Modelling Performance Loss due to Thread Imbalance in Stochastic Variable-Length SIMT Workloads

Stephen Nicholas Swatman*[†], Ana-Lucia Varbanescu[‡], Attila Krasznahorkay[†], Andy Pimentel*
* University of Amsterdam, Amsterdam, The Netherlands
[†] European Organisation for Nuclear Research, Geneva, Switzerland
[‡] University of Twente, Enschede, The Netherlands

***Abstract*—When designing algorithms for single-instruction multiple-thread (SIMT) devices such as general purpose graphics processing units (GPGPUs), thread imbalance is an important performance consideration. Thread imbalance can emerge in iterative applications where workloads are of variable length, because threads processing larger amounts of work will cause threads with less work to idle. This form of thread imbalance influences the design space of algorithms—particularly in terms of processing granularity—but we lack models to quantify its impact on application performance. In this paper, we present a statistical model for quantifying the performance loss due to thread imbalance for iterative SIMT applications with stochastic, variable-length workloads. Our model is designed to operate with minimal knowledge of the implementation details of the algorithm, relying solely on an understanding of the probability distribution of the lengths of the workloads. We validate our model against a synthetic benchmark based on a Monte Carlo simulation of matrix exponentiation, and show that our model achieves nearly perfect accuracy. Compared to empirical data extracted from real hardware, our model maintains a high degree of accuracy, predicting mean performance loss within a margin of 2%.**

***Index Terms*—SIMT, imbalance, performance modelling**

I. Introduction

As the landscape of high-performance computing has evolved over recent years, single-instruction multiple-thread (SIMT) processors—usually in the form of general-purpose graphics processing units (GPGPUs)—have become popular for high-performance computation in many domains [1]. By sacrificing the independence of individual processing cores, SIMT processors are able to pack significantly more processing cores, and thus provide much more raw processing power, compared to their traditional multiple-instruction multiple-data (MIMD) counterparts [2].

However, not every conceivable computational workload can be efficiently handed off to an SIMT device. The increased raw processing power of these devices comes at the cost of reduced flexibility, and algorithms must be carefully designed to run efficiently on SIMT devices, lest their computational prowess goes to waste. One important consideration when programming SIMT devices is the concept of *thread divergence*. In an SIMT device, a group of threads can—by definition—perform only a single, common instruction at a time; colloquially, these threads run in *lockstep*. Thus, cases where the execution paths of threads diverge will cause some of the threads to be idle. If care is not taken to minimise thread divergence in algorithms designed to run on SIMT devices, it can severely degrade performance [3].

Thread divergence emerges not only in situations with conditional branches in the common *if-else* sense, but it can also arise in iterative processes in the form of *thread imbalance*. When the number of iterations of a loop varies between threads, the result is divergence: threads will be idle until the thread with the largest amount of work has performed the necessary number of iterations. Throughout this paper, we refer to workloads where the number of iterations is not fixed and may differ between threads as *variable-length workloads*. When the number of iterations is described by some probabilistic process, we refer to them as *stochastic* workloads. While it is well understood that thread imbalance in variable-length workloads is detrimental to the performance of SIMT devices [3], [4], we are unaware of any quantitative models that predict exactly how much performance is lost.

The question how we can model the impact of thread imbalance in stochastic variable-length workloads is the core focus of this paper. With this work, we are the first to design and implement an accurate statistical model for the expected performance loss of a given application, given *only* that it is an iterative process, that it is executed on an SIMT device, and that the number of iterations required to complete the process follows a known (albeit arbitrarily complex) distribution. We validate our model using empirical measurements gathered using a dedicated benchmark running on an NVIDIA GPU. The results of this validation show that our model agrees with simulated data with a relative error of less than 0.1%, and that it agrees with measurements taken on a real device within 2%.

Our accurate model can help motivate more precisely the design process of (future) SIMT applications—in particular in terms of processing granularity—in domains where stochastic iterative processes are common, such as machine learning [5], cryptography [6], graph processing [7], and scientific computing [8]. The importance of thread imbalance and granularity is further supported by our own results, which show (in Table I) that thread imbalance in SIMT devices can lead to execution that is nearly four times slower if thread granularity is not chosen carefully.

In short, our paper makes the following contributions:

- We provide a statistical framework for reasoning about the performance loss due to thread imbalance in variable-length workloads on SIMT processors (Section III);
- We assess the accuracy of our model using empirical results gained from a custom synthetic benchmark for this form of performance loss (Section IV);

II. Background

A traditional multi-core CPU architecture consists of a number of processing cores, each of which is equipped with dedicated arithmetic and control hardware. Because all cores in such an architecture possess their own control, they can function largely independently of one another, executing multiple instructions on (potentially) different data. Thus, such an architecture is classified as *multiple-instruction multiple-data* (MIMD) [9]. In recent years, we observe a stark rise in popularity of a different kind of architecture: *single-instruction multiple-thread* (SIMT). SIMT architectures omit the control flow from individual cores in favour of having a larger number of (less flexible) cores and—as a result—more arithmetic prowess compared to a similar MIMD device.

A. Thread Divergence and Imbalance

In an SIMT architecture, multiple threads—which we refer to as a *thread group*—share the same control flow. As a result, instructions on such architectures can only ever be issued to an entire thread group at the same time, rather than to individual threads like on an MIMD architecture. This behaviour is referred to as *lockstep* execution. In this execution model, conditional branches are challenging and are implemented through a process known as *masking*: when a thread group encounters a conditional block, each thread determines whether it should execute or ignore the corresponding instructions. Threads not participating in conditional execution are unable to perform other useful work during this time: they are idle, and computing resources are wasted. As the number of conditional paths—or the length of those paths—grows, more threads are idle for longer periods of time, and we lose more performance. We refer to this behaviour as *thread divergence*, and it can be a significant source of performance degradation [3].

Thread divergence also arises in iterative structures such as loops, which rely on conditional branches that jump back to the start of the loop body. Thus, if one thread in an SIMT processor has concluded the iterative process but another thread has more iterations to perform, their execution paths diverge: the first thread will need to idle until the second completes the loop. We refer to this particular manifestation of threads divergence as *thread imbalance*.

B. Reducing Thread Imbalance

The SIMT programmer's toolbox provides at least two strategies to ameliorate the effects of thread imbalance: changing the thread granularity, and load balancing. Thread granularity refers to the way work is mapped onto the threads of the processor. Most commonly, SIMT programmers map small, independent parts of a workload onto individual threads, but it is also possible to spread that work over multiple threads. This effectively reduces the number of independent jobs executed by the thread group, thereby reducing the degree of imbalance. However, spreading work over multiple threads is often non-trivial. We see increased support for programming at these levels of granularity; NVIDIA, for example, implements so-called *cooperative groups* in version 9.0 of its CUDA platform [10], [11]. Similarly, SYCL features *sub-groups* [12]. Both of these features allow programmers to tune their code to minimise the impact of thread imbalance.

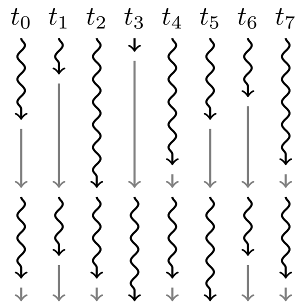
Load balancing, on the other hand, operates by pre-processing the workload of the SIMT processor such that the work performed by each thread group is more balanced [7]. For example, a programmer may choose to sort the workload before offloading it to the SIMT device, guaranteeing that jobs of similar length will end up in the same thread group. Of course, sorting a sufficiently large workload is in itself a costly operation, and approximate solutions—which nevertheless more balanced execution—may be used [13].

Choosing a suitable thread granularity or balancing loads between SIMT threads requires an understanding of how much performance we lose due to imbalance, but we do not currently have quantitative models for this. Rather, these important (and potentially time-consuming) optimisations are often processes of trial and error. Our contribution, therefore, is to provide a quantitative model for the impact of thread imbalance in order to allow application developers to make more informed decisions about the design of their applications.

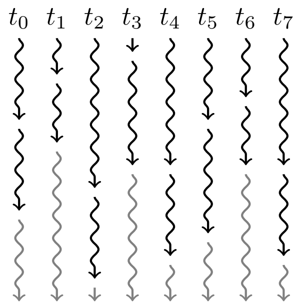
III. Modelling

Throughout this paper we assume that a workload is comprised of an arbitrary number of *units of work* which can be performed independently and in parallel. Each unit is described by a natural number, indicating the number of iterations required to complete it. For example, the workload $\{5, 2, 3, 7\}$ consists of four units of work, requiring 5, 2, 3, and 7 iterations to complete. Executing one iteration has some application-specific computational cost T; this might represent, for example, a number of cycles or an amount of wall-clock time. It follows that the computational cost of a unit of work is given by the number of iterations required to complete it multiplied by the iteration cost T.

As workloads can be arbitrarily large and hardware is inherently limited in its ability to execute processes in parallel, the workload is naturally partitioned into *work groups*. On an SIMT device with two lanes, our previous workload might be partitioned into two work groups: $\{\{5, 2\}, \{3, 7\}\}$. Work groups are analogous to thread groups in the same way that data is analogous to hardware; work groups are mapped onto thread groups for processing, and a single thread group may process many work groups throughout the running time of a program. The mapping of individual units of work onto individual threads is determined by the thread granularity. The manner in which the work groups within a workload are mapped onto thread groups depends on the hardware, and has



(a) The SIMT device runs in lockstep, leading to periods of idleness (straight grey arrows). The cost, including idle time, is 56 for w and 40 for w'.



(b) The MIMD device incurs no performance loss due to lockstep execution (amortised over continuous execution, wavy arrows). The cost is 33 units for w, and 32 for w'.

Fig. 1: An example of 8-way SIMT and MIMD execution of two work groups from a larger workload: $w = \{4, 2, 7, 1, 6, 4, 3, 6\}$ and $w' = \{4, 3, 4, 5, 4, 5, 3, 4\}$. The performance loss due to imbalance is $\mathcal{H}(w) = {}^{56}\!/_{33} = 1.70$ and $\mathcal{H}(w') = {}^{40}\!/_{32} = 1.25$.

no bearing on the rest of our model: on a strict SIMT device, work groups might be executed sequentially by a single thread group, while on an NVIDIA GPU the work groups might be executed in parallel across multiple independent streaming multiprocessors.

Importantly, the idea of imbalance exists only *within* work groups, and there is (by definition) no dependency between different work groups. This allows us, without loss of generality, to study work groups individually. As such, we denote work groups using the symbol w, and we use $|w|$ to denote the size of work group w. Finally, w_i shall denote the number of iterations required to complete the ith unit of work in w.

We define performance loss due to SIMT execution as the computational cost of executing a work group w on a SIMT device, $C_{\text{SIMT}}(w)$, divided by the cost of executing the same work group on an idealised MIMD device, $C_{\text{MIMD}}(w)$. This idealised device has equivalent computational power to the SIMT device, but does not run in lockstep and as such, its performance does not degrade due to thread imbalance. Figure 1 illustrates the execution of a workload on these two devices. Formally, the performance loss of executing a work group w on the SIMT device, $\mathcal{H}(w)$, is defined as:

$$\mathcal{H}(w) = \frac{C_{\text{SIMT}}(w)}{C_{\text{MIMD}}(w)} \tag{1}$$

From our assumptions made about the devices, it follows that $\mathcal{H}(w) \in [1, \infty)$; indeed, since the MIMD device has the same computational power, but is not affected by thread imbalance, it should always perform as well as or better than the SIMT device. In this framework, $\mathcal{H}(w) = 1$ implies that the computational cost of running the work group on the SIMT device is the same as the cost of running it on the MIMD device, and indicates that the work group incurs no performance loss at all. Intuitively, as $|w|$ becomes larger (in other words, as we process more work in parallel), we expect the performance loss to grow accordingly. Finally, $|w| = 1$ implies $\mathcal{H}(w) = 1$, as there is no possibility for a single unit of work to be imbalanced.

A. Modelling SIMT and MIMD Devices

In order to model the computational cost of executing a work group on our SIMT device we must consider the fact that, in lockstep execution, a thread cannot proceed until all threads in its group have completed their required number of iterations. Thus, the threads need to wait for the thread with the largest number of iterations: the *depth* of the work group [14]. Because *all* threads in the group are occupied (albeit possibly idle) throughout the entire process, the total computational cost for the work group w, $C_{\text{SIMT}}(w)$, is given by the following expression, where T represents the computational cost of executing a single iteration:

$$C_{\text{SIMT}}(w) = |w| \max_{i=1}^{|w|} T w_i = T|w| \max_{i=1}^{|w|} w_i \tag{2}$$

Next, we model the computational cost of executing the same work group on our idealised MIMD device. This device executes the units of work in parallel, but can immediately perform meaningful work when previous work is completed such that threads do not incur additional cost by idling. Therefore, the computational cost of executing the work group on our MIMD device is equal to the *sum* of the costs of the individual units. This gives us the following definition of $C_{\text{MIMD}}(w)$, where the cost T to perform one iteration is the same as for the SIMT device:

$$C_{\text{MIMD}}(w) = \sum_{i=1}^{|w|} T w_i = T \sum_{i=1}^{|w|} w_i \tag{3}$$

We can substitute Equations 2 and 3 into Equation 1 to obtain the following expression for the loss of performance:

$$\mathcal{H}(w) = \frac{T|w| \max_{i=1}^{|w|} w_i}{T \sum_{i=1}^{|w|} w_i} = |w| \frac{\max_{i=1}^{|w|} w_i}{\sum_{i=1}^{|w|} w_i} \tag{4}$$

It is worth noting that, in Equation 4, the constant cost factor T is eliminated, which imparts a powerful property on our model: it is wholly independent of the implementation details of the iterative code, as well as the hardware on which it will run. This means that no knowledge about the implementation is required to construct a model of this type. Rather, we only need to know how the number of iterations for each work unit is distributed.

B. Modelling Stochastic Work Groups

From this point forward, we treat our work groups as random samples, which necessitates some change in notation. In a stochastic framework, our work group w becomes a realisation of an independently and identically distributed sample of size $n = |w|$ drawn from a discrete distribution W, such that $w = \{W_1, W_2, \ldots, W_n\}$ and $W_1, W_2, \ldots, W_n \sim W$. Thereby,

$\mathcal{H}(w)$ necessarily becomes a random variable, which we shall denote $X_W(n)$, such that $X_W(n) \sim \mathcal{H}(\{W_1, W_2, \ldots, W_n\})$. This gives the following rephrasing of Equation 4 in terms of random variables:

$$X_W(n) = n \frac{\max_{i=1}^{n} W_i}{\sum_{i=1}^{n} W_i} \tag{5}$$

The idea that iteration counts are drawn from or described by a statistical distribution arises naturally in many kinds of computation. Programs which model or process data from real-world processes might be described—with sufficient domain knowledge—by one of many well-known parametric distributions, such as the Poisson distribution. In many other cases where a closed-form distribution is not available, we can still describe the iteration-counts distribution through simulation.

This stochastic framework also elucidates some of the potential use cases for our model. Indeed, the idea of configuring the thread granularity of a program corresponds directly to changing the size n of the random sample of work lengths, and load balancing corresponds to changing the underlying distribution W; balancing the load of a SIMT program amounts to partitioning it into multiple smaller loads, each of which has a more narrow distribution and—as a result—loses less performance due to imbalance. Because our model captures these effects so directly, we believe it to be a useful tool in reasoning about these kinds of optimisations.

Given that the distribution of our work unit lengths, W, is discrete, we could—in theory—find the distribution of the performance loss $X_W(n)$ by enumerating all possible values of $W_1, W_2, \ldots, W_n$. However, this solution runs in $\mathcal{O}(|\text{supp}(W)|^n)$ time and is not generally computationally tractable: a distribution supported by only ten possible outcomes would lead to 10^{32} possible combinations in a model for thirty-two parallel units of work.

We proceed by creating an equivalent model that is more computationally efficient. In order to do so, we first re-write the numerator in Equation 5 using order statistics: given that $W_1, W_2, \ldots, W_n$ are random variables drawn from W, we can sort these values in ascending order such that the ith order statistic, denoted $W_{(i:n)}$, is the kth smallest value. Thus, $W_{(1:n)}$ represents the smallest value in the sample, $W_{(2:n)}$ represents the second-smallest value, and so forth. Given that there are n values in total, $W_{(n:n)}$ naturally represents the largest value in the sample—the maximum. Next, we reduce the denominator of the fraction to a single random variable. This new random variable, denoted $Z_W(n, a)$, represents the distribution of the sum of n random variables drawn from W, given the fact that the maximum of that sample is known to be a. Since the maximum value is know from the numerator, we set $a = W_{(n:n)}$ and obtain the following equation:

$$X_W(n) = n \frac{a}{Z_W(n, a)} \text{ where } a \sim W_{(n:n)} \tag{6}$$

The number of random variables required to compute the performance loss $X_W(n)$ in Equation 6 is now only two (as opposed to the n random variables required to compute Equation 5): $W_{(n:n)}$ and $Z_W(n, W_{(n:n)})$. This greatly reduces the combinatorics required to enumerate all possible outcomes of this division; indeed, we find that the number of outcomes is now bound by $\mathcal{O}(n|\text{supp}(W)|^2)$. We proceed by computing the distribution of the sample maximum $W_{(n:n)}$ in Section III-B1, and the distribution of the sum of a sample given its maximum, $Z_W(n, a)$, in Section III-B2. Finally, we calculate the ratio distribution $X_W(n)$ in Section III-C.

1) Distribution of the Sample Maximum: In order to find an analytical solution for the distribution of the maximum of a series of i.i.d. random variables $W_1, W_2, \ldots, W_n \sim W$, we note that this maximum is equal to the n-th order statistic of that sample, denoted $W_{(n:n)}$. The distribution of order statistics for discrete distributions is well understood [15], and the distribution of $W_{(n:n)}$ is described by the following probability mass function:

$$\begin{aligned} f_{W_{(n:n)}}(x) &= P(W \leq x)^n - P(W < x)^n \\ &= F_W(x)^n - (F_W(x) - f_W(x))^n \end{aligned} \tag{7}$$

It is worth noting that the distribution of the maximum value preserves the support of the original distribution; intuitively, if one were to roll a six-sided die ten times and select the maximum roll, that outcome would never be more than six, nor would it ever be lower than one.

2) Distribution of the Sample Sum: In Equation 6, $Z_W(n, a)$ denotes the distribution of the sum of an i.i.d. sample $W_1, W_2, \ldots, W_n \sim W$ given a priori knowledge that the maximum value in that sample is equal to a, thus:

$$f_{Z_W(n,a)}(x) = P\left(x = \sum_{i=1}^{n} W_i \;\middle|\; W_{(n:n)} = a\right) \tag{8}$$

In order to derive this distribution we construct a parametric, *non-normalized* function $g_W(x; i, m)$, which denotes the probability of the sum of i values drawn from W, each of which is no greater than m, being equal to x. This function is defined inductively from a degenerate distribution supported at zero (capturing the idea that additive processes start at zero) as follows:

$$g_W(x; i, m) = \begin{cases} [x = 0] & \text{if } i = 0 \\ \sum_{j=0}^{m} g_W(x - j; i - 1, m) f_W(j) & \text{otherwise} \end{cases} \tag{9}$$

By its definition, $g_W(x; n, a)$ nearly models the target distribution but it crucially fails to model that, in for the maximum of a sample to equal a, the sample must contain a at least once. In order to resolve this, we subtract the function $g_W(x; n, a - 1)$, which intuitively models the probability of *never* drawing a, point-wise. Then, we only need to normalize the resulting probabilities to find an expression for the distribution of the sample sum given its maximum:

$$f_{Z_W(n,a)}(x) = \frac{g_W(x;n,a) - g_W(x;n,a-1)}{\sum_{j=a}^{na} (g_W(j;n,a) - g_W(j;n,a-1))} \quad (10)$$

C. Modelling Performance Loss

In order to find the distribution of the loss of performance in our model, we calculate the ratio distribution between $W_{(n:n)}$ and $Z_W(n, W_{(n:n)})$. This process does not generally permit a closed form solution, but this distribution can be calculated through a brute-force approach. Indeed, since all distributions involved in the process are discrete and because their supports are subsets of the integers, our ratio distribution is supported by a subset of $\mathbb{Q}$ which must be countable. The probability of each supported outcome is then given by summing up the probabilities of all outcomes which map to the same irreducible fraction:

$$f_{X_W(n)}(x) = \sum_{a,b \in \mathbb{N},\ n\frac{a}{b}=x} f_{W_{(n:n)}}(a) f_{Z_{W(n,a)}}(b) \quad (11)$$

It should be noted that, for this process to be tractable, the support of the underlying distribution W must be finite. If the support of W is infinite, then the support of $W_{(n:n)}$ is infinite and, consequently, the support of $X_W(n)$ is also infinite; as such, we would not be able to meaningfully enumerate the possible outcomes. We can resolve this issue by approximating W using a finite truncated distribution.

The enumeration of the possible outcomes, and the computation of their probabilities, marks the end of the construction of our model. This model gives us insight into the distribution of the loss of performance for a randomly drawn work group. For example, $\mathbb{E}(X_W(n))$ gives the expected loss of performance for a work group. Because most real-world workloads will have many thousands of work groups, the performance loss of such workloads will naturally tend towards the expected value as a consequence of the law of large numbers.

IV. Evaluation

In order to evaluate the predictive power of our model, we construct a benchmark based on a simple iterative process: matrix exponentiation. Our validation process has two stages. First, we simulate a synthetic workload—with exponents drawn from a given distribution—using a Monte Carlo method. This process allows us to compute the loss of performance as given by Equation 4 exactly, and we will refer to this as the *simulated* loss of performance. Second, we execute the matrix exponentiation kernel—with the iteration counts given by our earlier simulation—on a real-world SIMT device; by measuring the execution time of each of the work groups, we can determine the *measured* loss of performance. Both of these metrics can then be compared to each other, as well as to the *modelled* loss of performance, to confirm whether they are in agreement.

```
Data: Source distribution $W$, number of trials $r$, work group
      size $n$
Result: Simulated results $x_0, \ldots, x_r$ and measured results
        $y_0, \ldots, y_r$
for $i \leftarrow 0$ to $r$ do
    for $j \leftarrow 0$ to $n$ do
        $p_{i,j} \leftarrow W$;
        $M_{i,j} \leftarrow$ random matrix;
    end
end
for $i \leftarrow 0$ to $r$ do
    $C_{\mathrm{SIMT}} \leftarrow 0$;
    $C_{\mathrm{MIMD}} \leftarrow 0$;
    parallel for $j \leftarrow 0$ to $n$ do
        $M'_{i,j} \leftarrow I$;
        $c_j \leftarrow$ clock();
        for $k \leftarrow 0$ to $p_{i,j}$ do
            $M'_{i,j} \leftarrow M'_{i,j} M_{i,j}$;
            $c_{\mathrm{MIMD},j} \leftarrow$ clock();
        end
        $c_{\mathrm{SIMT},j} \leftarrow$ clock();
        $C_{\mathrm{SIMT}} \leftarrow C_{\mathrm{SIMT}} + c_{\mathrm{SIMT},j} - c_j$;
        $C_{\mathrm{MIMD}} \leftarrow C_{\mathrm{MIMD}} + c_{\mathrm{MIMD},j} - c_j$;
    end
    $x_i \leftarrow n \frac{\max_{j=0}^{n} p_{i,j}}{\sum_{j=0}^{n} p_{i,j}}$;
    $y_i \leftarrow \frac{C_{\mathrm{SIMT}}}{C_{\mathrm{MIMD}}}$;
end
```

Algorithm 1: Benchmark of the loss of performance when running work units drawn from a given distribution. We assume execution on an SIMT machine—thus, the parallel block is executed in lockstep—to determine C_{SIMT}, and *emulate* the equivalent MIMD execution to calculate C_{MIMD}.

A. Benchmark Design

Our benchmark operates by computing powers of square dense matrices M^p (with $p \in \mathbb{N}$) through repeated multiplication[1]. This problem matches the class of algorithms targeted by our model very well: each work unit iteratively executes matrix multiplication operations, and the number of iterations is equal to the exponent p. In addition, the cost of each iteration is fixed: for our benchmark, we operate on 16×16 matrices. As discussed in Section III, the execution time of each individual step is irrelevant to the outcome of our model, and as such the size of the matrices should not matter. However, we find that if the run-time of each individual step is very small (as it is for, say, 3×3 matrices), we incur additional noise in our measurements. Pseudo-code for our benchmark is given in Algorithm 1.

Note that our validation strategy relies on an emulation of MIMD behaviour on the SIMT device. To this end, we measure the cost of performing a computation in accordance with the SIMT model as the time between the start of the computation and the time at which *all* threads are done. In contrast, we emulate the behaviour of an MIMD device by

[1]The astute reader may have noticed that, because square matrices under multiplication form a monoid, this operation can be performed more efficiently in $\mathcal{O}(\log_2 p)$ time. However, this defeats the purpose of our benchmark, and is therefore not implemented.

TABLE I: Descriptive statistics of the probability distributions used to validate our model, comparing modelled, simulated, and measured distributions.

		Modelled		Simulated		Measured	
Dist.	n	μ_A	$\eta_{S,A}$	μ_S	$\eta_{M,S}$	μ_M	$\eta_{A,M}$
$\mathrm{B}(40, 0.5)$	2	1.090	0.01%	1.090	0.86%	1.099	0.86%
	4	1.163	0.00%	1.163	0.83%	1.173	0.84%
	8	1.225	0.02%	1.225	0.85%	1.236	0.88%
	16	1.278	0.01%	1.278	0.75%	1.287	0.74%
	32	1.325	0.01%	1.325	0.11%	1.326	0.12%
$\mathrm{Geo}(0.05)$	2	1.476	0.05%	1.477	1.36%	1.497	1.43%
	4	2.047	0.03%	2.047	0.91%	2.065	0.89%
	8	2.668	0.01%	2.668	0.77%	2.689	0.77%
	16	3.317	0.02%	3.317	0.12%	3.321	0.14%
	32	3.979	0.02%	3.979	1.64%	3.915	1.59%
$\mathrm{Pois}(30)$	2	1.104	0.00%	1.104	0.57%	1.110	0.58%
	4	1.191	0.00%	1.191	0.55%	1.198	0.55%
	8	1.268	0.01%	1.268	0.57%	1.275	0.57%
	16	1.335	0.00%	1.335	0.44%	1.341	0.45%
	32	1.397	0.00%	1.397	0.26%	1.393	0.25%
$\mathrm{U}(20, 40)$	2	1.118	0.00%	1.118	0.57%	1.124	0.58%
	4	1.213	0.01%	1.213	0.55%	1.220	0.54%
	8	1.275	0.01%	1.275	0.58%	1.282	0.59%
	16	1.309	0.01%	1.309	0.53%	1.316	0.54%
	32	1.326	0.00%	1.326	0.16%	1.328	0.16%
$\mathrm{NB}(5, 0.3)$	2	1.301	0.02%	1.301	1.64%	1.323	1.68%
	4	1.587	0.06%	1.586	1.38%	1.608	1.34%
	8	1.860	0.04%	1.860	1.32%	1.885	1.30%
	16	2.123	0.06%	2.124	0.84%	2.142	0.90%
	32	2.375	0.01%	2.375	0.72%	2.358	0.70%

μ_A denotes the expected performance loss as derived analytically using our model, μ_S denotes the mean performance loss derived from our simulation, and μ_M denotes the mean of the measured data. $\eta_{a,b}$ denotes the relative error between the means μ_a and μ_b: $\eta_{a,b} = |(\mu_a - \mu_b)/\mu_a|$.

calculating the time at which each thread is ready to proceed with further useful work, without being constrained by the lockstep execution model.

We evaluate the accuracy of our model using the following five underlying probability distributions for work unit lengths:

• $\mathrm{B}(40, 0.5)$	Binomial with $n = 40$ and $p = 0.5$.
• $\mathrm{Geo}(0.05)$	Geometric with $p = 0.05$.
• $\mathrm{Pois}(30)$	Poisson with $\lambda = 30$.
• $\mathrm{U}(20, 40)$	Uniform with $a = 20$ and $b = 40$.
• $\mathrm{NB}(5, 0.3)$	Negative binomial with $r = 5$ and $p = 0.3$.

These distributions have been selected for evaluation because they: (1) occur naturally and commonly in real-world processes; (2) they have a wide range of supports (including infinite ones); and (3) they have a wide variety of shapes (including fat-tailed and thin-tailed). Please note that our model is not limited to such well-behaved distributions; instead, the model works for arbitrary discrete distributions. Even a categorical distribution assigning an arbitrary probability mass to each of a set of natural numbers can be used with our modelling strategy.

B. Experimental Setup

We have implemented our benchmark in C++; the Monte Carlo simulation of work items follows the MT19937 pseudo-random number generator provided by the C++ standard library [16]. The code for the SIMT device was written in CUDA [11], and it was compiled and executed on the CUDA 11.5 platform. The compiler was configured to emit code for Compute Capability and PTX version 8.0 (the most recent version supported by our target GPU). The host code was compiled using gcc version 9.4.0. Our results were generated on a node of the DAS-6 cluster [17] using an NVIDIA A100 PCI-e GPU with $40\,\mathrm{GB}$ of HBM2 memory based on the *Ampere* microarchitecture [18]. The kernels were launched with 256 threads per block. For each experiment, we used 2^{18} work groups to ensure the availability of sufficient device memory to store the input matrices.

As our GPU is of a post-*Volta* architecture, it is equipped with *Independent Thread Scheduling* which implies that it is not strictly an SIMT device [19]. In order to more accurately simulate true SIMT behaviour, explicit thread group-level synchronisation[2] was added to the kernel. This also allows us to evaluate the effects of Independent Thread Scheduling by disabling this synchronisation, as explored in Section IV-D.

As discussed in Section III-C, some of our underlying distributions had to be truncated to ensure finite support; in these cases, the acceptable loss of precision was set to a threshold of $\epsilon = 10^{-6}$.

C. Validation Results

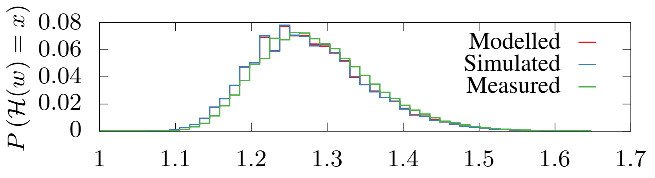
In order to evaluate the quality of our model, we calculate the expected and mean performance loss for our modelled, simulated, and measured results. The nature of our problem makes it difficult to apply many of the usual goodness-of-fit tests; some of the probabilities we model are extremely small, leading to very small numbers of expected observations, which invalidates the use of Pearson's χ^2 and similar tests. Because our data is discrete, the Kolmogorov-Smirnov test (and other statistical tests for continuous distributions) are not applicable. This need not be a problem, however: as discussed in Section III-C, we posit that the mean is one of the most meaningful statistics for our model, as it allows us to estimate the performance loss for entire workloads.

The results of our analysis are shown in Table I. These results indicate that our model manages to predict the mean performance loss of both the Monte Carlo simulation and the measurements on the GPU with a high level of accuracy: the relative error between our model and the Monte Carlo simulation never exceeds 0.1% in our validation, and the relative error between our model and the timing results from the GPU—a noisy environment—never exceeds 2%. A visual comparison between the modelled, simulated, and measured results for a subset of distributions[3] is shown in Figure 2. These figures confirm—on a visual level—our previous findings that the model agrees well with simulated and measured data.

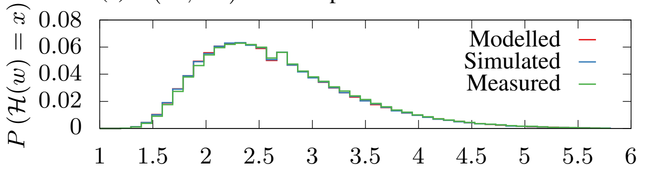
We observe that finitely-supported distributions (namely, the binomial and uniform distributions) are modelled more accurately, due to the fact that they do not require truncation; the truncation required to make our model work with infinite distributions (in this case, the geometric, Poisson, and negative

[2]In CUDA terminology, this is referred to as warp-level synchronisation.

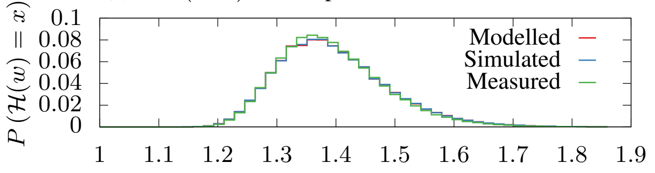
[3]The remaining distributions exhibit similar levels of agreement, but were not included due to space limitations.



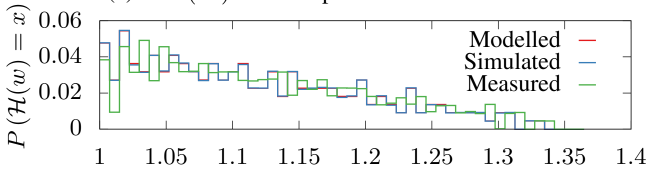
(a) $B(40, 0.5)$ with 16 parallel units of work.



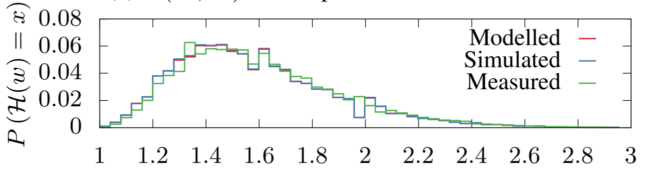
(b) $\text{Geo}(0.05)$ with 8 parallel units of work.



(c) $\text{Pois}(30)$ with 32 parallel units of work.



(d) $U(20, 40)$ with 2 parallel units of work.



(e) $\text{NB}(5, 0.3)$ with 4 parallel units of work.

Fig. 2: Comparison between the modelled, simulated, and empirically measured performance loss distributions for a subset of distributions and degrees of parallelism.

binomial distributions) discards a small but non-zero amount of information.

D. *Effects of Independent Thread Scheduling*

Due to the *Independent Thread Scheduling* architectural feature, the A100 GPU used in our validation is not a true SIMT device, as thread group-level synchronicity is not guaranteed by the hardware. As discussed in Section IV-B, we use explicit thread group synchronisation to more accurately emulate SIMT behaviour, but it remains prudent to investigate the effect of this non-SIMT architecture on our model. To this end, we have performed all our measurements with the explicit thread group synchronisation disabled and computed the difference in performance loss between the two sets of results. These differences are shown in Figure 3.

We observe that, in all of our benchmarks, the relative error between the experiments with and without explicit thread group-level synchronisation is less than 1%. We conclude that the presence of Independent Thread Scheduling has little impact on the accuracy of our model and, therefore, our model remains adequate even for future devices which may not follow the SIMT execution model in the strictest sense. It is worth noting that these results are consistent with the notion that Independent Thread Scheduling primarily serves to guarantee forward progress in parallel algorithms, rather than to provide significant gains in performance [20]. While we are not aware of any existing studies examining the effects of this microarchitectural feature in and of itself, our results are consistent with studies which—in passing—examine its impact on the performance of other applications [6], [21].

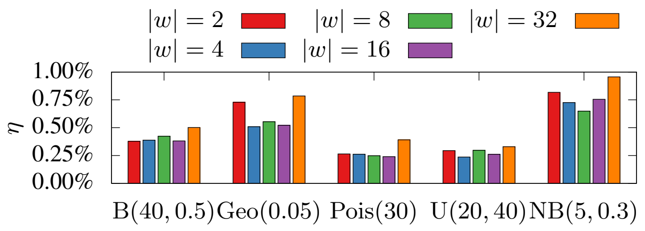


Fig. 3: Relative error of measured performance loss with explicit thread group-level synchronisation enabled and disabled, demonstrating the impact of *Independent Thread Scheduling*.

E. *Limitations*

The main limitation of our model is that the MIMD device which underpins it is inherently theoretical; however, we are not aware of any real-world pair of SIMT-MIMD processing devices with exactly equivalent computational power. This has consequences for the predictive power of our model, as we cannot use it to make concrete predictions about program runtime; instead, our model can be used—for example—to rank different implementations, as they are compared against the same theoretical optimum.

V. Related Work

Performance models for SIMT devices—GPUs in particular—have been widely studied. A survey of existing modelling techniques, as well as a framework for classifying models, is given by Madougou, Varbanescu, Laat, *et al.* [22]. Within their framework, the model presented in this paper could be classified as a model for optimisation space exploration at a coarse abstraction level requiring zero knowledge about the hardware. While Madougou, Varbanescu, Laat, *et al.* specify a class of modelling techniques described as *statistical*, this class does not accurately describes our model: rather than extracting the impact of the design space from a posteriori knowledge of execution time through data-centric, machine-learning based methods, we use statistical methods to make a priori predictions of a program's performance, which fits in the category of analytical models. This categorisation would imply that the closest existing models to ours are models such as *Eiger* [23], *Stargazer* [24], and the model by Zhang, Hu, Li, *et al.* [25]. However, all these models require knowledge about the implementation of the SIMT program, which our model does not; therefore, we believe that our model can be applied much earlier in the application development process.

VI. Conclusion

In this paper, we have presented a model that gives insight into the performance of iterative applications with stochastic variable-length workloads on SIMT architectures such as GPUs; using our model, we can estimate the performance loss that such applications incur due to thread imbalance. Our model is designed specifically to require as little a priori knowledge as possible, relying solely on an understanding of the statistical distribution of the amount of work that is to be processed by each thread. This information can be extracted from domain knowledge or from simulation through an existing implementation, thus requiring little to no information about the details of an SIMT implementation of the program, and allowing our model to be used in the early stages of application development.

Our model is shown to be accurate within a relative error of 0.1% compared to a Monte Carlo simulation, and within 2% when compared to measurements on a real device. We believe our model can be used to quantitatively motivate and guide important optimisations of SIMT programs, in particular thread coarsening and load balancing. We aim to apply our model to real-world applications in domain science in the near future. In addition, we aim to investigate how our model can be used in the exploration and (automated) tuning of the design space of SIMT applications.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank the Advanced School for Computing and Imaging for providing us with access to the DAS-6 computing cluster, which has been invaluable for the validation of our work.

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, *et al.*, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007. DOI: 10.1111/j.1467-8659.2007.01012.x.

[2] V. W. Lee, C. Kim, J. Chhugani, *et al.*, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: Association for Computing Machinery, 2010, pp. 451–460, ISBN: 9781450300537. DOI: 10.1145/1815961.1816021.

[3] P. Bialas and A. Strzelecki, "Benchmarking the cost of thread divergence in CUDA," in *International Conference on Parallel Processing and Applied Mathematics*, Krakow, Poland: Springer International Publishing, 2016, pp. 570–579. DOI: 10.1007/978-3-319-32149-3_53.

[4] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, Florida, United States of America: IEEE, 2014, pp. 284–295. DOI: 10.1109/HPCA.2014.6835939.

[5] J. R. Cheng and M. Gen, "Accelerating genetic algorithms with GPU computing: A selective overview," *Computers & Industrial Engineering*, vol. 128, pp. 514–525, 2019, ISSN: 0360-8352. DOI: 10.1016/j.cie.2018.12.067.

[6] L. Oden and J. Keller, "Improving cryptanalytic applications with stochastic runtimes on GPUs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Portland, Oregon, United States of America: IEEE, 2021, pp. 459–468. DOI: 10.1109/IPDPSW52791.2021.00077.

[7] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, San Antonio, TX, USA: Association for Computing Machinery, 2011, pp. 267–276, ISBN: 9781450301190. DOI: 10.1145/1941553.1941590.

[8] L. Murray, "GPU acceleration of Runge–Kutta integrators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 94–101, 2012. DOI: 10.1109/TPDS.2011.61.

[9] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273.

[10] M. Harris and K. Perelygin. "Cooperative Groups: Flexible CUDA thread programming." (2017).

[11] NVIDIA Corporation. "CUDA C++ programming guide." (2022).

[12] T. K. S. W. Group. "SYCL 2020 specification (revision 4)," Khronos Group. (2021).

[13] S. Frey, G. Reina, and T. Ertl, "SIMT microscheduling: Reducing thread stalling in divergent iterative algorithms," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Munich, Germany: IEEE, 2012, pp. 399–406. DOI: 10.1109/PDP.2012.62.

[14] Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ parallel max-flow algorithm," *Journal of Algorithms*, vol. 3, no. 2, pp. 128–146, 1982, ISSN: 0196-6774. DOI: 10.1016/0196-6774(82)90013-X.

[15] H. N. Nagaraja, "Order statistics from discrete distributions," *Statistics*, vol. 23, no. 3, pp. 189–216, 1992. DOI: 10.1080/02331889208802365.

[16] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998, ISSN: 1049-3301. DOI: 10.1145/272991.272995.

[17] H. Bal, D. Epema, C. de Laat, *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 05, pp. 54–63, May 2016, ISSN: 1558-0814. DOI: 10.1109/MC.2016.127.

[18] NVIDIA Corporation. "NVIDIA A100 Tensor Core GPU architecture." (2020).

[19] ——, "NVIDIA Tesla V100 GPU architecture." (2017).

[20] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018. DOI: 10.1109/MM.2018.022071134.

[21] H. Anzt and J. Dongarra, "A Jaccard weights kernel leveraging independent thread scheduling on GPUs," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Lyon, France: IEEE, 2018, pp. 229–232. DOI: 10.1109/CAHPC.2018.8645946.

[22] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort, "The landscape of GPGPU performance modeling tools," *Parallel Computing*, vol. 56, pp. 18–33, 2016, ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.04.002.

[23] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili, "Eiger: A framework for the automated synthesis of statistical performance models," in *2012 19th International Conference on High Performance Computing*, Pune, India: IEEE, 2012, pp. 1–6. DOI: 10.1109/HiPC.2012.6507525.

[24] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based GPU design space exploration," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, New Brunswick, New Jersey, United States of America: IEEE, 2012, pp. 2–13. DOI: 10.1109/ISPASS.2012.6189201.

[25] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, Dalian, China: IEEE, 2011, pp. 149–158. DOI: 10.1109/NAS.2011.51.