ABSTRACT

SHARMA, SIDDHARTH. Parameter Estimation for System Biology Models on GPU Clusters . (Under the direction of Dr. James Tuck.)

In this work, we consider the parallelization of interval based parameter estimation with uncertainty propagation for non-linear ODE models. This approach is especially important in systems biology research as it addresses the uncertainty often present in data recorded from biological experiments. As systems biologists model larger systems, parameter estimation quickly becomes intractable due to the large space of possible solutions that need to be evaluated in the presence of uncertainty. Efficient parallel computing is necessary to solve these problems.

The work outlined here can be broken down into three stages and outcome of each stage can be seen as a contribution. First, the interval analysis based parameter estimation workload is characterized which helps in identifying the nature of available parallelism in the workload. Next, the insights gained from the characterization step are used to design and compare two very different approaches to estimate parameters on a single GPU. GPU approaches exploit the task level parallelism in the workload by maintaining a task-queue on the GPU. Optimizations are proposed to efficiently manage and spread workload within a GPU. Together, the first two stages enabled the development of scalable multi-GPU implementation of interval based parameter estimation, which is the third contribution of this work. The key design highlight here is a novel fully dynamic load balancing technique to share pending work between GPU task-queues over an MPI communication layer. The GPU implementations achieve a speedup of 16x on a single Fermi GPU and 156x on a cluster of 16 Fermi GPUs when compared to a 8-core CPU implementation.

Parameter Estimation for System Biology Models on GPU Clusters

by
Siddharth Sharma

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2014

APPROVED BY:

Dr. Cranos Williams	Dr. Greg Byrd

Dr. James Tuck
Chair of Advisory Committee

BIOGRAPHY

The author was born into a family of engineers. He was introduced to computers at a very young age and watched his grand father learn and apply computer programming skills in early 90s. By the time he reached high school, the author had turned into an avid gamer and a computer enthusiast. He was fortunate to go to a wonderful place, Manipal University, to pursue Electronics Engineering in 2005. During the last two years of his bachelors degree he worked part-time at a start-up called Manipal Dot Net where he was introduced to GPU programming. After graduating from Manipal in 2009, he took up a research assistant position at Image Processing lab of Seoul National University in S.Korea. After dabbling with various problems in real-time computer vision and eating lots of kimchi he left Korea in 2011 to join Computer Engineering masters program at North Carolina State University.

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my advisor Dr. Tuck for his excellent guidance and persistent help without which this thesis would not have been possible. I thank him for giving me the freedom to choose my work-style, for gracefully tolerating various mistakes that I made along the way and for always being available for discussions. I thank Dr. Byrd for helping me improve my thesis structure and pointing out ways to better express my ideas. I thank Dr. Williams for introducing me to this interesting problem statement and for his priceless feedback during presentations and team meetings.

I thank my parents and Dadiji for supporting and encouraging me through all my endeavors. I thank my girlfriend, Mridu, for always off-loading all her problems onto me so that I would never be free to realize that I might have problems of my own. Special thanks to Rishkul and Ashirwad for pretending to understand my ideas and allowing me to think out loud with them and in the process helping me give better shape to my ideas.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In recent years, the development of sophisticated experimental tools in molecular biology has allowed the acquisition of large variety of time-series data. Such data has the potential to reveal complex interactions in biological systems. This has compelled researchers from many disciplines to empirically study the data and gain a deeper understanding of how biological systems function.

One approach for characterizing such biological systems is by using mathematical models that elucidate and predict key behaviors of the system. Biological systems are multi-component reactors and can be mathematically expressed through systems of non-linear Ordinary Differential Equations (ODEs). Using these models, researchers can predict the behavior of a biological system and then experimentally test the consistency of the prediction to reject invalid hypotheses.

A variety of processes including biochemical reaction networks, genetic regulatory systems, population models and metabolic pathways have been modeled as systems of ODEs. A key challenge, however, is the lack of tools that are able to accurately identify systems biology models and explore their behavior in reasonable time frames. Finding solutions for ODEs is a vital part of exploring system behavior with ODE models. While techniques for solving ODEs is a long studied problem ([2], [36], [18], [32]), the approaches needed in the biological domain are still evolving and often differ from those needed in other disciplines. A good example of these differences can be seen in Parameter Estimation which is the focus of this work.

For this work, we assume that an ODE based model hypothesis is known, perhaps derived from a biological function or observed behavior. However, what is missing is the set of parameters that govern the model. Given an ODE, a set of discrete time experimental output measurements, and an unknown set of parameters, the goal of *Parameter Estimation* is to determine a set of parameters that produce model behavior fitting to the experimental observations.

Experimental data for biological system is available only at sparse time-points and includes

uncertainty. This is because the biological experiments that produce this data are expensive, time consuming and highly sensitive to experimental conditions. Traditional estimation algorithms that solve for a single best fitting parameter set based on curve fitting techniques [23] [14] are not applicable because fitting the model to an error prone data could greatly degrade the prediction quality. Note that the accuracy and confidence in estimated parameters is vital as predicted behavior is directly dependent on these estimates. We can instead solve for a range of parameter solutions that are consistent with all known data points within an error bound. The resultant model thus encodes the information about measurement uncertainty and describes all possible variations in model behavior with full confidence.

1.1 Motivation

Solving for a possible range of parameters has been accomplished using a global branch-and-bound search over a space that likely contains the solution. The search algorithm recursively divides and evaluates the search space and in the process prunes away inconsistent parameter sets. Such a search begs for parallelization since the evaluation of any two sub-regions in the search space is independent of each other. Prior work has examined parallelization of such algorithms on multicore processors and reported scalable speed-ups. But time required to estimate parameter intervals for typical system biology models on CPUs can still quickly grow from minutes to hours to even days depending on model complexity. Thus our work on exploring alternate parallel architectures becomes vital to enable researchers to obtain results within a reasonable time-frame.

Modern graphics processing units(GPUs) have emerged as general purpose computing platforms that can accelerate compute-heavy applications. GPUs were originally designed as many-core architectures geared to throttle embarrassingly parallel and computationally heavy graphics shaders. However, emergence of GPU programming models such as CUDA and OpenCL has exposed enough of the hardware that allows the computing society to efficiently harness large computing capabilities of GPUs for non-graphics parallel workloads. Scientific workloads such as ours tend to be arithmetically intensive. GPU architecture promises to be a better fit for compute heavy parameter estimation problem. This is because, unlike a CPU core, a GPU core dedicates majority of chip area to integer and floating point compute units and has a simpler control unit. This allows GPUs to achieve floating point throughputs that are orders of magnitude greater than that of modern CPUs. Moreover, GPUs give significantly better GFLOPs/watt and GFLOPs/dollar compared to CPUs making GPU computing more power efficient and cost effective.

GPUs are thus fast becoming the de-facto architecture for cost efficient high performance computing and that makes them an obvious choice as target platform for solving compute heavy

parameter estimation problem for complex biological models.

1.2 Challenges and Contributions

Our work introduces parameter estimation as a parallel application design problem to the GPU computing community. We characterize the behavior of the parameter estimation algorithm and examine techniques to map it on to a heterogeneous distributed cluster with multiple GPUs. We show that the branch&bound based solution space search exhibits task level parallelism but tasks may have different run-lengths. Moreover, the algorithm is highly irregular since task may generate new tasks leading to unpredictable workload at runtime. Through workload characterization we demonstrate that irregularity in available parallelism and uneven task run-length are the two key complexities to be addressed in order to efficiently utilize single and multiple GPU platforms. The irregular workload thus require an efficient task scheduling scheme within a GPU and a balanced task distribution scheme across multiple GPUs.

We explore two widely different execution paradigms for our single GPU implementation and present optimizations for both. A key design optimization to improve GPU occupancy by reducing GPU warp divergence is also presented. Using a microbenchmark that generates artificial workloads with pre-defined irregularity, we compare and evaluate the performance of the two single GPU approaches for workloads with varying degree of irregularity. The optimized single GPU implementation achieves an average speed-up of 16x on a Fermi GPU compared to an optimized 8-core CPU implementation. Single GPU implementation is extended to multiple GPUs by adding two software layers to achieve load balancing across multiple GPUs. The software layers can be summarized as follows:

- **A host-device communication stack.** We develop a novel software technique that allows host to access live task-lists on GPU in a synchronous manner. The technique makes use of two additional set of queues and queue status flag that is maintained by both host and device.

- **An inter-host communication stack.** For exchanging pending work between host nodes we develop a master-slave load balancer using Message Passing Protocol (MPI). A key decision here was to use non-blocking MPI procedures for the load balancer. This enables load sharing to happen in parallel with the core compute.

Live task queue management coupled with the master slave work sharing scheme forms the basis of our fully dynamic load balancer. We will show that on a 16-GPU system our fully dynamic load balancer performs 1.8x faster than a well-designed static load balancer. Ultimately, we'll show that parameter estimation for a relatively small 4-state model that requires more than a

week to process on a 8-core processor can be completed within an hour using a heterogeneous cluster with 16 Fermi GPUs giving an overall speedup of 156x over an 8-core CPU.

Our findings are relevant to the parallel programming community and to the larger systems biology community which needs efficient parallel algorithms.

1.3 Organization

In chapter 2 we give details about the problem of parameter estimation. In chapter 3 we present a characterization of the workload and describe techniques to map the workload onto a single GPU. Chapter 4 compares the two GPU approaches developed in previous chapter. It then discusses main results and optimization benefits. In chapter 5 we discuss and evaluate our static and dynamic load distribution approach which allows us to scale our workload across multiple GPUs. In chapter 6 we provide additional discussion about the algorithmic and mathematical limitations of the parameter estimation approach that often leads to very long run-times for some models. We conclude with future work in chapter 7.

Chapter 2

Background

In this chapter we formally define the Parameter estimation problem. We talk about the importance of bounded error estimation in the context of system biology modeling. We then provide a background on a branch and bound type algorithm named SIVIA that is used to perform parameter estimation in a bounded error context. Some of the key computational and algorithmic characteristics that make SIVIA a challenging problem to parallelize are also highlighted. We end the chapter with a brief summary of the GPU architecture and the CUDA programming model.

2.1 Problem Statement: Parameter estimation

The first step to predict the behavior of a biological process is to create a model that closely mimics the real system. Prior information about the interaction mechanisms governing the system is often limited. This may result in multiple model hypotheses. All possible hypotheses must be experimentally tested for validity and the process is formally known as *Model Identification*. Figure 2.1 shows the cyclical experimental methodology being undertaken today for identifying most representative hypothesis. Unknown parameters for each possible model are first estimated using the existing experimental data. Models are then simulated with known initial conditions to generate model state estimates. State estimates are compared against experimentally measured states. Hypotheses that lead to inconsistent state estimates are rejected. At the end of the cycle new set of experimental data is produced and is used to validate remaining hypotheses. Once a mathematical model is determined with considerable confidence it is used to simulate and predict system behavior.

A model hypothesis is expressed as a set of Ordinary Differential Equation (ODE) which has the following form:

$x(t)$ is a time-dependent function that describes a state of the system, p is the model

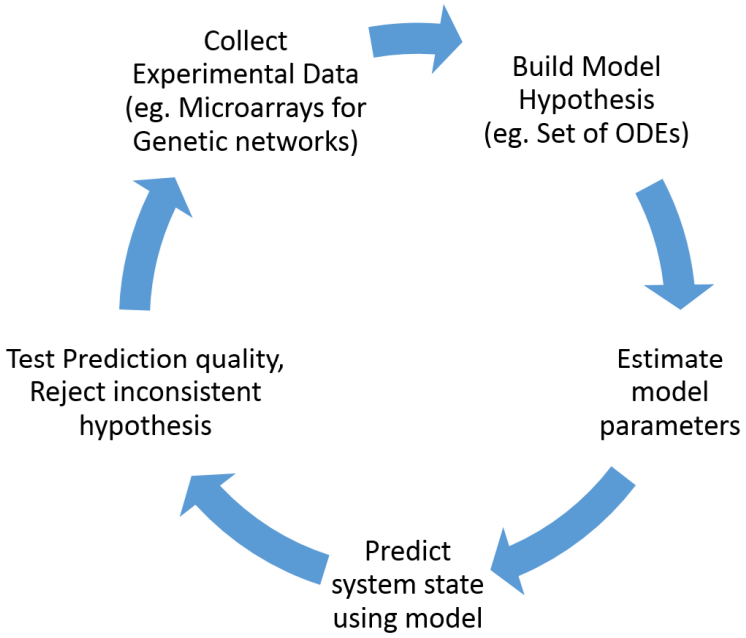

Figure 2.1: Model identification flow

parameter set and $y(t)$ is model output. Future values of $x(t)$ are determined by its derivative with respect to time, $x'(t)$ for all time points, $t > t_0$, where t_0 is a time value at which $x(t_0)$ is known. Note, while this is a single state system, multiple state systems are common. In Eqn. 2.1, f and g are parameterized functions in terms of $x(t)$. For example, consider that $f(t, x(t)) = (a - bt)x(t)$ and $g(t, x(t)) = x(t)$. In this case, a and b form the parameter set p of the model.

$$\left.\begin{aligned} x'(t) &= f(t, p, x(t)), \quad t > t_0 \\ y(t) &= g(t, p, x(t)), \\ x(t_0) &= \eta \end{aligned}\right\} \quad \text{Model Eq.} \tag{2.1}$$

Given the model, parameters and an initial condition, the values for $x(t)$ can be projected using Newton's method. We assume a small enough step, h, such that $x(t_0 + h) = x(t_0) + h \times x'(t_0)$. If h is small enough, then the estimate for $x(t_0 + h)$ should be close to the real value because the derivative will likely not change much over that range. By repeating the process, a full time series can be created. However, for some models, this method is not accurate enough so $x'(t)$ is replaced by a Taylor series expansion which better approximates the derivative over the time interval.

Some of the model parameters in Eqn. 2.1 may be known constants or can be experimentally determined. The rest of the unknown parameters have to be mathematically predicted. Empirically, we could pick a possible parameter value set arbitrarily and then produce a time series using Newton's method. If the produced time series matches the original closely enough, we can adopt it as the solution. By exploring a suitable set of possible parameters, we should be able to find ones that closely match the original data.

Hence, parameter estimation is usually solved as an optimization problem wherein a cost function is minimized using global optimization methods [23, 14]. Global optimization approaches need good initial estimate for parameters.

2.2 Parameter Estimation Applied to Uncertain Experimental Data

Estimating parameters for biological systems poses additional challenges since time series data collected through experimentation are inherently uncertain. Apart from measurement errors, the uncertainties may arise due to extrinsic reasons such as fluctuation in concentration and reaction rates [6], [38]. Moreover, intrinsic variations, for example cell-to-cell variations [3] and difference in cell history [21], may also lead to uncertainties. Discussions related to noise in gene regulatory networks and its implications on modeling can be found in [16], [31]. Hence, applying traditional techniques on uncertain experimental data to look for a unique parameter set may result in incorrect model predictions. Further, many model parameters are structurally non-identifiable and can only be bounded by a range of values based on biological considerations [35].

To compensate for the data uncertainty, instead of finding a unique best fitting parameter set, bounded error estimation techniques attempt to find all possible sets of parameter vectors (P_o) that are guaranteed to be consistent with the experimental data within a given error bound. The comprehensive set of consistent parameters in turn can produce a range of state predictions for each future time point guaranteeing the inclusion of actual system state. Moreover, variation in parameter vectors for a given range of uncertainty also provides information about the model prediction sensitivity to noise in measurement data.

Formally, if a set of known range of experimental uncertainty ϵ_i $[\epsilon_i^-, \epsilon_i^+]$ and a set of experimental data $y(t_i)$, where $t_i \in$ set of known measurement times, is given then a parameter set $p \in P_o$ and is consistent for all $i = 1 \ldots N$ if :

$$\epsilon_i^- \leq y(t_i) - y_m(p, t_i) \leq \epsilon_i^+ \tag{2.2}$$

Where $y_m(p, t_i)$ is the model output at time t_i with parameter set p and P_o is the consistent parameters set.

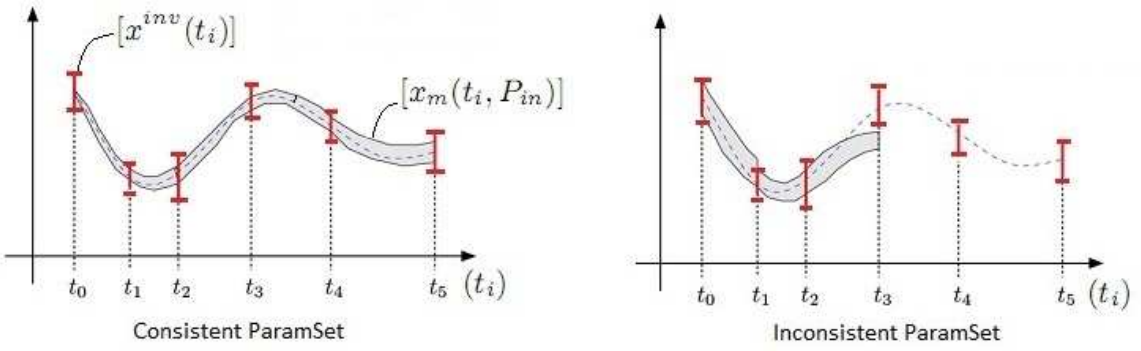

Figure 2.2: Example cases for Consistent and Inconsistent P_{in} [13]

Standard methods to approximate ODE solutions are not applicable since they work on point solutions not interval solutions. Guaranteed integration methods [27], [26], [36] based on interval analysis [10] are used instead. A parameter region $[P_{in}]$ can thus be tested for consistency in a bounded error context based on:

$$[y_m(t_i, P_{in})] \subseteq [y(t_i)] \tag{2.3}$$

where $[y_m(t_i, P_{in})]$ is the guaranteed enclosure of model output at time t_i for parameter range P_{in} and $[y(t_i)]$ is experimental output bounded by error interval $[\epsilon_i]$ at measurement time t_i. Figure 2.2 plots experimentally measured state $[y(t_i)]$ and the estimated state $[y_m(t_i, P_{in})]$ for a given $[P_{in}]$ across time.

2.2.1 Set Inversion Via Interval Analysis (SIVIA)

SIVIA proposed by Jaulin [11] combines interval set estimation techniques with a Branch & Bound type solution space exploration algorithm to recursively search for consistent parameter sets within a parameter space P_{in}.

Evaluation of P_{in} is a two stage process. The *Prediction* stage uses Guaranteed integration techniques based on Taylor expansion combined with Extended Mean Value algorithm [34] to compute a conservative over-approximation of the model state $[x_m(t_i, P_{in})]$ at t_i originating from initial condition $[x_m(t_{i-1}, P_{in})]$.

The discrete time data $[y(t_i)]$ is mapped onto the model state-space as $Y^{inv}(t_i)$ using a non-minimal inverse function $G([y(t_i)], P)$. Guaranteed set-inversion technique developed by Jaulin [11] is used to formulate $G([y(t_i)], P)$ from model output function $g(x(t), p)$ in Eqn. 2.1.

The *Pruning* stage checks for overlap between predicted state $[x_m(t_i, P]$ and experimental

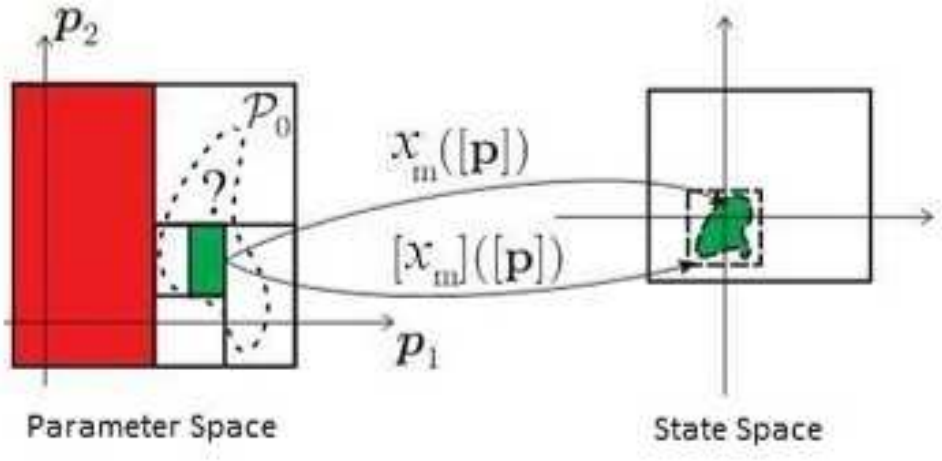

Figure 2.3: Example showing Bounded error state approximation for $[P]$ [13]

state $Y^{inv}(t_i)$ at each t_i. P_{in} is classified as *Feasible* if the model prediction $[x_m(t_i, P]$ remains Consistent with $Y^{inv}(t_i)$ for all i. P_{in} is evaluated to be *Infeasible* if at any time point there is no overlap between the two sets, indicating the absence of any consistent parameters in P_{in}. No exploration is carried out for the infeasible P_{in} beyond this point. Parameter regions that do not fully satisfy either of the above conditions are classified as *Indeterminate*. *Indeterminate* regions are bisected to produce smaller parameter regions and evaluated again until a minimum edge size of ϵ is reached. An *Indeterminate* region satisfies the following two conditions:

$$\left.\begin{array}{rll} \cap & [Y^{inv}(t_i)] & is\ not\ \emptyset\ for\ all\ i \\ [x_m(t_i, P_{in})] \not\subset & [Y^{inv}(t_i)] & for\ atleast\ one\ i \end{array}\right\} \quad \text{Indeterminate condition} \tag{2.4}$$

The SIVIA method thus describes the branch-&-bound methodology of taking larger parameter spaces and dividing them into smaller ones until feasible regions are found. Worst case bound on such an algorithm grows exponentially with the number of unknown parameters since each additional unknown parameter adds a new dimension to parameter solution space which in turn adds to the number of ways a region may be bisected.

Computational Upshot. For bounded error compute, SIVIA uses intervals to perform all computation rather than scalar values. Intervals are always specified using a [min,max] pair and thus increase burden on the memory hierarchy of a compute machine. For example, an interval pair would require double the number of hardware registers which is a scarce resource on GPUs. Furthermore, when operating on intervals, it's necessary to determine the min and

max of the new interval created by adding/subtracting or multiplying/dividing them [10]. This increases the number of computations and hardware registers used to hold intermediate results. For example, multiplication of two intervals $[p]$ and $[q]$ results in four multiply, and three sets of min and max operations as shown below:

$$\begin{aligned}[p] \times [q] \quad =&[min(p^-.q^-, p^+.q^-, p^-.q^+, p^+.q^+) \\ & max(p^-.q^-, p^+.q^-, p^-.q^+, p^+.q^+)]\end{aligned}$$

2.3 GPU architecture features

In this section, we give a brief overview of nVidia GPU architecture [42] on which we implement our algorithm. The shader compute units of GPUs are formed by a set of streaming multiprocessors(SM). Number of SMs on a machine vary based on the machine type and typically stays between 13-16. Each SM constitutes of a group of processing elements (PE) or CUDA compute cores. The number of cores on a SM may vary across architectures - for eg. Fermi GPU architecture has 32 where as more recently released Kepler architecture has 192 cores per SM. The compute cores execute threads of vector instructions in a SIMD(single instruction multiple data) fashion. A set of 32 threads that run in this fashion is called a *warp*. Each SM has a thread instruction scheduler that maintains separate program counters for a maximum of 48 warps at a time. The scheduler picks up the next warp to be executed from a pool of active warps based on a proprietary warp scheduling policy and issues the next instruction of that warp. This instruction is executed in parallel on a group of 32 PEs. Active warps are warps that do not have any threads blocked on previous instruction. Note, that the granularity of instruction scheduling is a *warp* of 32 threads and not a single thread. This effectively means that the threads in a *warp* must always execute in lock stepped fashion. If different threads within a warp need to traverse different execution paths say due to a conditional branch instruction then the warp is said to be divergent and the diverging paths are executed sequentially. Divergence hurts hardware utilization since some part of the SIMD execution pipeline is always left unused when divergent parts of the warp are being executed.

Thread warps are logically grouped as thread blocks which are assigned for execution onto a SM by a GPU-wide block scheduler. Each SM has a pool of resources shared among all the active thread blocks. Among these is a large register file and a 64Kb of read-write cache that may be configured as user managed shared cache or standard L1 cache. Threads within a block may communicate with each other via shared memory. Each thread is allocated a maximum of 63 registers on Fermi and 263 registers on kepler which often acts as a limitation as seen in the later chapters.

Threads in a block can communicate with threads from other block or the CPU via a

GPU DRAM main memory called Global memory. Global memory is connected to the SMs via a shared L2 cache and to the CPU via PCI-E link. Since CPU and GPU do not share a common address space, users must invoke explicit library calls to transfer blocks of data to or from GPU's main memory. PCI-express bus link is comparatively slow, hence transfers should be kept to a minimum for good overall performance. Global memory also suffers from a high latency and applications with high memory bandwidth requirement generally rely on large number of parallel warps to partially hide this latency. Thus applications must have the right balance between memory and compute instructions to avoid memory bottlenecks.

Chapter 3

Designing a parallel implementation

Classification of a parameter region across multiple time-stamps with SIVIA is essentially sequential. Model prediction from time t_{i-1} is used as input to predict model state at t_i. However, the process of classifying any two parameter regions is independent of each other. Thus, SIVIA algorithm exhibits task level parallelism with a natural unit of parallel task - evaluation of a parameter sub-region P_{in}. The interval ranges of unknown parameters constituting P_{in} are the inputs to such a task. Identification of an *Indeterminate* region leads to creation of new tasks that can be stored in queue like data structures. These queued up tasks amount for the available parallelism in the algorithm. Figure 3.1 illustrates the basic workflow for evaluating parameter regions using SIVIA.

In the following section we introduce two popular system biology models. These are used to understand the workload and evaluate our implementations throughout this thesis. To efficiently map our workload onto a manycore GPU architecture we characterize the task creation and execution pattern of the workload in section 3.2. We employ the understanding gained from the characterization step to propose two techniques for mapping the SIVIA algorithm on to a GPU system in section 3.3. The chapter is concluded with section 3.4 that describes a set of design optimizations for the proposed GPU implementations.

3.1 Biological models

This section describes two biological processes modeled as set of Ordinary Differential Equations. Each model consists of a set of parameters and states. To generate an estimation problem we assume a subset of model parameters to be unknown and bound them with an initial estimated interval. Initial values of model states, values of known parameters and experimental state data at discrete time points are additionally required to complete the description of an estimation problem. These estimation problem sets serve as close subsitutes to real world esti-

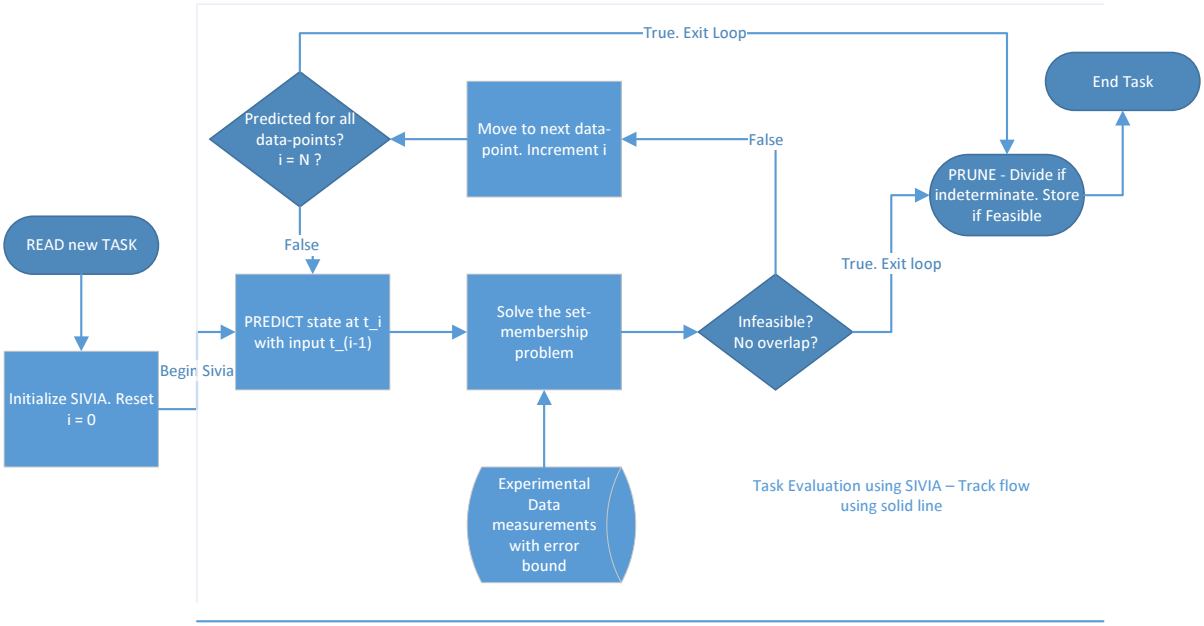

Figure 3.1: Parameter region evaluation using SIVIA

mation problems and are used to evaluate the correctness and performance of various parallel implementation techniques that we develop.

3.1.1 Lotka-Volterra prey-predator model

This is a 4 Parameter - 2 State model that captures the relation between prey and predator populations (given by x_1 and x_2 repectively) using following ODEs:

$$x_1'(t) = x_1(t)(p_1 - p_2 x_2(t)) \tag{3.1}$$
$$x_2'(t) = -x_2(t)(p_3 - p_4 x_1(t)) \tag{3.2}$$

where p_1 is the prey birth rate, p_2 is the fall in prey population due to predators, p_3 is the predator death rate and p_4 represents the growth in predator population. The experimental data points, taken from [20], consisted of 1400 data points measured at a time step of .0005. The experimental setup detailed in [34] had $p_1 = 1$, $p_2 = 0.01$, $p_3 = 1$, $p_4 = 0.02$ with initial states $x_1(0)$ and $x_2(0)$ being 50 each. An error of 1.5 was added to x_1 to realize bounded error measurements.

For experimentation, p_1 and p_3 were assumed to be known with bounded error values of p_2 and p_4 being the final result of our compute. The initial parameter intervals for p_2 and p_4 were

both [-1,1] and the minimal allowed size of intermediate param interval was given by epsilon = .00001.

3.1.2 SEIR model

This is a 4 Parameter - 4 State model that describes spread human infectious diseases as described by the following ODEs -

$$x_1'(t) = p_1 N - (p_1 + p_2)x_1(t) \tag{3.3}$$
$$x_2'(t) = p_2 x_1(t) - (p_1 + p_3)x_2(t) \tag{3.4}$$
$$x_3'(t) = p_3 x_2(t) - (p_1 + p_4)x_3(t) \tag{3.5}$$
$$x_4'(t) = p_4 x_3(t) - p_1 x_4(t) \tag{3.6}$$

where p_1,p_2,p_3,p_4 are birth/death rates, ratio of contacts by total population, average latent period of disease and recovery rate respectively. States x_1 corresponds to susceptible population, x_2 is exposed, x_3 is infectious and x_4 relate to recovered population. The dataset of 1000 points with time step of 0.01 is used with an error bound of 5. Measured settings for (p_1,p_2,p_3,p_4) are (0.5,0.1,0.4,0.2). For experimentation, p_1 was assumed to be known and initial conditions were the same (700,0,300,0).

3.2 Workload Characterization

The section is organized into two parts. First sub-section characterizes the irregularities in task creation pattern and the second half describes uneven task sizes.

3.2.1 Irregularity in available parallelism

A key property of a branch and bound type algorithm, such as SIVIA, is the inherent irregularity in the workload. Understanding and characterizing the source of irregularity is an important step in designing optimal parallel execution and load sharing strategies for a single or multi-GPU implementation. We show that task creation at runtime can result in a workload with largely varying available parallelism across time domain and solution space domain.

Workload density in solution space domain. The SIVIA algorithm searches for feasible parameters by sub-dividing indeterminate parameters into new tasks. The search trajectory cannot be predicted and leads to a highly irregular distribution of tasks in the parameter space. The task creation pattern can be represented as an an unbalanced tree as shown in Figure 3.2. Each *Feasible* and *Infeasible* region forms an end node and *Indeterminate* regions give birth

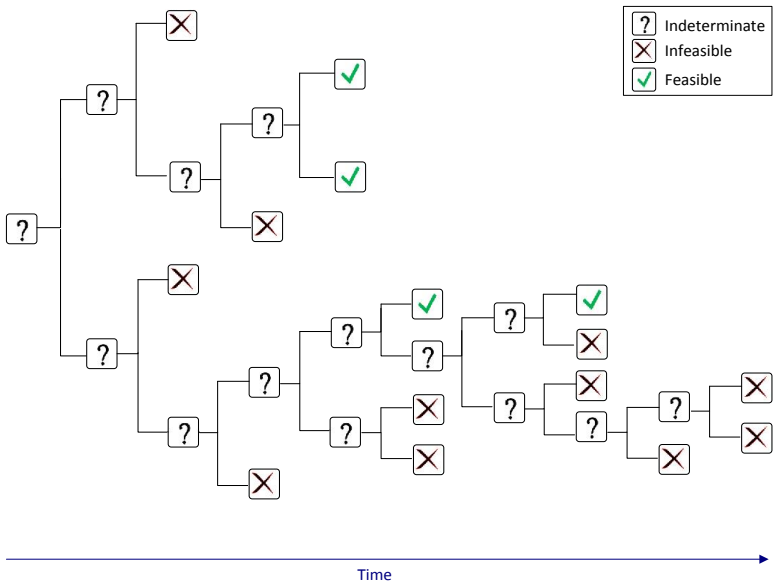

Figure 3.2: Unbalanced task creation tree

to two new child nodes. For an ODE system with two unknown parameters, one may imagine the final distribution of parameter sub-regions in 2D space to be similar to Figure 3.3. The figure illustrates the varying density of parameter sub-region in the solution space. The areas with high density of sub-region boxes are highly likely to consequently have large compute requirement. This irregularity poses interesting challenges when spreading the workload within the GPU and across multiple GPUs as there is no apriori notion for compute requirement in data-space. For example, evenly allocating compute units to search for solutions in different regions of parameter space at the start of the search process would always result in a highly imbalanced load distribution.

Temporal variation in available parallelism. Along with irregular compute density in spatial domain, the workload exhibits a varying degree of available parallelism across time. As evaluation of one P_{in} is independent of the other, each newly generated P_{in} adds to the available parallelism. For a SEIR model test setup with two unknown parameters, Figure 3.4 shows the available parallelism across simulation time in terms of number of uncharacterized regions (P_{in}). Task creation rate roughly dominates the first half of the execution since the solution space gradually unfolds with detection of numerous *Indeterminate* regions. The algorithm effectively converges in the second half as most regions reach the minimum allowed size and can either be characterized as *Feasible* or *Infeasible*. Note that higher ϵ value accelerates the convergence of the algorithm at the expense of coarser search granularity. We use this property in Chapter 5 The figure also shows that the available parallelism is limited in the initial and final stages of

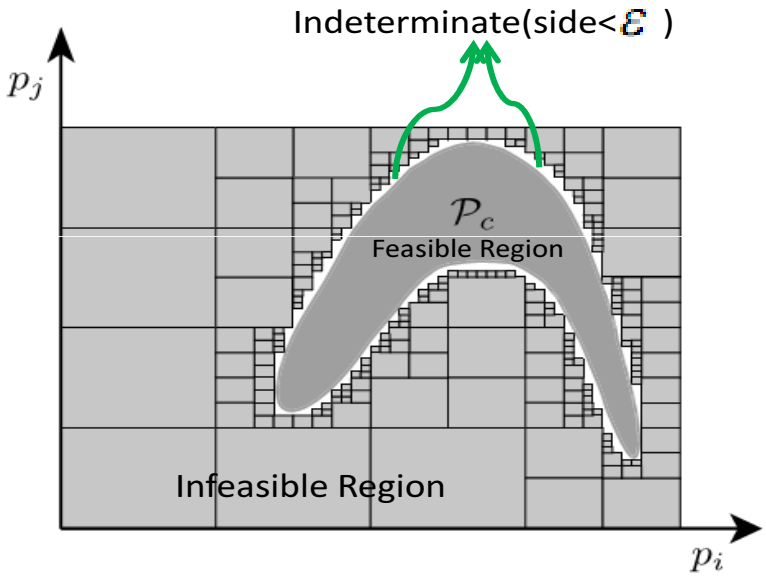

Figure 3.3: Spatial distribution of parameter sub-regions [37]. Light grey region represents infeasible regions. Dark grey region form the final feasible parameter set.

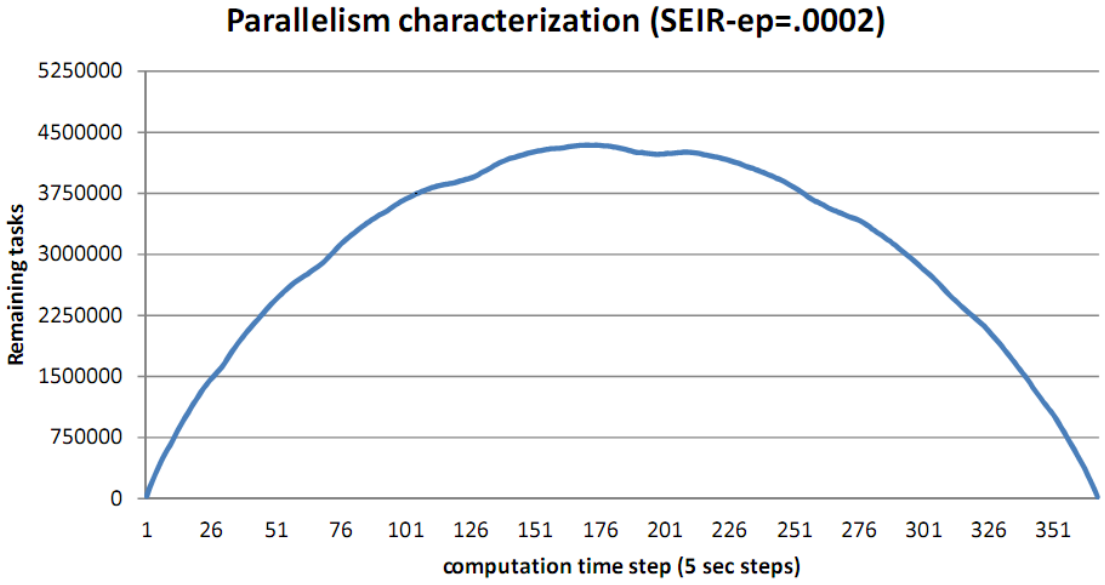

Figure 3.4: Available parallelism: Growth pattern of task list

```
// SIVIA Pseudo code
func_sivia():
Inputs: P  (Parameter interval vector),
        X_0 (Initial state at t=0),
        Y  (Error bounded experimental data)
Output: class_P (Classification of P - Feasible or
                 Infeasible or Indeterminate)

while (j < N) do          <-- Looped execution
        t= t+dt
        // Interval prediction at next t = t+dt
        X_j = Guaranteed-Set-Integration(f,X_(j-1),P,t)

        //Compare predicted state at 't' to experimental data at 't'
        if ( overlap (X_j,Y_j) = Empty Set ) // no overlap
            class_p = Infeasible;
            exit while loop;      <-- Early Exit from loop!
        }

        if (class_p != Indeterminate)
        {
            if(inclusion(X_j,Y_j) = TRUE) //X_j lies within Y_j
                class_p = Feasible
            else
                class_p = Indeterminate
        }
        j = j+1
end of while
if (class_p = Indeterminate)
    if(long_end(P) > epsilon)
        bisect P
end.
```

Figure 3.5: Pseudocode for SIVIA

the execution and we name them as *Start-up* and *Cool-down* periods.

3.2.2 Varying task run-length

The amount of compute required to complete a region evaluation task varies greatly depending on input parameter region and is not known apriori. A region is determined as *Feasible* or *Indeterminate* only after solving the set-membership problem for all t_i where $i = 1...N$. However, a parameter region may get classified as *Infeasible* at any time point for which there is no common region between experimental state interval set and predicted state interval set. The *Infeasible* region is not evaluated further and the execution thread makes an *Early Exit* from

```
main ():
    read_experiment_data( dataPoints )          //cpu
    read_initial_search_space ( task_list )     //cpu
    init_gpu_mem ( dataPoints  )                //cpu -> gpu

    while ( task_list_size is not ZERO )
    {
        init_gpu_mem ( task_list )              //cpu -> gpu
        compute_kernel_config ( task_list_size)
        run_GPU_kernel ( task_list, new_list )  //gpu
        read_gpu_data ( new_list )              //gpu -> cpu
        clear task_list
        swap ( new_list, task_list)             //cpu
    }
    transfer_final_result                       // gpu -> cpu
```

Figure 3.6: Pseudocode for naive GPU implementation
Here *task_list* is a read-only list and *new_list* is shared globally and requires access synchronization via locks.

the evaluation loop. Figure 3.5 presents a pseudo code for parameter evaluation using SIVIA and highlights the *Early Exit* condition that causes variation in task run-length.

On a GPU architecture, where multiple threads execute together in a thread warp, threads that exit early have to wait for rest of the threads in the warp to finish evaluation and reach a common convergence point. Such a situation, also known as Warp Divergence, can result in high underutilization of GPU hardware.

3.3 Design of a Single GPU implementation

On a CPU, the problem of parameter estimation is best expressed as a coarse grained parallel workload with each thread responsible for independently classifying a parameter region. Marvel [20] designed a Pthreads [17] based multi-core implementation to accelerate bounded error parameter estimation using SIVIA. The design is built on a task queue based execution model. CPU threads running SIVIA deque tasks from a task queue which is protected by a global lock. Threads utilize the available core to perform evaluation and newly created tasks are added back to the task queue. Marvel's work demonstrated linear speed-up with each additional CPU core.

Prior efforts on mapping irregular workloads onto GPU suggest two possible execution schemes. The problem can either be expressed as a standard data-parallel problem or a task parallel problem [39]. The approaches typically differ in the way the GPU consumes the newly generated tasks. The following subsections develop and explore these two approaches in further

detail.

3.3.1 Multiple kernel call approach

A standard GPU kernel has a large number of threads that perform data-parallel tasks statically distributed among them at kernel launch. A thread pool is maintained by the GPU thread scheduler which sequentially schedules threads onto GPU cores for execution. GPU threads are programmed to retire once they finish work on data initially subscribed to them thus allowing next set of threads to begin execution. GPU programming models provide poor support for algorithmic paradigms showing irregular parallelism. The only traditional way to consume tasks produced at runtime is to wait for the kernel pass to finish. The CPU can then invoke a new kernel with enough threads to consume work produced in the previous kernel pass. We call this the *Multi-kernel* approach. This is a simple data parallel expression of the algorithm in which work is statically distributed among threads at the start of the kernel.

Since GPUs do not guarantee data consistency across two kernel calls, the task data generated in the current kernel run needs to make an unnecessary roundtrip to the CPU incurring data transfer overheads. Note that a complete run of Parameter Estimation problem typically produces very large number of tasks and the size of task data grows linearly with each additional unknown parameter. The data transfer roundtrip can thus add a considerable amount to the total runtime. However, we acknowledge that more recent versions of CUDA programming model provide ways to transfer data between CPU-GPU asynchronously allowing data transfer to overlap the kernel execution. Thus a dual buffer coupled with asynchronous memory copy instructions can effectively remove the CPU-GPU data transfer bottleneck. An outline of this approach is shown in Figure 3.6 with comments indicating if the operation is performed on host, GPU or if it is a data transfer.

3.3.2 Persistent thread approach

An alternate execution strategy would be to force the threads to stay in loop and consume all newly produced tasks within the same kernel run. This is achieved by a GPU programming style called *Persistent Threads (PT)*. *Persistent Threads* approach assumes the threads to be live until no new work is produced by any thread. The number of parallel threads spawned in this case equals maximum number of concurrently active threads allowed by the GPU architecture. This can be achieved by adding the following:

- A task queue to store unclassified parameter regions
- Work fetch and dispatch logic
- A global termination condition

```
// CPU CODE
main ():
    read_experiment_data( dataPoints )          //cpu
    read_initial_search_space ( task_list )     //cpu
    init_gpu_mem ( dataPoints , task_list  )    //cpu -> gpu
    run_GPU_kernel ( task_list, new_list )      //gpu
    transfer_final_result                       // gpu -> cpu
}

// GPU CODE
kernel():
while ( EXIT condition not reached ) // Queue not empty OR
                                     // global_token_count != total_threads

{
    if( threadId == Queue Management Thread )
        fetch_task_read_index (task_index)  // get queue array index
                                            // for all warp threads to read
    }
    read_task_data (task_index)     // use task_index to read data
    perform_SIVIA (task,new_task)   // may create new_task
    update_thread_status (WRITE)    // status[tid] = WRITE if needs to
                                    // write to the queue
    if( threadId == Queue Management Thread )
        fetch_task_write_index (task_index)  // get queue array index
                                             // for all warp threads to write
    }
    update_thread_status (READ)         // status[tid] = READ if needs new task
}
end kernel.
```

Figure 3.7: Pseudocode for Persistent threads based GPU implementation

Global Task queue. The global task queue is configured as multiple array of C structs in the GPU DRAM such that each array stores interval set for one of the unknown parameters and data in all struct arrays at a particular index make up an unclassified parameter solution space. Note that splitting the data for unknown parameters into multiple arrays supports coalesced read/write. New tasks are added to the tail of the queue and consumed from the queue head. Threads compete for a global lock via atomic instructions to gain access to queue.

Fetch and Dispatch logic. A simple fetch logic would require threads to independently grab the lock and read the next available task in the queue. However, threads in a warp necessarily run in lockstep and thus would always reach the task fetch region at the same time. Also, we observe that the lock synchronization is only needed to update head or tail of the global queue as current readable or writable location is obtained. The actual task-data may then be read/written asynchronously.

This motivates our three step Fetch/Dispatch logic wherein only one thread in each warp needs to access the work queue synchronously. *First*, all threads in a warp update a shared status array to request READ/WRITE to the global queue depending on the state of execution. *Next*,a single queue management thread counts the number of READ/WRITE requests and accordingly manipulates head/tail of the global queue. This reserves the required number of read or write location in the global task queue. The operation is performed with a lock. The queue management thread then computes the global queue indices for each of the requestor based on its view of the global queue and stores these indices in a shared array visible to all threads in the warp. *Finally*, requestor threads of the warp read the global read/write indices and issue read/write instructions without needing a lock. Overall, this approach reduces the lock contention by up to 32 times and allows lock-free parallel accesses to task queue.

Termination Condition. Correct termination is guaranteed by maintaining a *Global Token* system. Each thread holds a token while performing a task and releases the token if its idle. A *Global Token Count* thus represents number of busy threads. A value of 0 implies no idle threads and a value of *total_threads* represents zero busy threads. Guaranteed termination is hence detected if task queue is empty and *Global Token Count* equals *total_threads*.

Figure 3.7 and Figure 3.8 outlines our Persistent Thread type GPU implementation.

3.3.3 Relevant prior work

Most of the prior work on irregular algorithms uses variants of Multi-Kernel approach. Some examples include graph algorithms [9], morph algorithms [25] and N-body simulation problems [4]. Task parallelism based approaches that use Persistent threads and a managed work queues have been explored in recent literature by [1], [8], [39] and [43]. Nasre [24] compares Persistent thread based approach with variants of Multi-kernel approach for graph algorithms.

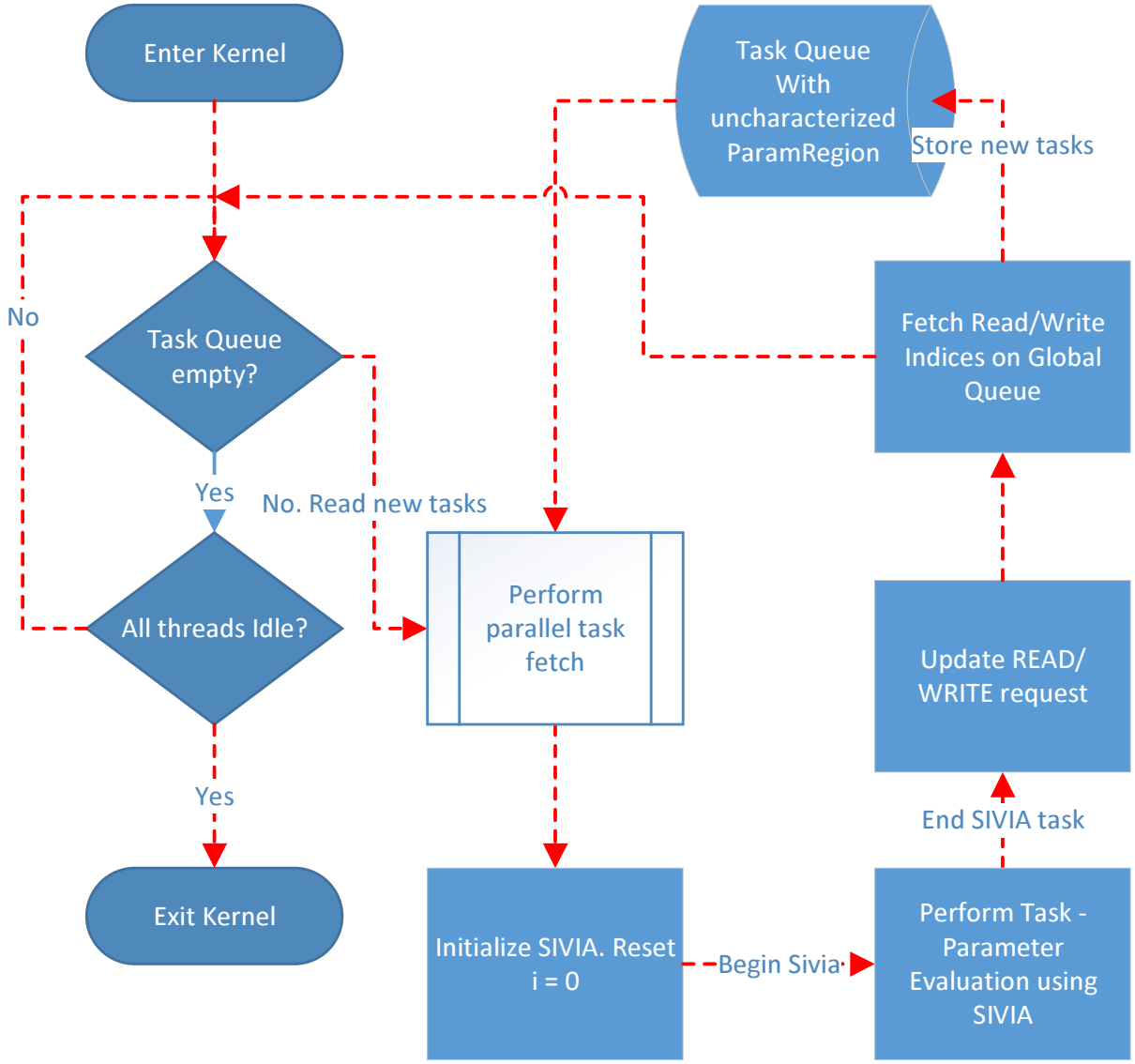

Figure 3.8: Parameter region evaluation using SIVIA

They show that persistent thread approach provides a more generic form of load balancing while Multi-kernel approach is more responsive to algorithm specific optimizations and can be more effective if directed optimizations are possible.

Work queue based scheduling techniques for irregular workloads on GPUs have been discussed earlier with reference to smooth surface subdivision task in Reyes rendering pipeline [19]. Patney [30] used a centralized queue for work division. Tzeng [40] proposed a work stealing approach with local queues to reduce contention for access to the queues. Lauterback [15] discussed work redistribution technique to further reduce contention for fine grained parallel tasks such as hierarchy construction and quicksort algorithm. These implementations have been aimed at applications with comparable main-task time to work fetch time in case of [15] or relatively moderate imbalance in workload generation as in [40]. Workload generation in parameter estimation often gets highly imbalanced with many threads only generating new tasks where as others ending up with many infeasible classifications. This may require many iterations of work stealing in case of a local task queue implementation. Thus we advocate a simpler queue management technique based on global queues with improvements specific to our algorithm behavior.

3.4 Optimizations and design improvements

High order taylor expansion of model equations coupled with interval arithmetic results in larger register footprint for complex models. We propose a set of optimizations to partially mitigate the effect of high register pressure. The optimizations are applicable to both execution schemes described in previous section. Two improvements to Persistent thread approach, namely Eager Fetch and Eager Create, are also suggested. These address inefficiencies due to warp divergence.

3.4.1 Register pressure

Biological models rely on double precision computations to accurately estimate parameter regions within desired error bounds. Nvidia GPUs have progressed rapidly in supporting double precision compute and have increased dedicated DP compute units with every new architecture release [42], [28]. However, double precision compute adds extra pressure on already scant resources such as shared memory, L1 cache and thread registers. Moreover, every arithmetic operation is performed on interval sets which further increases the register pressure. Adopting programming techniques to judicially use these resources becomes even more important.

Loop unrolling. We found loop unrolling to be a very effective technique to gain performance in compute heavy sections of our algorithm such as matrix multiplications and Jacobean calculation. Loop unrolling increases the number of available independent operation which in-

creases instruction level parallelism(ILP). It allows the compiler to use these independent operations to hide pipelining and memory access latencies. However, aggressive scheduling for ILP led to increased register pressure and thus register spilling. On GPUs registers spill to local memory which is placed in the main GPU DRAM and excessive spilling can thus shift the performance bottleneck back to memory bandwidth. To address this problem we designed our implementation to be more modular with separate device functions for each subsection of code. This approach logically provides a smaller scheduling window to the compiler allowing the compiler to make much better decisions on register allocation. This design decision allowed us to effectively reduce the overall register pressure by over a factor of 3.

Moreover, local memory loads and stores use caches to reduce latencies hence we configure the L1 cache as 48Kb instead of default 16Kb and consequently limit available shared memory to 16Kb. To reduce cache pollution we utilize constant memory for read-only data such as experimental state measurements. Shared memory is used in our implementation to maintain thread execution state flags and to share load/store indices of global queue computed by the queue management thread. With a reduced 16Kb shared memory configuration shared memory required by a thread block can limit the number of simultaneously active blocks on the SM. We notice that two bit state flag is enough for a thread and hence design 64 bit-wide single state variable for all the threads in a warp. The large state variable is updated and aggregated using ballot and vote functions provided in CUDA to avoid possible overheads from sequential shared data read.

3.4.2 Eager Task Fetch

An approach to reduce the impact of warp divergence due to unequal task run-length is presented. The key idea of *Eager Task Fetch* scheme is to allow threads in the warp to pause their evaluation and move out to job fetch stage when a predefined number of threads have become idle (called the max_idle). Multi-kernel approach has a simple and rigid task scheduler. The optimization is thus applied only to Persistent thread approach by adding two additional capabilities to its flexible task execution flow -

- Each thread maintains an execution state flag in the shared memory
- Pause-Resume capability - An additional exit condition is added to the evalution loop to allow busy threads to pause execution and converge with the waiting threads.

Upon completion of its evaluation process, a thread marks its execution state flag as 'idle'. Threads in the evaluation loop aggregate the total idle flags and accordingly decide whether to pause or continue. When enough number of threads become idle, the busy threads pause and change their execution state flag from 'busy' to 'paused' and the warp moves to the job fetch

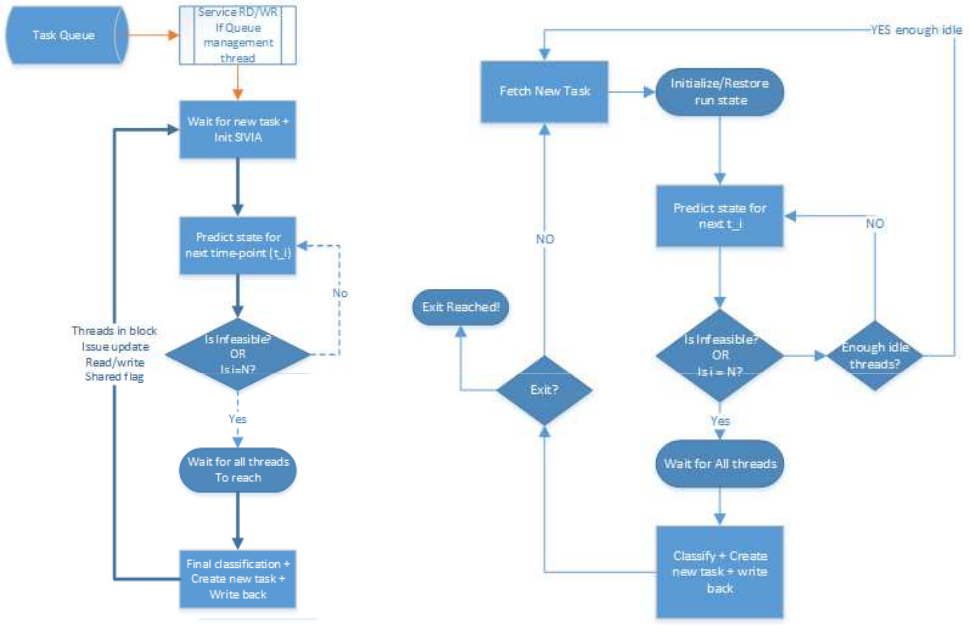

Figure 3.9: Optimized persistent thread approach

stage. The 'idle' threads that are able to obtain new parameter regions for evaluation mark their state flag as 'busy'. The warp then re-enters the evaluation loop. Threads with 'busy' state initiate a new evaluation process and threads with 'paused' state resume the previous evaluation loop. Figure 3.9 presents a comparative flowchart of a basic Persistent thread scheme and Persistent thread with Eager Task Fetch optimization.

Maximum allowed number of idle thread directly effect the frequency of pause-resume and is tuned based on the last seen state of available work on the global work list. Overhead of threads waiting for other threads to complete execution is higher than job fetch overhead hence max_idle value is set to 1 when number of pending tasks on global work list is high. Max_idle is gradually increased to maximum of 32 if available tasks on the global list is less than 64. This greatly reduces futile visits to work queue when there is not enough work present.

3.4.3 Cold startup and Eager Task Create scheme

The SIVIA algorithm typically starts with a single large parameter search space and gradually creates new tasks by bisecting indeterminate regions. The algorithm goes through a cold-startup period in which many threads remain idle as they wait for busy threads to create and post new tasks onto the tasklist. A larger fraction of threads must be kept busy in order to accelerate the progression through the slow startup phase. To achieve this the task scheduler can preemptively

divide larger regions and use them as tasks for the idle threads. Theoretically this may lead to creation of additional unwanted parameter sub-regions. The additional compute cost to evaluate these extra regions, however, is compensated by faster relevant work creation.

The idea is implemented in Persistent thread approach by adding an *Eager Task Creation* logic. This enables the task scheduler to aggressively divide newly fetched tasks whenever enough tasks are not available in the global work list. Thread execution state flags explained in previous section are used to determine the threads that have remained 'idle' after the task fetch phase. *Eager Task Creation* phase then forces threads with 'busy' state to bisect their newly fetched parameter region and donate the bisected sub-region to an 'idle' threads.

Chapter summary. The chapter presented a discussion of the inherent irregularities present in the SIVIA workload. The challenges that are anticipated while mapping such a workload onto a GPU architecture were also pointed out. The algorithm was shown to exhibit an irregular task creation and an irregular termination pattern. Two GPU approaches to map the workload, namely Multi-kernel approach and Persistent thread approach, were described. The approaches typically differ in the way they address the irregularities arising from runtime task creation property of SIVIA search algorithm. Key optimizations were presented at the end of the chapter. Next chapter provides a thorough evaluation of the two GPU approaches presented in Section 3.3.

Chapter 4

Evaluation of Single GPU approaches

We present a comparison of the two GPU approaches developed in previous chapter and quantify the benefits of various optimization techniques. We then apply GPU based SIVIA to solve unknown parameters for two real-world biological systems - Lotka-volterra model and SEIR model. The speedup obtained over a multi-core CPU implementation is reported.

4.1 Hardware details

The platform used for CPU-only simulations has two quad-core 3.2 GHz Intel Xeon processors. For single GPU simulations we use a system with one quad-core 2.27 GHz Intel Xeon processor hosting a Fermi C2075 GPU. The system is running RHEL 5.5 with Nvidia driver 304.54 and CUDA toolkit 4.1. The code was written using Boost 1.46 C++ interval library, OpenMPI 1.4 and compiled with gcc 4.1.2.

4.2 Comparing Multi-Kernel and Persistent thread approach

The two GPU approaches developed in previous chapter employ widely different execution paradigms. A key advantage of MultiKernel approach is its light weight task scheduler that fetches a predefined number of tasks from the task list. The load distribution among GPU cores is static and the number of tasks performed by each thread is known before kernel launch. The approach is generally easier to implement and doesnot require explicit worklist management. Persistent thread approach, on the other hand, facilitates dynamic load scheduling. As a result, thread groups can end up performing widely different number of evaluation tasks. Persistent thread approach expresses the problem as a task parallel system and thus provides more room

Table 4.1: Microbenchmark Performance summary

GPU performance chart			
Load imabalance	Load Config	Approach	Run-time (Sec)
Low	1000 iter - 85% 500 iter - 5% 250 iter - 5% 125 iter - 5%	Persist-NoOptim Persist-EagerFetch MultiCall	28 27 25
Medium	1000 iter - 25% 500 iter - 25% 250 iter - 25% 125 iter - 25%	Persist-NoOptim Persist-EagerFetch MultiCall	20 13.3 18
High	1000 iter - 50% 500 iter - 0% 250 iter - 0% 125 iter - 50%	Persist-NoOptim Persist-EagerFetch MultiCall	22 9.8 20

to develop fine-grained task scheduling schemes such as Eager Fetch.

4.2.1 Comparison methodology

To illustrate the benefits of fine-grained eager-fetch task scheduling optimization we develop a micro-benchmark that artificially introduces warp imbalance to the workload. This is achieved by generating a workload constituting of tasks with different run-lengths. One such workload may have 100000 tasks with 25% of tasks needing 1000 prediction iterations, 20% needing 500 iterations, 30% and 25% of the remaining tasks needing 250 and 125 iterations respectively. Tasks with different run-lengths are randomly mixed together to represent a realistic workload.

4.2.2 Microbenchmark results

Table 4.1 shows runtime for workloads with different degrees of imbalance on Persistent and Multi-kernel GPU implementation. The table greatly highlights the benefits of using Persistent thread approach in conjunction with fine grained task scheduling schemes such as Eager Fetch. Real parameter search problems are expected to show high imbalance in the initial stage and medium to low imbalance in the later stage of the execution. Hence optimized Persistent thread approach is our natural choice as the primary single GPU approach.

Table 4.2: Causes of thread divergence

Warp Imbalance and Thread Divergence		
Reason	Exec phase	Solution
Insufficient tasks in Q	Start and End phase	Eager-Create
Task finishing early-Infeasible	Any-time	Eager-Fetch

4.3 Effects of optimization on real models

Table 4.2 summarizes the reasons for warp imbalance and the techniques we implement to reduce it.

To demonstrate the effects of our techniques Fig. 4.1 shows the histogram of threads that are active during the state-prediction step. The X-axis depicts the number of active threads in the warp and Y-axis is the number of computer cycles with x number of active threads. The figure shows that without warp level optimizations the GPU is at full occupancy for only 55%. It runs at 80-95% occupancy for 30% of time primarily due to randomly occurring imbalances from infeasible regions and it runs at 5-30% occupancy for 10% of the time. The Eager-Create primarily reduces imbalance at the left section of the graph where threads are idle due to insufficient tasks (param-regions) at the start or end of the estimation task. Eager-Fetch primarily addresses the random imbalance producing more than 99% of compute cycles with full-warp occupancy. The net performance speed-up from these optimizations are summarized in Fig. 4.2

4.4 GPU performance discussion

Table 4.3 compares run-time for parameter estimation on Lotka-Volterra and SEIR model. Two different configurations of SEIR model were timed with 2 and 3 unknown parameters respectively. The multi-core CPU implementation is based on Pthreads and uses a global work queue similar to our GPU execution model. Threads up to the number of available CPU cores are spawned progressively as work queue gets filled. Once maximum number of allowed thread are up, they iteratively fetch work or add work onto the work list. The GPU kernel implements the global work queue based persistent thread approach. The kernels are launched with 32 threads/block to allow simplified block-wide synchronization. Small block size is compensated by spawning enough kernel blocks to have 8 Blocks/SM and thus achieving the optimal active warps

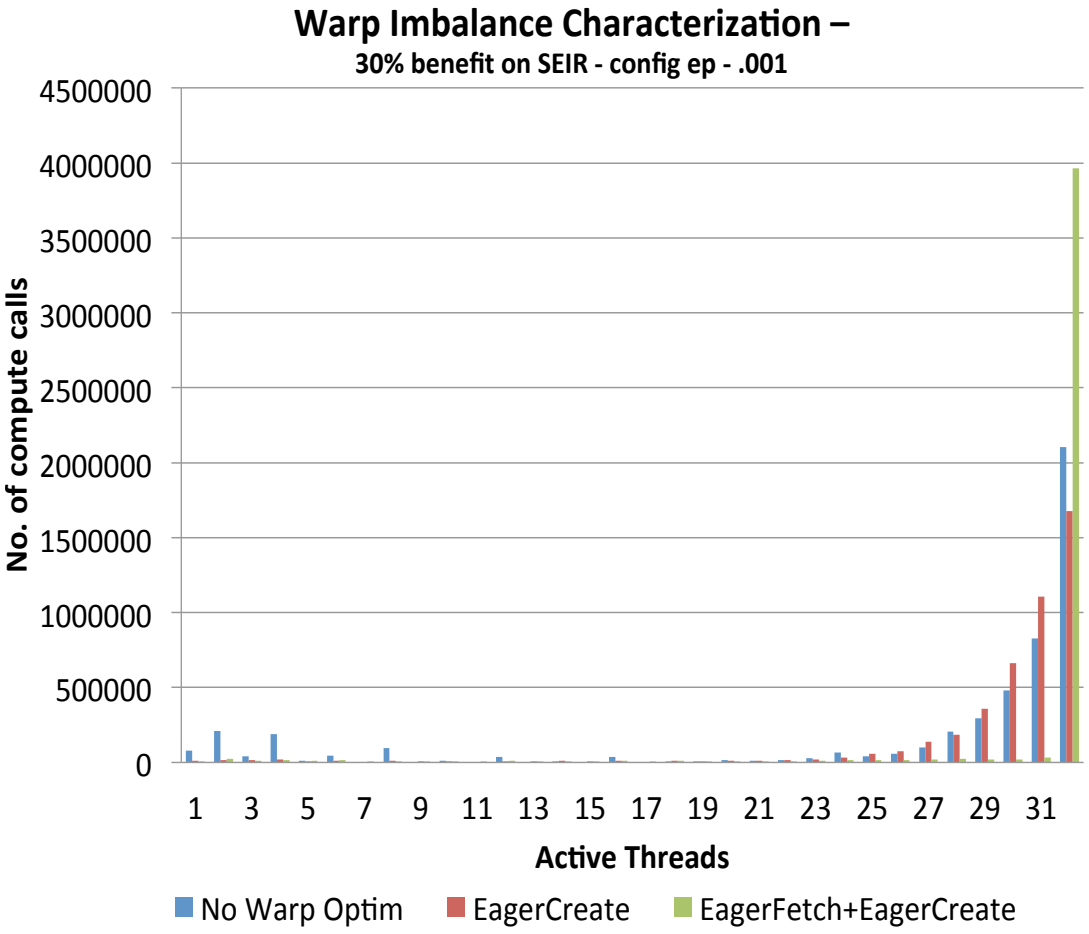

Figure 4.1: Improved warp occupancy

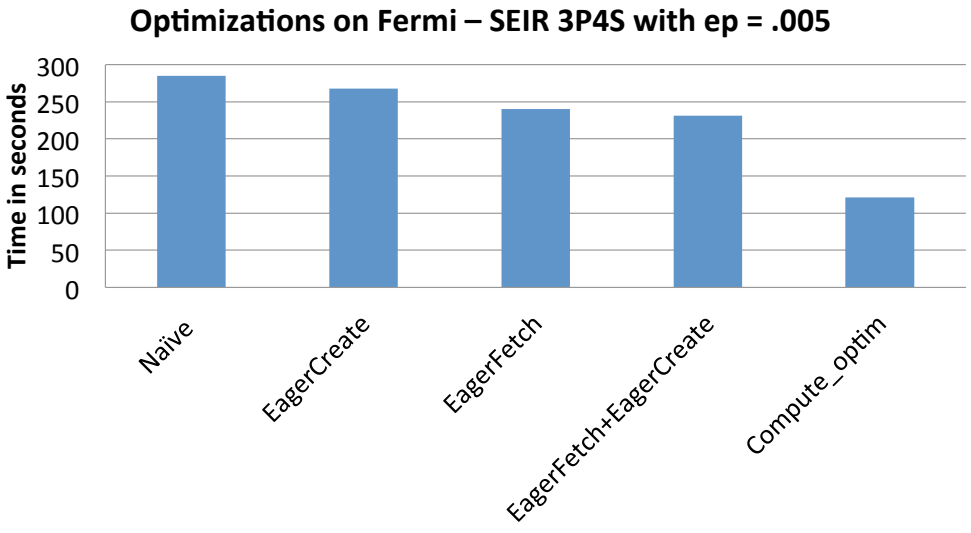

Figure 4.2: Performance gain from Optimizations

Table 4.3: Performance summary

GPU performance chart			
Model	Config	Avg.Time (s)	Speed-up
Lotka-Volterra	CPU-1	31.4	1
	CPU-8	4.14	7.59
	GPU-Fer	1.52	20.65
	GPU-K20	.85	37
SEIR-2P4S	CPU-1	4515	1
	CPU-8	574	7.86
	GPU-Fer	50.3	89.76
	GPU-K20	12.61	358.63
SEIR-3P4S, ϵ=.0005	CPU-1	16523	1
	CPU-8	2134	7.76
	GPU-Fer	120.8	137.06
	GPU-K20	26.5	624.63

number of 8 warp/SM which is generally enough to effectively hide memory latency [12]. Thus Grid-size can be computed as $8 \times number\_of\_SM$. Single Fermi GPU implementation achieves an average speedup of roughly 20x, 90x and 137x over a single core CPU implementation for Lotka-Voletrra and SEIR models respectively. The speedup improves with growing model complexity. This can be attributed to a more complex state-estimation (guaranteed integration) step leading to increase in compute intensity of each task. Also, complex models result in more divisions of parameter region resulting in higher available parallelism allowing GPU to stay fully occupied for longer fraction of total run-time. Note that for SEIR model true feasible regions are obtained only with $\epsilon = .0001$. However, owing to the large execution time required for simulation of a full SEIR model with $\epsilon = .0001$ on CPU only systems, the table reports execution time of SEIR model with ep=0.0005. The final result of estimation run with $\epsilon = .0005$ gives a coarser approximation of parameter region set consisting of only indeterminate regions. The projected execution time for full SEIR model on an 8-core CPU is about 11 days compared to actual runtime of 13 hours clocked on the Fermi GPU and 2.8 hours on Kepler.

Running estimation on Kepler results in further 2X and 4.5X speed-up over Fermi for Lotka-Volterra and SEIR model respectively. An important architectural change that helps Kepler over Fermi GPUs for parameter estimation problem is the maximum allowed Registers Per Thread. Kepler allows a maximum of 255 Registers compared to only 63 Reg/thread on Fermi. Owing to the complexity of state-prediction step number of available registers quickly become a bottleneck resulting in large register spills onto costly Global memory. The cache hierarchy

on GPUs and coalesced memory accesses partially hide the large DRAM fetches due to register spills. However, we still observed heavy DRAM access for SEIR model on Fermi particularly due to 63reg/thread limit resulting in heavy spilling. Kepler threads with 255 available registers greatly avoid the expensive spilling and hence are less memory-bottlenecked and hence providing a much bigger speedup for complex SEIR model.

4.5 Additional discussion

A key difference between parameter estimation workload and the standard irregular workloads such as graph and unbalanced tree algorithms is the amount of compute required per task element. A region evaluation involves varying number of compute heavy state prediction steps. Low GPU occupancy due to divergent warps is hence the key performance barrier which we effectively remove by implementing a fine grained task scheduling policy - *Eager Fetch*. We demonstrate the impact of our fine grained task scheduling policy in section 4.2. We argue that*Eager Fetch* policy can benefit many irregular applications provided they are expressed as task parallel systems and have varying number of compute iterations per task. Monte Carlo simulations and Fractal calculation problem are two such applications that we intend to evaluate as part of future work.

Chapter 5

Parallelization on Multiple GPUs in a Heterogeneous Cluster

The primary challenge in designing a load balanced multiGPU implementation is the spatial irregularity of task creation. A naive static work distribution technique that simply pre-divides and distributes the n-D parameter space among GPU nodes is bound to suffer from high load imbalance. GPUs that receive a share of actual solution region based on static distribution tend to go through numerous subdivisions. Each new region is then subsequently evaluated. On the other hand, GPUs that are designated to search in regions that contain no feasible solution may not need to subdivide and evaluate as many regions since they are more likely to evaluate large chunks of search space as infeasible. Thus GPUs can end up having highly uneven runtimes depending on the number of subdivisions their designated search sub-space goes through.

The lack of prior knowledge about the density of compute requirement in the search space demands a dynamic load sharing approach. In the following section we present two possible approaches for dynamic load balancing and evaluate them for performance and scalability towards the end of this chapter.

5.1 Load balanced multiGPU design

In the previous chapters, persistent thread based execution model was established as the optimal approach for a single GPU implementation. In this approach the CPU simply offloads the problem onto the GPU, the persistent kernel exhaustively evaluates the search space and returns back to the CPU with the final results. However, in order for the CPU to dynamically distribute ongoing work among GPUs we need an execution model in which CPU can access a part of workload being generated on the GPU. This property motivates two possible approaches:

Breaking the kernel into multiple stages One approach is to somehow have multiple kernel calls within the persistent thread execution model. This would allow the host to have access to the task list at the end of each kernel run. Some task distribution technique can then be used to fairly divide this load. This forms the basis of our *Semi-dynamic staged execution scheme*.

Work exchange within live kernel If the host is able to add or extract tasks from GPU task queue while the GPU kernel is in progress then it can allow naturally free-flowing work sharing among GPUs. Challenge here is to achieve this in a non-intrusive way so as to add minimum overhead to the kernel compute path. The *ActiveGPU load balancing scheme* attempts to achieve this goal.

The single GPU design focused on load balancing within the GPU and the above two approaches effectively form a communication layer between GPU and CPU. However, any multi-GPU approach would also require an efficient communication layer between different hosts. In the following sub-sections we describe our host-to-host communication layer and then discuss the load balancing schemes *Semi-dynamic staged execution* and *Fully dynamic ActiveGPU load balancer* in more detail.

5.1.1 Distributing load among host nodes

Message passing interface(MPI) library [7] is used to design host side load sharing communication layer. MPI provides two classes of send/receive functions - Blocking and Non-blocking. Code cannot proceed beyond a blocking call while non-blocking calls post the requested send/receive command onto a buffer and proceed forward. Non-blocking calls need to be checked for completion which clears the pending state of the call. The call that checks completion can also be blocking or non-blocking.

We implement a demand based master-slave work distribution layer using MPI library which allows CPU thread on slave node to either request work from the master node or release excess work to master node. It is not possible to predict when slave node would request or release more work and when master node would have enough work to service demand request from slave node. Hence in order to account for this non-deterministic behavior we choose non-blocking MPI calls that allow flexibile communication between master and slave nodes.

Pending_Task_Queue All nodes maintain a $Pending\_Task\_Queue$ which is used to aggregate pending tasks coming from the GPU and other nodes. Tasks are exchanged between nodes at $Task\_Chunk$ granularity and chunk size can be tuned using $Group\_size$. A smaller chunk size allows finer grained task sharing.

Master and slave node operation. Slave node sends a non-blocking request for more tasks when $Pending\_Task\_Queue$ has less than $requestThreshold$ number of task chunks. Slave

```
master node:
    while (GLOBAL_EXIT == FALSE)
        gpu_cpu_task_exchange();    //cpu -> gpu
        for loop (all slave nodes)
            msg = check_inbox();
            if(msg == TASK_REQUEST)
                push_req_to_requestQ();
            if(msg == TASK_DONATE)
                push_task_to_pending_Q(); // other cpu -> cpu
            if(msg == SLAVE_IDLE)
                add_to_idleNodeList();
        end for loop.

        for loop (all pending request)
            if(requestQ != EMPTY)
                send_task_to_slave(); // cpu -> other cpu
                remove_from_idleNodeList();
        end for loop.

        if (no_of_IDLE == total_slave && GPU == IDLE)
            send_exit_msg();
            GLOBAL_EXIT = TRUE
    end while.
```

```
slave node:
    while (GLOBAL_EXIT == FALSE)
        gpu_cpu_task_exchange();    //cpu -> gpu
        if(pending_Q.size() < requestThreshold)
            request_task();
            task_Requested = TRUE;
        if(pending_Q.size() >releaseThreshold)
            release_task(); // cpu -> master cpu


        msg = listen_to_master();
        if(msg == TASK_CHUNK)
            push_task_to_pending_Q(); // master node -> cpu
            task_requested = FALSE;
        end if.
        if(msg == GLOBAL_EXIT)
            GLOBAL_EXIT = TRUE
    end while.
```

Figure 5.1: Pseudocode for non-blocking master-slave communication

node may also release task chunks to master node if *Pending_Task_Queue* has more than *releaseThreshold* number of task chunks. Nonblocking sends are queued in a buffer until read by the receiving node. Master node checks the MPI receive buffer corresponding to each slave node in a round-robin fashion. All incoming work requests from slave nodes are placed in a *Request queue* and incoming excess task chunks are stored in its *Pending_Task_Queue*. Master node then picks task packets from *Pending_Task_Queue* and services pending task requests from *Request queue*.

Exit condition. All host threads execute in a while loop and exit only when a global exit condition has been met. The global exit condition checks if all GPUs on all nodes are idle and there is no more regions to be evaluated. Slave nodes notify the master node if their GPU is idle. Master node in turn determines if global exit condition has been met and broadcasts an exit message to all nodes.

Interested reader may refer to [5] and [33] that summarize master-slave and work-stealing based load balancing techniques. The pseudocode in Figure 5.1 gives an overview of the MPI based communication stack for master and slave nodes. For completion of analysis a synchronous host-to-host communication layer based on blocking calls was also implemented. MPI Gather function was used to aggregate all extra work at master node and MPI Scatter function was used to redistribute tasks in a one time static fashion.

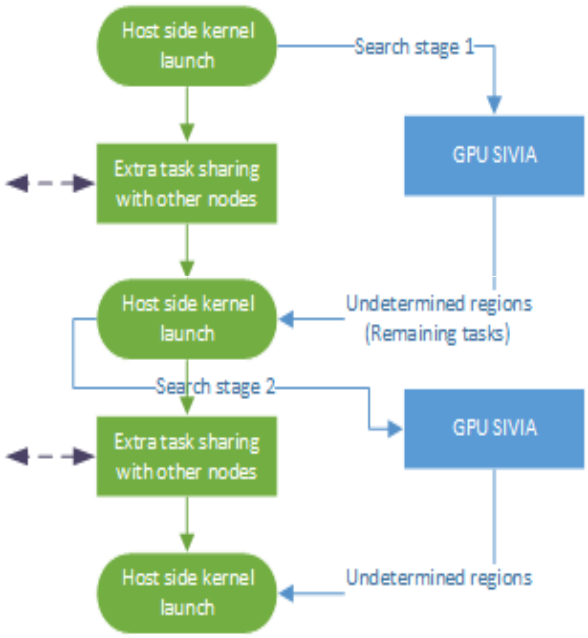

Figure 5.2: Flow diagram of Staged kernel execution

5.1.2 Staged Execution

The idea behind staged execution scheme is to break the GPU kernel execution by creating artificial execution stages and balancing the load dynamically between these stages. In SIVIA, the search granularity can be controlled using the ϵ parameter that defines the minimum edge length of a sub-region. Setting a high ϵ value would force the search to stop at a coarser granularity. A parameter search problem can thus be broken into hierarchical stages using multiple ϵ values obtained from a decreasing step function. For example, parameter search problem for a model with minimum $\epsilon = 0.001$ can be broken down into two stages using a single intermediate $\epsilon = 0.005$. The first stage runs with $\epsilon = 0.005$ and the resulting indeterminate regions are used as input search regions for the second stage which is run at $\epsilon = 0.001$.

After breaking the problem into stages, CPU thread can collect the intermediate regions from the GPU at the end of each execution stage. These intermediate regions are effectively regions that still need to be searched at finer granularity. The CPU thread adds them to a $Pending\_Task\_Queue$. CPU Nodes then perform load distribution using the previously described host-to-host task sharing protocol and use the work-balanced task list as input to the next stage of GPU kernel. We define two execution variants based on blocking and non-blocking host-to-host task sharing protocols. *Static Staged execution* employs scatter gather based blocking host-to-host task sharing while *Semi-dynamic Staged execution* uses non-blocking host-to-host task sharing protocol.

By allowing GPU kernels to exit at multiple stages, this technique allows frequent host side work aggregation and succeeds in cutting down the extent of imbalance. Fig. 5.2 illustrates this execution paradigm. Multiple kernel calls with different ϵ are shown. Moving top to bottom represents time progression.

5.1.3 ActiveGPU load balancer

In standard GPU execution schemes CPU only participates in data transfer and kernel launch. Instead, we design an execution model where host thread acts as a helper thread that continuously manages GPU workload by silently injecting and extracting tasks as the GPU kernel continues to consume tasks from the task queue.

The CUDA execution model provides two ways to communicate between host(CPU) and device(GPU) while a kernel is running [41]:

- Asynchronous memory transfer calls coupled with CUDA streams
- Device mapped host memory [29].

Asynchronous memory transfer allows CPU thread to progress while data is being transferred to the GPU via PCIe bus. A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. The operations within a stream are guaranteed to execute in order but operations from different streams can be interleaved. Launching a kernel from one CUDA stream and exchanging data from another CUDA stream enables us to read results from the GPU while kernel is still running. Asynchronous memory transfer calls, however, have heavy start-up overhead and only perform optimally when transferring large chunks of data. Device mapped host memory allocates page-locked memory on the host DRAM, which can directly be accessed by the GPU kernel as well as the host. However, allocating large amount of page-locked memory to exchange big data chunks between CPU and GPU can quickly degrade system performance since it reduces the amount of memory available to for memory paging resulting in larger number of page faults. Thus we only use device mapped host memory to facilitate small sized data exchange like signaling operations between host and device. The heavy data transfer to read out or push in task chunks in performed using asynchronous memory transfer calls.

Two consecutive memory accesses from either host or device may be deterministically ordered. However, due to lack of ordering and locking mechanism between CPU and GPU, read after write(RAW) or write after read (WAR) type race conditions are highly possible if both CPU and GPU access the same memory location simultaneously. To avoid simultaneous access of same memory location by host and device threads we introduced two new sets of fixed sized queues namely $Overflow$ and $Incoming$ queue residing in the device memory as an addendum

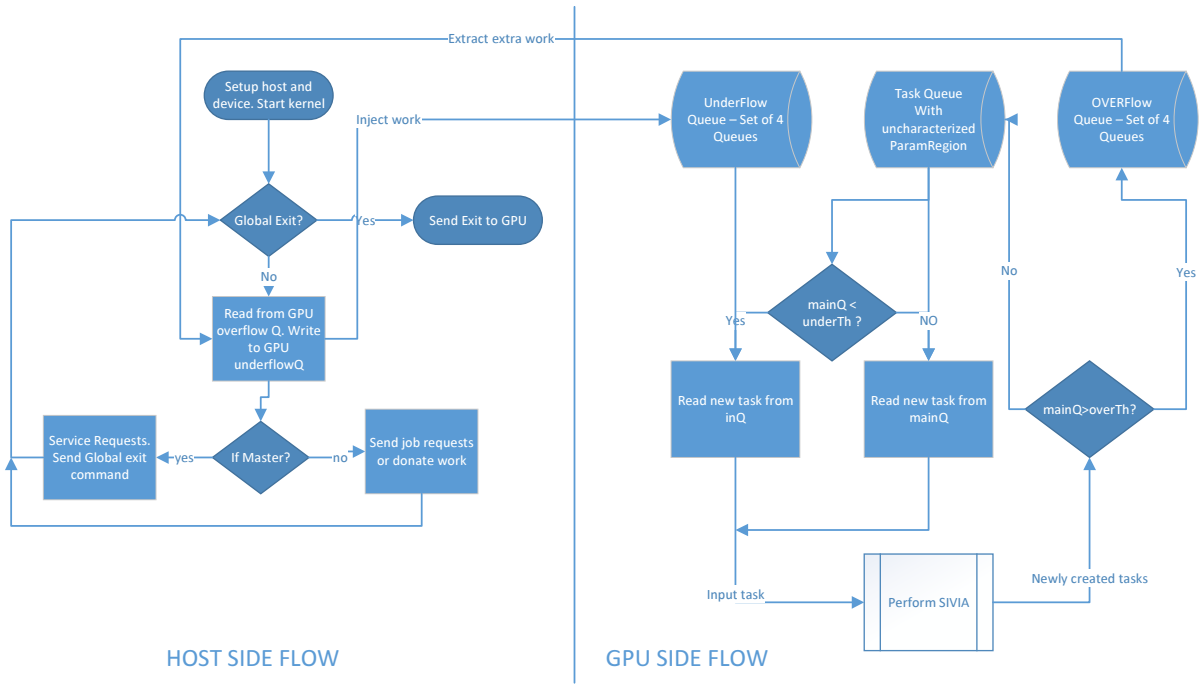

Figure 5.3: Flow diagram of ActiveGPU load balancing approach

to the main task queue. We add a $Queue\_Management\_Logic$(QML) between the GPU threads and the queues. Two states ($Q\_in\_overflow$ and $Q\_in\_starvation$) representing the state of the main task queue are defined along with statically tuned thresholds $overflowTh$ and $starveTh$. These thresholds are used by QML to determine if the main queue is in overflow or starvation state. Task read/write requests, originating from GPU threads, are thus accordingly routed towards one of the three correct queues. If main queue is overflowing then QML forces GPU threads to fill in $Overflow$ queues when writing back new tasks where as if main queue is starving QML directs GPU threads to pick new tasks from incoming queues.

Additionally, a set of $Queue\_Status$ flags are introduced that maintain the current status of $Overflow$ and $Incoming$ queues. There are three valid states for the $Queue\_Status$ flag - $Empty$, $Full$ and $Busy$. When a host or device empties a queue it sets corresponding $Queue\_Status$ to $Empty$ and similarly the status flag is set to $Full$ once host/device has filled up a queue. GPU threads set the queue state to $Busy$ while they are filling up or reading from queue. Note, that CPU host always reads/writes to these addendum queues in queue size chunks and hence only writes onto a queue when its $Empty$ and reads from a queue when its $Full$. Defining the $Queue\_Status$ is significantly important to enable mutually exclusive access to a particular queue in absence of a CPU-GPU wide memory access ordering or locking mechanism. The host thread continuously empties tasks that have spilled GPU threads into the $Overflow$

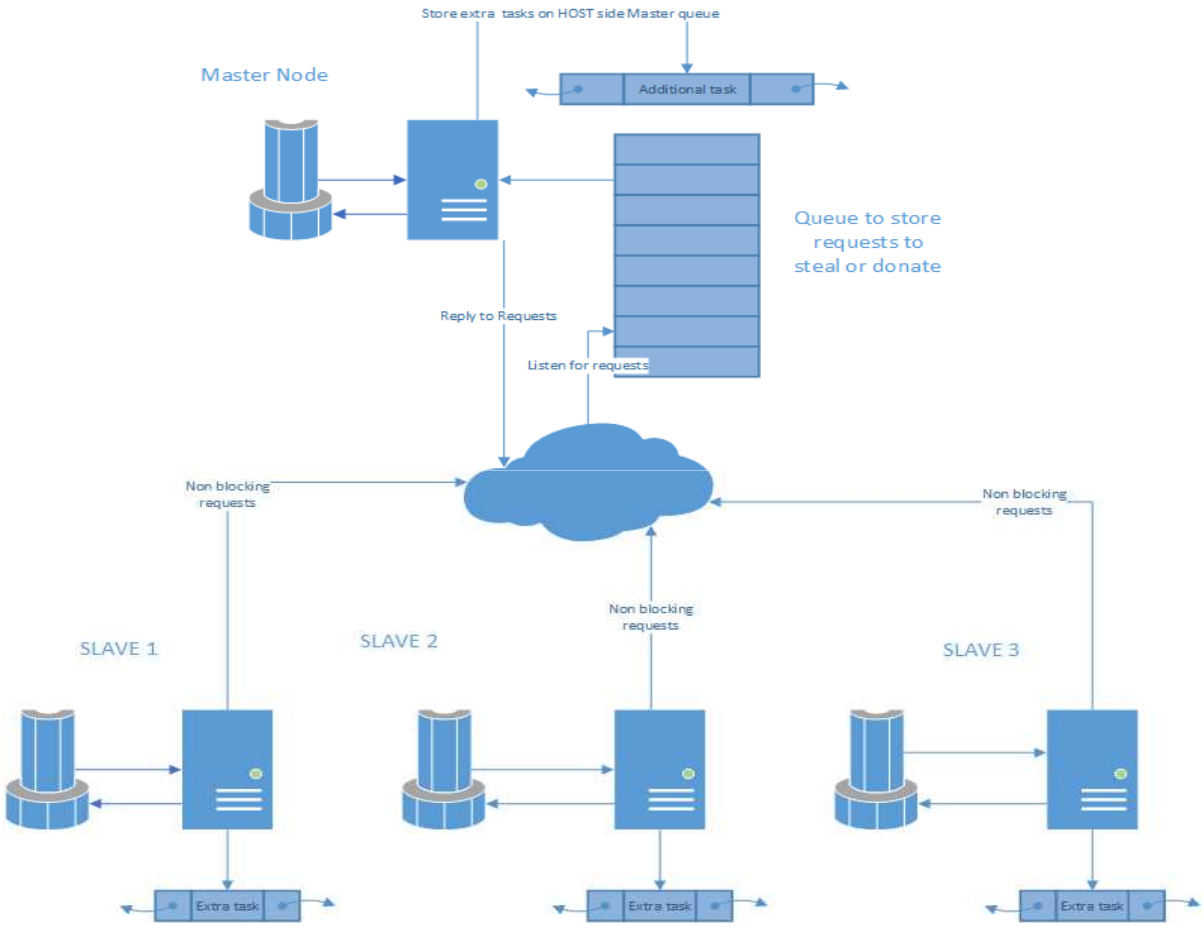

Figure 5.4: Dynamic load balancing across multiple GPUs

queue and stores them in the host side $Pending\_Task\_Queue$. These tasks can now take part in the load sharing effort. Moreover, if any $Incoming$ queues are empty host thread tries to fill it using the tasks in $Pending\_Task\_Queue$.

We also modify the exit logic of the kernel so that it notifies the CPU when GPU is idle and exits only when signaled by the CPU thread. Along with managing the device queues, host thread continuously engages in non-blocking communications with other hosts to exchange tasks as explained earlier. Fig. 5.3 illustrates the host and device side of activities under a dynamic load shared multiGPU implementation. Fig. 5.4 provides an overall view of multiGPU setup with host-device communication and inter-host communication via the network cloud. The diagram also shows the request list that the Master node maintains and the host-side task queue that is maintained by nodes.

5.2 Evaluation of multiGPU implementation

In this section we evaluate the two multi-gpu approaches we develop in previous sections - *Staged execution* and fully dynamic *ActiveGPU load balancer*. To illustrate the benefits of non-blocking host-to-host communication layer we present two variants of *Staged execution* - *Semi_Dynamic* distribution that uses non-blocking MPI calls for task sharing and *Static* distribution that uses blocking MPI calls for task distribution. Multi-GPU simulations are performed on a cluster of upto 16 C2050 Fermi GPU nodes and a maximum of 618 AMD Opteron cores. The GPUs on the cluster used CUDA toolkit version 5.0.

The execution time for a system with up to 16 distributed GPUs is reported in Fig. 5.5. For $StagedExecution$ we developed two stages. Note that the ϵ used as stage boundary, was carefully chosen with prior knowledge of the run-behavior. In this approach all GPUs were globally synchronized across stages with MPI_Barrier and tasks are distributed via a static One-time scatter operation. This approach is shown to be least scalable owing to the static load-balancing strategy which is a misfit for an irregular applications such as ours as noted before. The results are shown primarily to compare and demonstrate the benefits of the other two approaches. *Semi-Dynamic Staged Execution* also employs two stages but the large pending task-list extracted from GPU in an earlier stage is distributed using non-blocking host-to-host communication stack. This variant shows improved scalability. Finally the *ActiveGPU load balancer* is a fully autonomous self-balancing system and performs best in terms of scalability.

$OverflowTh$ and $StarveTh$ of *ActiveGPU load balancer* governs the frequency of task exhange between CPU and GPU. In the same way, $ReleaseThreshold$ and $RequestThreshold$ control the granularity at which tasks chunks are exchanged between CPU nodes. Using different set of thresholds and *chunk_size* we created different sharing setups. A fine grained sharing system would have smaller thresholds and smaller *chunk_size* for higher frequency of task ex-

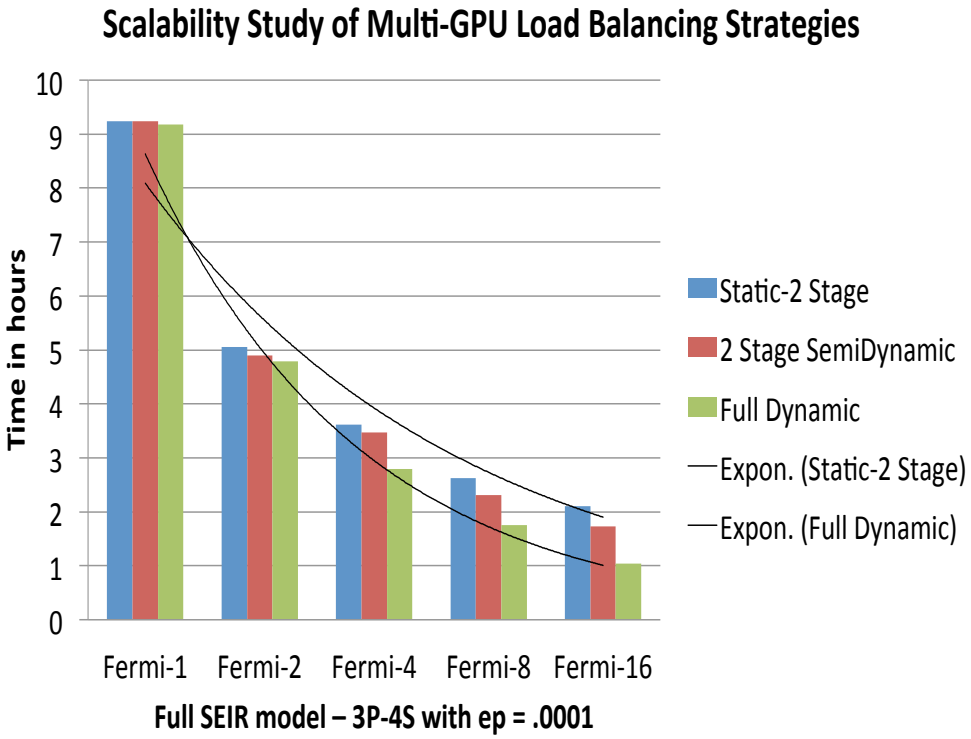

Figure 5.5: Performance comparison of multiGPU load balancing techniques

change throughout the system. For a small 16-GPU cluster fine-grained sharing produce a much more balanced load distribution as shown in Fig. 5.6. The resulting performance benefits are shown in 5.7. While finer grained sharing is the preffered setup the knobs may have to be tuned for a very large distributed system with interconnect bandwidth constraints to reduce the higher load-sharing traffic fine-grained task exchange.

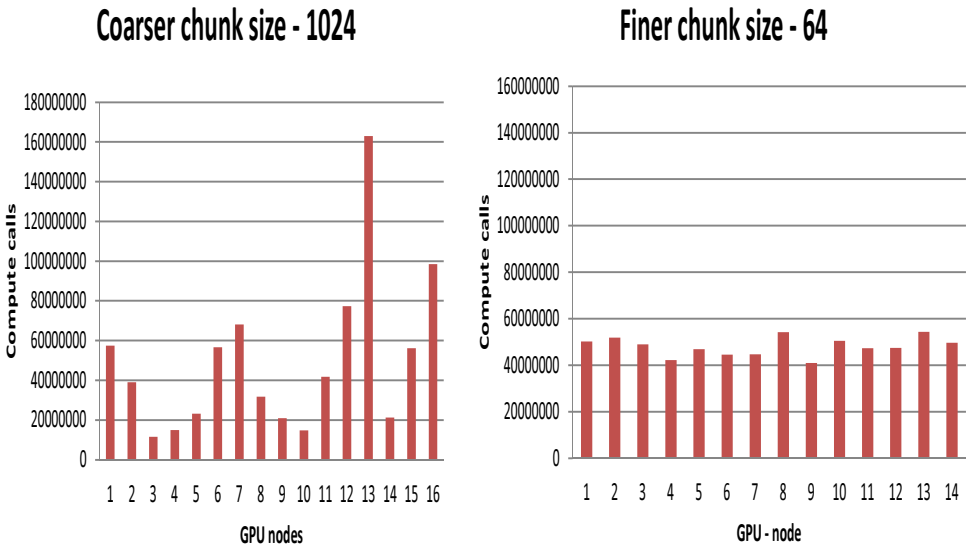

Figure 5.6: Load balancing with Coarse and Fine grained work sharing

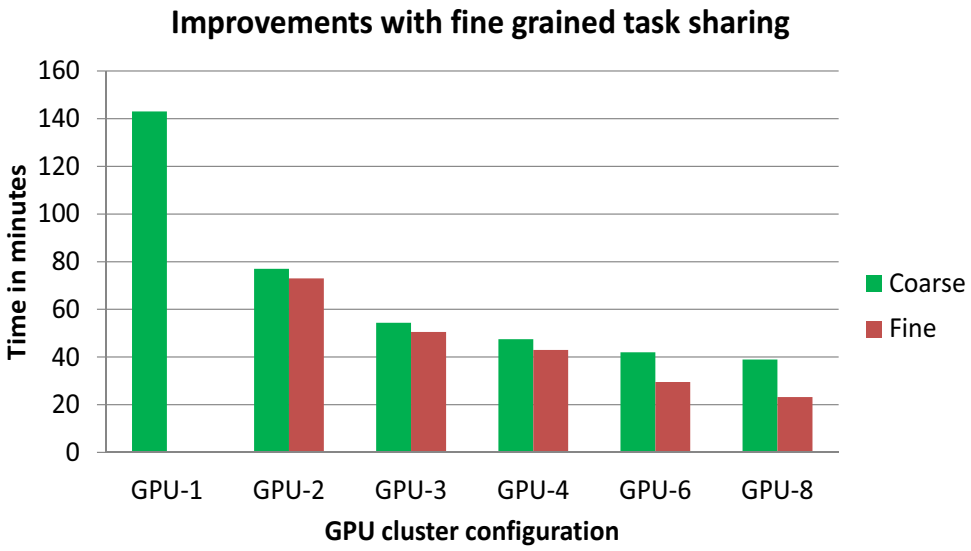

Figure 5.7: Performance of Coarse and Fine grained work sharing

Chapter 6

Additional Discussion

Guaranteed integration techniques form the core of Bounded error estimation. In this section we make key observations about the mathematical properties and deficiencies of the Guaranteed integration technique.

6.1 Wrapping effect and pessimistic bounds

The key property of Guaranteed integration methods is that the they produce guaranteed or validated enclosure of the predicted state. Unlike numerical methods which report solutions with an unknown global error, this approach is able to place a guaranteed bound around its prediction. In order to produce a guaranteed enclosure, initial error or uncertainty is propagated through out the multi-step prediction process and gets accentuated with each step. Hence, in cases of considerably large uncertainty in initial conditions the enclosures soon start to become very large and this property is termed as *wrapping effect*. Understandably, wrapping effect also starts to show up when predicting across large number of time-points due to repeated propagation and accentuation of error. Note, that this property of over-approximation and hence bloating of propagated error is a property of Interval analysis. For eg, addition of two intervals would always result in a larger interval.

Prior work based on mean value forms and matrix pre-conditioning has tried to address the wrapping issue. Work by Rhim [36] and later Raisi [34] addressed this issue by using high order Taylor expansions to produce tighter enclosures. By gaining a tighter enclosure they try to decrease the pessimism introduced by the propagating error since the interval sizes are kept small. Note that use of higher order of Taylor expansion increases the compute requirements and researchers are still attempting to find better performing techniques that are not as compute hungry. Some of the recent work [22] has been focused on finding newer ways to get tight enclosures specifically for models showing monotonic properties.

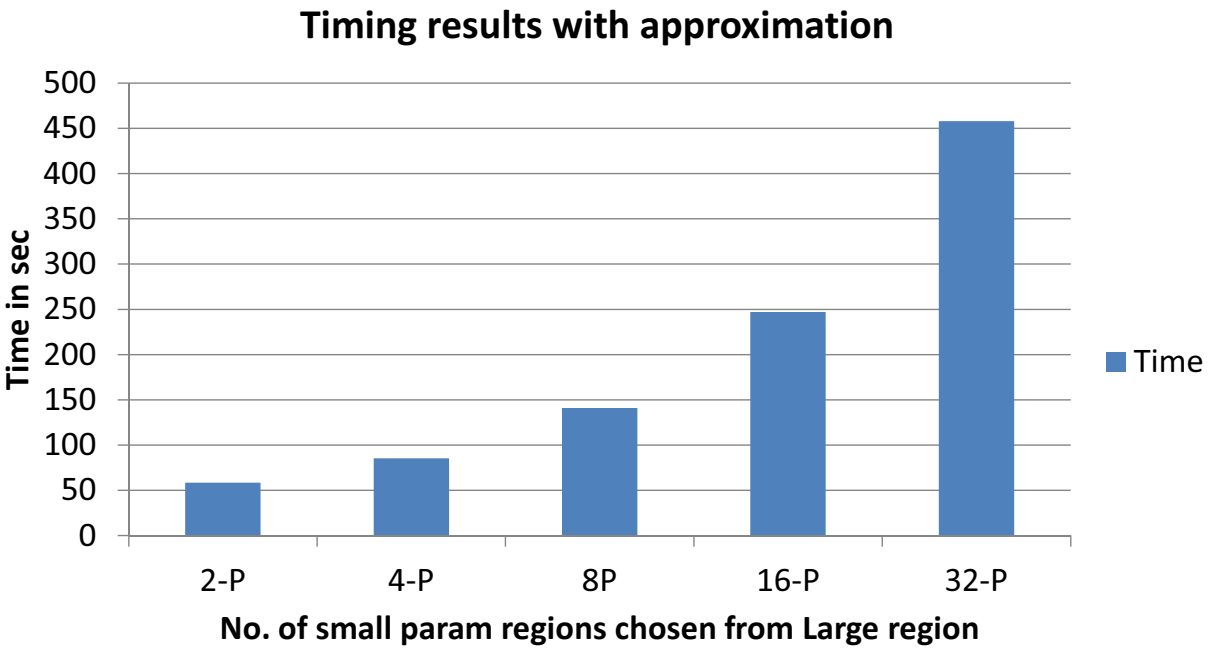

Figure 6.1: Performance gain with approximation

However, we argue that our approach focuses on the irregular nature of the SIVIA algorithm and is not completely dependent on the guaranteed integration or the enclosure methodology used. Future techniques should still result in similarly irregular workload and hence would benefit from the parallel GPU approaches presented in this work.

6.2 Algorithmic solutions

In this section we present some algorithmic approximation techniques that can greatly reduce compute time and give useful insights for long running estimation problems. Running different models on SIVIA, we observed that a feasible region P_f is often broken down into large number of smaller regions and ultimately most of them are classified as feasible. This is an artifact of pessimism introduced due to error propagation properties that behave worse on bigger regions. Hence, despite a region being feasible its bigger size forces the algorithm to categorize it as indeterminate resulting in further sub-division. Prior knowledge of this mathematical possibility allows us to develop an approximate method. During execution of estimation algorithm our implementation looks for cases where a region is repeatedly being bisected without rejecting any smaller sub-region. We then shift the execution process to pick smaller sub-regions from the large search-region based on a pre-defined selection logic with length of each side being ϵ. If all selected regions are found to be feasible then we approximate the whole box to be feasible or else

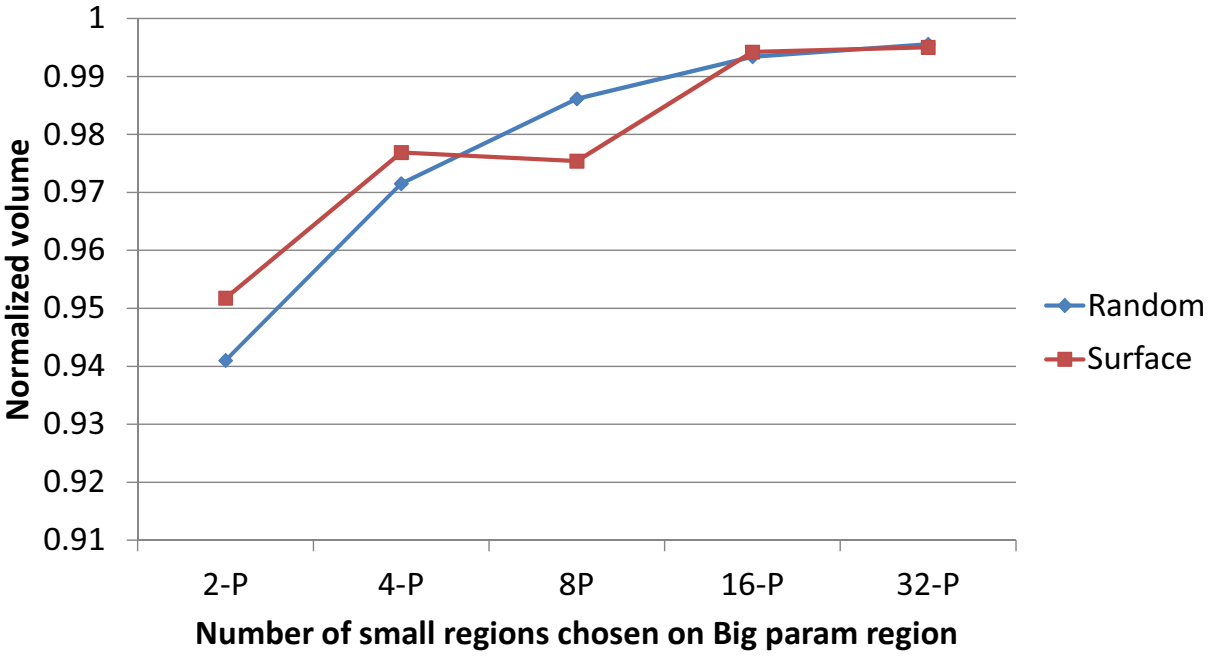

Figure 6.2: Accuracy of approximated param-set

revert back to normal searching for large search-region. In our experiment on SEIR model, the number of sub-regions selected was varied from 2 to 32. The larger search-regions typically had edge sizes of 0.0005 while the ϵ was set at 0.0001. For 2,4 and 8 sub-region cases (2-P,4-P and 8-P case) region selection logic was programmed to select sub-regions at diagonally opposite corners of the larger search-region. For the 16-P case, 8 more regions were picked from the center point of any 8 edges of the 3D region. The 32-P case consisted of 16 randomly selected regions in addition to the ones in 16-P case. The fig 6.1 shows the timing results for different number of param-sub regions chosen to test the hypothesis. In fig 6.2 we demonstrate with sub-regions having 1/5th of the original edge-size 16 sub-regions per larger-region is enough for SEIR model to detect feasible regions with over 99% accuracy compared to the original run. The accuracy was tested by comparing the volume of original feasible solution with the approximate one. Hence with a speed of additional 100x we are able to produce results with 99% accuracy for certain models. We acknowledge that these results may be heavily dependent on the model. Evaluation of this technique on more models is part of future work. Readers are however warned that the estimate of parameter set obtained through this approximation cannot always be a mathematically proven guaranteed set.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work we presented GPU based parallel design and evaluation of Bounded error parameter estimation for system biology models. Inspite of the design complexities due to irregularly available parallelism and unequal task sizes, we show that a GPU based design is able to provide considerable speedups over CPU-only designs. We demonstrate about 18x improvement than 8-core CPU implementation when running on an older Fermi GPU and about 65x speedup when running on latest Kepler GPU designs.

Our approach also re-emphasizes the importance of workload characterization when mapping complex workloads on parallel architectures. The workload was found to be highly irregular in nature with non-deterministic behavior in both spatial and temporal domain. Understanding the key aspects of the workload, namely task creation and task termination pattern, allowed us to effectively design and optimize our approaches. Two execution schemes to consume the dynamically growing workload were proposed -

- *Multi-kernel approach* generates a new kernel run to consume all new tasks generated by previous kernel run.
- *Persistent thread approach* spawns a fixed number of threads that iterate over a global task queue until parameter search is complete.

We argue that the flexibility originally present in Persistent thread approach can allows us to better map a highly irregular workload. We show that *Persistent thread approach* provides room for finer grained task scheduling policies such as *Eager Task Fetch*. With *Eager Task Fetch* optimization we observe a search time improvement of 35% for our sample models. We show that such policies are crucial in improving GPU load inefficiencies that arise due to warp divergence. For a highly irregular workload, *Eager Task Fetch* optimization can improve basic

Persistent thread implementation by over 100% and thus we choose optimized *Persistent thread approach* as our primary GPU approach.

The single GPU implementation was extended to utilize multiple GPUs. We developed a novel fully dynamic load balancer for the persistent thread based GPU kernel. For this we designed a special CPU-GPU interaction mechanism to extract or inject work from CPU while the kernel is live. A MPI based non-blocking master-slave system was added to the host enabling various nodes to exchange work in parallel while the GPU kernel continues to work. We show that this approach performs best in terms of load balancing. Overall, we demonstrated that large speedups are possible on irregularly parallel applications such as Parameter estimation.

7.2 Future Work

We plan to evaluate our implementation on a larger variety of models. A key task here would be include techniques such as automated differentiation to allow automatic code generation for performing guarenteed implementation on an arbitary model. Publicly available guaranteed integration approaches such as as VNODE, LSODA, AWA maybe incorporated to increase the applicability of our approach. To extend the contribution to parallel computing world, the new Persistent thread based dynamic load balancer may be evaluated on irregular workloads from other domains.

REFERENCES

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149. ACM, 2009.

[2] Roberto Barrio. Performance of the taylor series method for odes/daes. *Applied Mathematics and Computation*, 163(2):525–545, 2005.

[3] William J Blake, Mads Kærn, Charles R Cantor, and James J Collins. Noise in eukaryotic gene expression. *Nature*, 422(6932):633–637, 2003.

[4] Martin Burtscher and Keshav Pingali. An efficient cu da implementation of the tree-based barnes hut n-body algorithm. *GPU computing Gems Emerald edition*, page 75, 2011.

[5] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P Sadayappan, and C-W Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[6] Michael B Elowitz, Arnold J Levine, Eric D Siggia, and Peter S Swain. Stochastic gene expression in a single cell. *Science*, 297(5584):1183–1186, 2002.

[7] William Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI-: Portable Parallel Programming with the Message Passing Interface*, volume 1. MIT press, 1999.

[8] Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[9] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[10] Luc Jaulin. *Applied interval analysis: with examples in parameter and state estimation, robust control and robotics*. Springer, 2001.

[11] Luc Jaulin and Eric Walter. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29(4):1053–1064, 1993.

[12] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 395–406. ACM, 2013.

[13] Michel Kieffer and Eric Walter. Interval analysis for guaranteed non-linear parameter and state estimation. *Mathematical and Computer Modelling of Dynamical Systems*, 11(2):171–181, 2005.

[14] Shinichi Kikuchi, Daisuke Tominaga, Masanori Arita, Katsutoshi Takahashi, and Masaru Tomita. Dynamic modeling of genetic networks using genetic algorithm and s-system. *Bioinformatics*, 19(5):643–650, 2003.

[15] Christian Lauterback, Qi Mo, and Dinesh Manocha. Work distribution methods on gpus. *University of North Carolina Technical Report TR009-16*.

[16] Ioannis Lestas, Johan Paulsson, Nicholas E Ross, and Glenn Vinnicombe. Noise in gene regulatory networks. *Automatic Control, IEEE Transactions on*, 53(Special Issue):189–200, 2008.

[17] Bil Lewis, Daniel J Berg, et al. *Multithreaded programming with Pthreads*, volume 2550. Sun Microsystems Press, 1998.

[18] Youdong Lin and Mark A Stadtherr. Guaranteed state and parameter estimation for nonlinear continuous-time systems with bounded-error measurements. *Industrial & Engineering Chemistry Research*, 46(22):7198–7207, 2007.

[19] Charles Loop. Smooth subdivision surfaces based on triangles. 1987.

[20] Skylar W Marvel, Maria A de Luis Balaguer, and Cranos M Williams. Parameter estimation in biological systems using interval methods with parallel processing. In *8th International Workshop on Computational Systems Biology*, pages 129–132, 2011.

[21] Harley H McAdams and Adam Arkin. Simulation of prokaryotic genetic circuits. *Annual review of biophysics and biomolecular structure*, 27(1):199–224, 1998.

[22] Nacim Meslem, Nacim Ramdani, and Yves Candau. Using hybrid automata for set-membership state estimation with uncertain nonlinear continuous-time systems. *Journal of Process Control*, 20(4):481–489, 2010.

[23] Carmen G Moles, Pedro Mendes, and Julio R Banga. Parameter estimation in biochemical pathways: a comparison of global optimization methods. *Genome research*, 13(11):2467–2474, 2003.

[24] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474. IEEE, 2013.

[25] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156. ACM, 2013.

[26] Nedialko S Nedialkov, Kenneth R Jackson, and George F Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1):21–68, 1999.

[27] Markus Neher, Kenneth R Jackson, and Nedialko S Nedialkov. On taylor model based integration of odes. *SIAM Journal on Numerical Analysis*, 45(1):236–262, 2007.

[28] Nvidia. Nvidias next generation cuda compute architecture: Kepler gk110. *White Paper*, 2012.

[29] CUDA Nvidia. Nvidia cuda programming guide, 2011.

[30] Anjul Patney, Mohamed S Ebeida, and John D Owens. Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 99–108. ACM, 2009.

[31] Johan Paulsson. Summing up the noise in gene networks. *Nature*, 427(6973):415–418, 2004.

[32] Linda R Petzold and Uri M Ascher. *Computer methods for ordinary differential equations and differential-algebraic equations*, volume 61. Siam, 1998.

[33] Guilherme P Pezzi, Márcia C Cera, Elton Mathias, and Nicolas Maillard. On-line scheduling of mpi-2 programs with hierarchical work stealing. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 247–254. IEEE, 2007.

[34] Tarek Raıssi, Nacim Ramdani, and Yves Candau. Set membership state and parameter estimation for systems described by nonlinear differential equations. *Automatica*, 40(10):1771–1777, 2004.

[35] Andreas Raue, Clemens Kreutz, Thomas Maiwald, Julie Bachmann, Marcel Schilling, Ursula Klingmüller, and Jens Timmer. Structural and practical identifiability analysis of partially observed dynamical models by exploiting the profile likelihood. *Bioinformatics*, 25(15):1923–1929, 2009.

[36] Robert Rihm. Interval methods for initial value problems in odes. *Topics in Validated Computations*, pages 173–207, 1994.

[37] Philipp Rumschinski, Steffen Borchers, Sandro Bosio, Robert Weismantel, and Rolf Findeisen. Set-base dynamical parameter estimation and model invalidation for biochemical reaction networks. *BMC systems biology*, 4(1):69, 2010.

[38] Vahid Shahrezaei, Julien F Ollivier, and Peter S Swain. Colored extrinsic fluctuations and stochastic gene expression. *Molecular systems biology*, 4(1), 2008.

[39] Stanley Tzeng, Brandon Lloyd, and John D Owens. A gpu task-parallel model with dependency resolution. *Computer*, 45(8):34–41, 2012.

[40] Stanley Tzeng, Anjul Patney, and John D Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[41] Nicholas Wilt. 2 hardware architecture overview. 2012.

[42] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, 2011.

[43] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.