whether *nvprof* counted coalesced LD/ST instructions as a single or multiple instruction; each thread in the launched kernel wrote its ID in a memory locations,

Listing A.1: Probing Metric `ldst_executed` for Coalesced Mem-reqs

```
1  #include <cuda.h>
2
3  __global__ void kernel0(int *a) {
4    int tid = blockIdx.x * threadIdx.x; //compute thread ID
5    a[tid] = tid; //coalesced memory access (store)
6  }
7
8  int main (int argc, char **argv) {
9    int grid = 1, block = 32; // kernel size (one warp)
10   int *dev_a; //device pointer
11   //allocate memory space for grid*block integers in GPU memory
12   cudaMalloc((int**)&dev_a, sizeof(int)*grid*blocks);
13   kernel0<<<grid, blocks>>>(dev_a); //launch kernel
14   return 0;
15 }
```

where all memory locations were consecutive. For this program, the value of metric `ldst_executed` was exactly 1. Thus, coalesced LD/ST instructions would count as a single instruction.

Next, a simple kernel was written, shown in Listing A.2, which roughly resembled the general behaviour of branch #1 that was shown in Figure 4.1. In this kernel, the last thread in all warps would diverge and execute a non-compute statement. If `kernel1` was launched with only 32 threads, i.e. a single warp, using the driver

Listing A.2: A Kernel which Resembled Branch #1 in Figure 4.1

```
1  #include <cuda.h>
2  #define WARP_SIZE 32
3
4  // assuming a benchmark such that every 32nd thread would diverge
5  __global__ void kernel1(int *a, int cyc) {
6    int tid = blockIdx.x * threadIdx.x; //compute thread ID
7    if(tid % WARP_SIZE == 31) // last thread in this warp
8      a[tid] = 2; // set GPU memory location to constant
9
10   else // the other 31 threads
11     for(int i = 0; i < cyc; i++) { // simple computation
12       int val = i*2;
13       a[tid] = val; // set GPU memory location to 'val'
14     }
15 }
```

program shown in Listing A.3, metric `ldst_executed` would still indicate that only one store instruction was executed. That is, memory store requests were coalesced in a single instruction despite branch divergence. This meant that GPU managed to coalesce the memory-store requests of both branch paths by delaying the memory request of one path until both paths finished execution.

Listing A.3: Driver Program for the Kernal Shown in Listing A.2

```
1  int main (int argc, char **argv) {
2    int grid = 1, block = 32; // kernel size (one warp)
3    int *dev_a; //device pointer
4    //allocate memory space for grid*block integers in GPU memory
5    cudaMalloc((int**)&dev_a, sizeof(int)*grid*blocks);
6    kernel0<<<grid, blocks>>>(dev_a, 20); //launch kernel
7    return 0;
8  }
```

However, if the number of warps was increased by varying grid and block sizes

(`grid` and `block` in Listing A.3), then the ratio of `ldst_executed` to the number of warps would no longer be 1:1. In other words, some of the memory-store requests were not coalesced. As presented in Table A.1, if the grid size is greater than one, i.e. two or more SM were utilized, then every two warps would generate an extra memory-store instruction due to branch divergence. The ratio remained 1:1 regardless of grid and block sizes only if the branch was eliminated and that the kernel executed the compute portion alone.

`kernel1` Size $<<< G, B >>>$	Warps $W = G \times B \div 32$	`ldst_executed` S	Ratio S:W	Excess Ratio (S-W):W
$<<< 1, 32 >>>$	1	1	1:1	0:1
$<<< 1, 64 >>>$	2	2	1:1	0:1
$<<< 1, 512 >>>$	16	16	1:1	0:1
$<<< 2, 32 >>>$	2	3	1.5:1	1:2
$<<< 2, 64 >>>$	4	6	1.5:1	1:2
$<<< 2, 512 >>>$	32	48	1:5:1	1:2
$<<< 14, 32 >>>$	14	21	1.5:1	1:2
$<<< 14, 64 >>>$	28	42	1.5:1	1:2
$<<< 14, 512 >>>$	224	336	1.5:1	1:2
$<<< 112, 512 >>>$	1792	2688	1.5:1	1:2

Table A.1: Mem-store:Warp Ratios of Various Grid/Block Sizes (Listing A.3)

In this simple scenario where grid size ≥ 2, a single branch-diverging thread in all warps caused the number of executed LD/ST instructions to increase by 50% (excess ration 1:2). Perhaps, this percentage of increase would vary depending on the complexity of the non-compute path and the number of warps having diverging threads. This simple case showed that branch divergence impacted memory-request coalescing. This case also explain why eliminating branch divergence due to branch #1 in gcn.MOPS decreased executed LD/ST instructions.

Appendix B

Relevant Source Codes

B.1 Contents of File “Makevar”

```
1  # NOTE: macros which are not defined here are inherited from /etc/R/Makeconf
2  # Macros ending in ‘_UD’ indicate ‘User-Defined’
3  DBG = #-DDEBUG
4  ################ SOURCES & OBJECTS
5  ALL_SRCS := $(shell find . -type f -name '*.c' -o -name '*.cu' -o -name '*.cpp')
6
7  CUDA_SRCS = $(filter %.cu,$(ALL_SRCS))
8  CUDA_OBJS = $(patsubst %.cu, %.o,$(CUDA_SRCS))
9  CUDA_DLINK_OBJ = cu_DevicLinked.o
10
11 CPP_SRCS = $(filter %.cpp,$(ALL_SRCS))
12 CPP_OBJS = $(patsubst %.cpp, %.o, $(CPP_SRCS))
13
14 C_SRCS = $(filter %.c,$(ALL_SRCS))
15 C_OBJS = $(patsubst %.c, %.o, $(C_SRCS))
16
17 OBJECTS = $(C_OBJS) $(CPP_OBJS) $(CUDA_OBJS) $(CUDA_DLINK_OBJ)
18 ################ PATHS
19 CUDA_HOME = /usr/local/cuda
20 CUDA_LIB_DIR = $(CUDA_HOME)/lib64
21 # R_HOME is already defined in /etc/R/Makeconf as ‘R_HOME = /usr/share/R’
22 ################ INCLUDE DIRECTORIES
```

```
23  CUDA_INCLUDE_DIR = $(CUDA_HOME)/include
24  # R_INCLUDE_DIR R library directory is also defined by R
25  ################ COMPILERS
26  CXX  = g++
27  CC   = gcc
28  NVCC = $(CUDA_HOME)/bin/nvcc
29  ################ COMPILER FLAGS
30  # Flags specific for the package passed to the g++ compiler directly
31  PKG_CFLAGS = $(PKG_CXXFLAGS)
32  PKG_CXXFLAGS += $(DBG)
33  PKG_CXXFLAGS += -I$(CUDA_INCLUDE_DIR)
34  #PKG_CXXFLAGS += -DMAX_DRAM=1792 #MB.. definition without a value sets it to 1 MB
35  PKG_CXXFLAGS += -DBLOCK_SIZE=512
36  PKG_CXXFLAGS += -DGRID_SIZE=112
37
38  # Flags for the host code compiler (ie gcc) which are passed by nvcc. If CUDA 7.5
39  # with gcc-4.8 is used, then we need to override the R-inherited macro 'CXXFLAGS'
40  # since it has the flags '-fstack-protector-strong' and '-Wdate-time' which are
41  # not supported by gcc-4.8. gcc 4.8 will generate error if nvcc tries to pass
42  # CXXFLAGS to gcc via the '-Xcompiler' flag.
43  # NOTE: CFLAGS and CXXFLAGS macros are defined by R build environment and if we
44  # redefine them here, the re-definition here will be overriden by whatever
45  # defined in R build environment (hence '_UD'). If security is not desired,
46  # remove '-fstack-protector-all'.
47  #CXXFLAGS_UD  +=  -g -O2 -fstack-protector-all -Wformat -Werror=format-security -
        D_FORTIFY_SOURCE=2 -g $(LTO)
48  # For CUDA 8.0, gcc-5.4 is supported and hence all flags can be passed safely
49  CXXFLAGS_UD += $(CXXFLAGS)
50  CXXFLAGS_UD += $(CXXPICFLAGS)
51  CXXFLAGS_UD += $(DBG)
52
53  GPU_CARD  :=  -arch=sm_20
54  NVCCFLAGS +=  -D_FORCE_INLINES
55  #NVCCFLAGS  +=  --fmad=false
56  NVCCFLAGS +=  -Xptxas -v
57  NVCCFLAGS +=  -Xptxas -dlcm=cg
58  NVCCFLAGS += -DUSE_CONST_MEM # let kernel use const mem for array args
59  NVCCFLAGS += -DMAX_nSAMPLES=100 # might be changed to hi/lo-er num
60  NVCCFLAGS += -DMAX_nCLASSES=100
61  ################ LINKED LIBRARIES
```

```
62 # needed by R build environment for the final linking stage
63 CUDA_LIBS += -lcudart -d
64 PKG_LIBS= -L$(CUDA_LIB_DIR) -Wl,-rpath=$(CUDA_LIB_DIR) $(CUDA_LIBS)
65 ################ COMPILATION RULES
66 # Device linking is done twice. This was done to circumvent R's restriction
67 # on how many targets should be there.
68 %.o: %.cu $(CUDA_SRCS)
69   $(NVCC) -Xcompiler "$(CXXFLAGS_UD)" $(GPU_CARD) $(NVCCFLAGS) \
70     --relocatable-device-code=true --compile $< \
71     --output-file $@ \
72     -I$(CUDA_INCLUDE_DIR) -I$(R_INCLUDE_DIR)
73   $(NVCC) -Xcompiler "$(CXXFLAGS_UD)" $(GPU_CARD) $(NVCCFLAGS) \
74     --device-link $(CUDA_OBJS)  --output-file $(CUDA_DLINK_OBJ) \
```

B.2 R Script for Verifying gcn.MOPS Results

```
1  num_tolerance = 1E-100
2  message("*** checking correctness of results ***")
3
4  message("Lambda numeric accuracy:")
5  message(all.equal(cpu_L, gpu_L, tolerance = num_tolerance,
6                    check.attributes = TRUE, use.names = TRUE,
7                    all.names = TRUE, check.names = TRUE ))
8  message("Lambda Identical? ", identical(cpu_L, gpu_L))
9
10 message("Alphs numeric accuracy:")
11 message(all.equal(cpu_A, gpu_A, tolerance = num_tolerance,
12                   check.attributes = TRUE, use.names = TRUE,
13                   all.names = TRUE, check.names = TRUE ))
14 message("Alpha Identical? ", identical(cpu_A, gpu_A))
15
16 message("expectedCN accuracy:")
17 message(all.equal(cpu_CN, gpu_CN,
18                   check.attributes = TRUE, use.names = TRUE,
19                   all.names = TRUE, check.names = TRUE ))
20 message("expCN Identical? ", identical(cpu_CN, gpu_CN))
21
22 message("sini numeric accuracy:")
```

```
23 message(all.equal(cpu_sINI, gpu_sINI, tolerance = num_tolerance,
24                   check.attributes = TRUE, use.names = TRUE,
25                   all.names = TRUE, check.names = TRUE ))
26 message("sini Identical? ", identical(cpu_sINI, gpu_sINI))
27
28 message("ini numeric accuracy:")
29 message(all.equal(cpu_INI, gpu_INI, tolerance = num_tolerance,
30                   check.attributes = TRUE, use.names = TRUE,
31                   all.names = TRUE, check.names = TRUE ))
32 message("ini Identical? ", identical(cpu_INI, gpu_INI))
33
34 message("post numeric accuracy:")
35 message(all.equal(cpu_POST, gpu_POST, tolerance = num_tolerance,
36                   check.attributes = TRUE, use.names = TRUE,
37                   all.names = TRUE, check.names = TRUE ))
38 message("post Identical? ", identical(cpu_POST, gpu_POST))
```

B.3 R Script for Processing 2 Data Sets on 2 GPUs Concurrently

```
1  # This script must be run with shell command 'Rscript'
2  library(cn.mops)
3  library(future)
4  maxObjSize = 2 #GB
5  options(future.globals.maxSize= maxObjSize*1024^3)
6
7  #load("~/path/to/input/data1.bam")
8  #load("~/path/to/input/data2.bam")
9
10 plan(cluster, workers = c("localhost", "localhost"))
11 experiment1  %<-% {
12   library(cn.mops)
13   gcn.mops(input_data1, I = c(0.025, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4),
14     classes = c("CN0", "CN1", "CN2", "CN3", "CN4", "CN5", "CN6", "CN7", "CN8"),
15     priorImpact = 10, cyc = 20, parallel = 12, gpu=0, norm = 1,
16     normType = "poisson", sizeFactor = "mean", normQu = 0.25,
17     quSizeFactor = 0.75, upperThreshold = 0.5, lowerThreshold = -0.8,
```

```
18      minWidth = 5, segAlgorithm = "fast", minReadCount = 1,
19      useMedian = FALSE, returnPosterior = FALSE)
20  }
21
22  experiment2 %<-% {
23    library(cn.mops)
24    gcn.mops(input_data2, I = c(0.025, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4),
25      classes = c("CN0", "CN1", "CN2", "CN3", "CN4", "CN5", "CN6", "CN7", "CN8"),
26      priorImpact = 10, cyc = 20, parallel = 12, gpu=1, norm = 1,
27      normType = "poisson", sizeFactor = "mean", normQu = 0.25,
28      quSizeFactor = 0.75, upperThreshold = 0.5, lowerThreshold = -0.8,
29      minWidth = 5, segAlgorithm = "fast", minReadCount = 1,
30      useMedian = FALSE, returnPosterior = FALSE)
31  }
32
33  # the following statement block until results are ready
34  experiment1$ini[0]
35  experiment2$ini[0]
```

Bibliography

[1] Henrik Bengtsson. *future: Unified Parallel and Distributed Processing in R for Everyone*, 2017. R package version 1.4.0.

[2] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. John Wiley & Sons, Indianapolis, IN, 2014.

[3] T. M. Davies. *The Book of R: A First Course in Programming and Statistics*. William Pollock, San Francisco, CA, 2016.

[4] Kyriacos Felekkis and Konstantinos Voskarides. *Copy Number Variation in Human Health, Disease and Evolution*, pages 129–138. Springer, New York, 2015.

[5] Enrique Gonzalez, Hemant Kulkarni, Hector Bolivar, Andrea Mangano, Racquel Sanchez, Gabriel Catano, Robert J. Nibbs, Barry I. Freedman, Marlon P. Quinones, Michael J. Bamshad, Krishna K. Murthy, Brad H. Rovin, William Bradley, Robert A. Clark, Stephanie A. Anderson, Robert J. O'Connell, Brian K. Agan, Seema S. Ahuja, Rosa Bologna, Luisa Sen, Matthew J. Dolan, and Sunil K. Ahuja. The influence of CCL3L1 gene-containing segmental duplications on HIV-1/AIDS susceptibility. *Science*, 307(5714):1434–1440, 2005.

[6] Illumina Inc. Illumina sequencing by synthesis (now in 3D). https://www.youtube.com/watch?v=fCd6B5HRaZ8, Oct 2016. Accessed: 19 Feb 2017.

[7] Guenter Klambauer, Karin Schwarzbauer, Andreas Mayr, Andreas Mitterecker, Djork-Arne Clevert, Ulrich Bodenhofer, and Sepp Hochreiter. cn.mops: Mixture of poissons for discovering copy number variations in next generation sequencing data with a low false discovery rate. *Nucleic Acids Research*, 40:e69, 2012.

[8] Wetterstrand KS. The cost of sequencing a human genome. https://www.genome.gov/sequencingcosts, July 2016. Accessed: 10 Feb 2017.

[9] Alberto Magi, Lorenzo Tattini, Tommaso Pippucci, Francesca Torricelli, and Matteo Benelli. Read count approach for DNA copy number variants detection. *Bioinformatics*, 28(4):470, 2012.

[10] nVidia. Branch statistics. http://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/branchstatistics.htm. Accessed: 22 May 2017.

[11] nVidia. CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 19 Mar 2017.

[12] nVidia. CUDA parallel computing platform. https://www.nvidia.com/object/cuda_home_new.html. Accessed: 23 Feb 2017.

[13] nVidia. Instruction set reference. http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html. Accessed: 30 Mar 2017.

[14] nVidia. nVidia Tesla P100 whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf. Accessed: 15 May 2017.

[15] nVidia. Parallel thread execution ISA version 5.0. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 30 Mar 2017.

[16] nVidia. Profiler user's guide. http://docs.nvidia.com/cuda/profiler-users-guide. Accessed: 30 Mar 2017.

[17] U.S. National Library of Medicine. Congressional justification FY 2015. https://www.nlm.nih.gov/about/2015CJ.html, Feb 2015. Accessed: 10 Feb 2017.

[18] Nathan Pankratz, Alexandra Dumitriu, Kurt N Hetrick, Mei Sun, Jeanne C Latourelle, Jemma B Wilk, Cheryl Halter, Kimberly F Doheny, James F Gusella, William C Nichols, Richard H Myers, Tatiana Foroud, Anita L DeStefano, and PSG-PROGENI and GenePD Investigators, Coordinators and Molecular Genetic Laboratories. Copy number variation in familial parkinson disease. *PloS one*, 6(8):e20988, 2011.

[19] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, NY, 2004.

[20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

[21] Andrea Sboner, Xinmeng Jasmine Mu, Dov Greenbaum, Raymond K. Auerbach, and Mark B. Gerstein. The real cost of sequencing: higher than you think! *Genome Biology*, 12(8):125, 2011.

[22] Renjie Tan, Yadong Wang, Sarah E. Kleinstein, Yongzhuang Liu, Xiaolin Zhu, Hongzhe Guo, Qinghua Jiang, Andrew S. Allen, and Mingfu Zhu. An evaluation of copy number variation detection tools from whole-exome sequencing data. *Human Mutation*, 35(7):899–907, 2014.

[23] Anita Thapar and Miriam Cooper. Copy number variation: what is it and what has it told us about child psychiatric disorders? *Journal of the American Academy of Child and Adolescent Psychiatry*, 52(8):772–774, Aug 2013.

[24] Lauren A. Weiss, Yiping Shen, Joshua M. Korn, Dan E. Arking, David T. Miller, Ragnheidur Fossdal, Evald Saemundsen, Hreinn Stefansson, Manuel A.R. Ferreira, Todd Green, Orah S. Platt, Douglas M. Ruderfer, Christopher A. Walsh, David Altshuler, Aravinda Chakravarti, Rudolph E. Tanzi, Kari Stefansson, Susan L. Santangelo, James F. Gusella, Pamela Sklar, Bai-Lin Wu, and Mark J. Daly. Association between microdeletion and microduplication at 16p11.2 and autism. *New England Journal of Medicine*, 358(7):667–675, 2008. PMID: 18184952.

[25] Cassandra Willyard. Copy number variations' effect on drug response still overlooked. *Nature Medicine*, 21(3):206, 2015.

[26] Seungtai Yoon, Zhenyu Xuan, Vladimir Makarov, Kenny Ye, and Jonathan Sebat. Sensitive and accurate detection of copy number variants using read depth of coverage. *Genome Research*, 19(9):1586–1592, Sep 2009.

[27] Feng Zhang, Wenli Gu, Matthew E. Hurles, and James R. Lupski. Copy number variation in human health, disease, and evolution. *Annual Review of Genomics and Human Genetics*, 10(1):451–481, 2009. PMID: 19715442.