Statistical Models for the Overhead of Thread Divergence in Variable Length Workloads on SIMT Architectures

Stephen Nicholas Swatman[1,2]

18 February 2022

[1]CERN [2]University of Amsterdam

Prelude

- Please note I am not a statistician: take the statistical part of this with a grain of salt.
- Please feel free to ask any questions as they come to you during the talk.

Introduction

- Like physics, computing has many useful models!
- Performance models can describe, predict, and prescribe thing about computer programs:
    - Roofline models
    - Polyhedral models
    - Simulator models
    - Statistical models
- Can we use models to make parallel ACTS software even better?

	HPC	HEP
Descriptive	*“The application achieves 18.4 GFLOPs because the L2 cache bandwidth is saturated.”*	*“Neutrinos rarely interact with matter, because the weak nuclear force has such a short range.”*
Predictive	*“This algorithm will be able to utilise roughly 80% of the targeted GPU’s capacity.”*	*“The Higgs boson should decay into a charm-anticharm pair, but we’ve not observed this yet.”*
Prescriptive	*“It is worth investing dozens of person-hours into developing this implementation”*	*“It is worth investing billions of currency units into a new accelerator.”*

Introduction

- In this talk: can we model the *suitability* of some algorithm for execution on a GPU?
- This is a vague term, suitability for GPU computing can mean many things:
  - Is there sufficient parallelism in the algorithm? For example, SSSP vs APSP.
  - Is there no risk of exhausting the device's limited memory? For example, *scrypt* vs SHA-256.
  - Is there not too much chance for threads to diverge?
- We will focus, in this talk, on thread divergence.

Recap

- CPU, MIMT: fully independent processing cores, each of which has its own silicon for arithmetic and control flow.
- GPU, SIMT: replace the control flow with more arithmetic, increasing raw performance but reducing flexibility.
    - A group of threads share the same control flow, meaning that if *one* thread does something, they *all*.
- Thread divergence slows a program down if threads are constantly waiting for other threads hogging the control flow.

```
__global__ void diverge() {
    int tid = threadIdx.x + blockIdx.x *
     blockDim.x;

    if (tid % 4 == 0) {
        expensive_computation_1();
    } else if (tid % 4 == 1) {
        expensive_computation_2();
    } else if (tid % 4 == 2) {
        expensive_computation_3();
    } else if (tid % 4 == 3) {
        expensive_computation_4();
    }
}
```

Recap

- Food for thought: the code on this slide contains zero if-statements.
- Does this code suffer from thread divergence?

```
1  __global__ void matrix_pow(
2      matrix ms[], uint ps[]
3  ) {
4      int tid = threadIdx.x + blockIdx.x *
        blockDim.x;
5
6      matrix m = ms[tid];
7      matrix a = m;
8
9      for (int i = 0; i < ps[tid]; ++i) {
10         a *= m;
11     }
12
13     ms[tid] = a;
14 }
```

Recap

- Food for thought: the code on this slide contains zero if-statements.
- Does this code suffer from thread divergence?
- Yes, it does!
- If the number of times the loop is executes differs between threads, this is thread divergence.
- Remember: a loop is just a Dijkstra-approved replacement for a conditional go-to statement.

```
1  __global__ void matrix_pow(
2      matrix ms[], uint ps[]
3  ) {
4      int tid = threadIdx.x + blockIdx.x *
        blockDim.x;
5
6      matrix m = ms[tid];
7      matrix a = m;
8
9      for (int i = 0; i < ps[tid]; ++i) {
10         a *= m;
11     }
12
13     ms[tid] = a;
14 }
```

Thesis

- When we think of thread divergence, we tend to think of suitability as binary:
  - Algorithms which *are* suitable: pixel shading, horizontal sums, convolutions, etc.
  - Algorithms which are *not* suitable: branch-and-bound, many graph algorithms, etc.
- In reality, this is a spectrum: some problems have more thread divergence than others.
- Problem is there exists a massive void between the two ends of this scale, and we lack models to place algorithms on the spectrum.
- This spectrum is (if you squint your eyes) equivalent to the question of whether we should run algorithms in a one-element-per-thread mapping or a one-element-per-warp mapping.
- We're seeing increasing support for this intermediate range, a prime example being cooperative groups, where you can elegantly split up a warp into multiple parts.

Metrics

- Let's define the suitability of a particular implementation as the ratio between its computational cost on our real-world SIMT device and its computational cost on an idealised MIMT device of equivalent processing power.
- For the idealized MIMT device, imagine the exact same processor but with additional control flow installed so that each thread can run independently.
- Note that $h(f) = 1$ implies an ideal scenario, and higher is worse (less GPU-friendly).

$$h(f) = \frac{C_{\mathrm{SIMT}}(f)}{C_{\mathrm{MIMT}}(f)}$$

Algorithms

- Solving this modeling problem for arbitrary algorithms is the holy grail, we will consider a smaller class of algorithms:
- Formally, we restrict ourselves to the class of parallel anamorphisms (g, p) where $g \in \mathcal{O}(1)$.
- Less formally, we have some non-recursive endomorphism $g : A \to A$ and our algorithm is equivalent to $f : [A \times \mathbb{N}] \to [A]$ such that $f = \mathrm{map}\ (\lambda(v, n).g^{\circ n} v)$.
- Even less formally, if we have some operation which we can perform iteratively, repeatedly apply that operation to each element of some container, each element being processed in parallel. The number of times we apply the operation to each element is determined by come condition, or equivalently is just a given number.

Algorithms

- This might seem unreasonably specific, but I argue that this problem is common:
- Navigation between surfaces is an algorithm of this type given $g : \mathrm{TrackState} \to \mathrm{TrackState} \times \mathrm{Intersection}$ and $p : \mathrm{TrackState} \to \mathrm{Bool}$ s.t. g calculates the intersection with the closest surface.
- Propagation of a particle is an algorithm of this type given g a Runge-Kutta step[1].

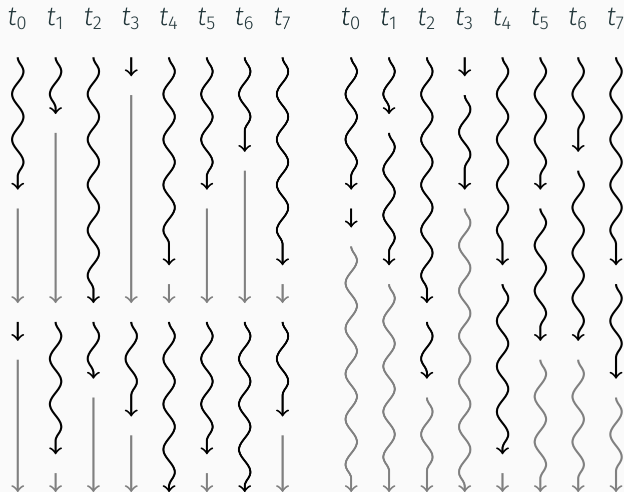
[1] Propagation is a hylomorphism, but this is its anamorphic part; the catamorphic part executes in $\mathcal{O}(1)$ time and is not very interesting.

Statistics

- This problem is still not quite solvable, we need *one* more piece of a priori knowledge.
- We need the distribution $\mathcal{F}$ of the sizes of our units of work.
  - We will look at some well-behaved distributions today: uniform, binomial, Poisson, and geometric.
  - Works for generalised distributions, including those defined as outcome-probability pairs!
- Such that one unit of work is $w = \{X_1, X_2, \ldots, X_n\}$ where $X_1, X_2, \ldots X_n \text{ i.i.d.} \sim \mathcal{F}$.
- As we are taking a discrete number of steps, will assume $\mathcal{F}$ is discrete, i.e. $X_1, X_2, \ldots, X_n \in \mathbb{N}$.
- Run-time is given by $t(x) \in \Theta(x)$, which we will now promptly forget about because it is irrelevant (we are looking for a ratio).

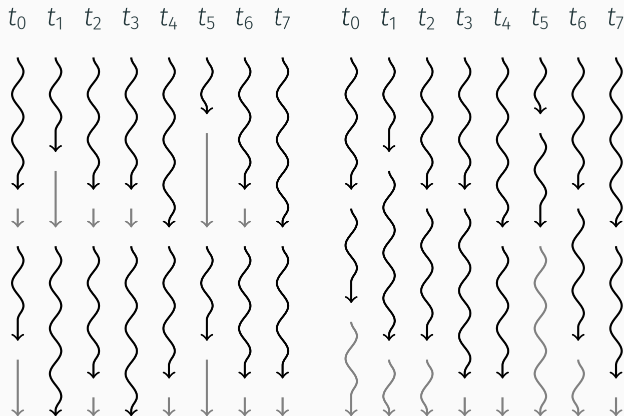
Example

- As an example, we see that work load is quite widely distributed.
  - $\{4, 2, 7, 1, 6, 4, 3, 6\}$
  - $\{1, 4, 2, 3, 5, 4, 5, 3\}$
- The SIMT computational cost is 96.
- The MIMT computational cost is 60.
- Thus, the thread divergence overhead is $\frac{96}{60} = 1.60$



Example

- In this example, the distribution of work lengths is much tighter.
  - $\{4, 3, 4, 4, 5, 2, 4, 5\}$
  - $\{3, 5, 4, 5, 4, 3, 4, 4\}$
- On the SIMT machine, the computational cost is now 80.
- On the MIMT machine, it is 63.
- Thus, the overhead is now only $\frac{80}{63} = 1.27$.



Model

- Key insight: we can express the computational cost for our SIMT and MIMT machines quite succinctly.
- Work a set of work items w (w.l.o.g. w equals the number of threads), we find:
  - For our SIMT device, the computational cost is equal to the w times the *span* of the computation:

$$C_{SIMT}(w) = \max\{w_0, w_1, \ldots, w_{|w|}\}$$

  - For our MIMT device, the computational cost is equal to the *work* of the computation:

$$C_{MIMT}(w) = \sum_{i=0}^{|w|} w_i$$

- To find the distribution of our SIMT computation cost, we look towards order statistics:
- Sort sample in ascending order: $X_{(0)}, X_{(1)}, \ldots, X_{(|w|)}$.
- Then find the distribution of $X_{(|w|)}$...
- This becomes trivial if we are looking for the maximum!
- To calculate the PMF of this distribution:

$$\begin{aligned} P(X_{(n)} = x) &= P(X_{(n)} \leq x) - P(X_{(n)} < x) \\ &= P(X \leq x)^n - P(X_{(n)} < x) \\ &= P(X \leq x)^n - P(X < x)^n \\ &= F(x)^n - (F(x) - f(x))^n \end{aligned}$$

- Finding the distribution of our MIMT computation cost is even easier if we're talking about well-behaved source distributions:
  - $\sum_{i=0}^{|w|}(X_i \sim B(n,p)) \sim B(n|w|,p)$
  - $\sum_{i=0}^{|w|}(X_i \sim \mathrm{Pois}(\lambda)) \sim \mathrm{Pois}(\lambda|w|)$
  - $\sum_{i=0}^{|w|}(X_i \sim G(p)) \sim \mathrm{NB}(|w|,p)$
- Ironically, finding the sum of uniform distributions is hardest, but can be found using the same technique we use for arbitrary distributions:
  - Add two variables together by finding all combinations that sum to each supporting outcome (multiplying their probabilities).
  - Extend to arbitrary number of distributions via induction.

Overhead

- Now it gets a little more funky, because we need the ratio distribution between the SIMT model and the MIMT model.
- First, take a look at the support of our two distributions $\mathcal{F}_{\mathrm{SIMT}}$ and $\mathcal{F}_{\mathrm{MIMT}}$ with source distribution $\mathcal{F}$.
  - $\mathrm{supp}(\mathcal{F}_{\mathrm{SIMT}}) = \mathrm{supp}(\mathcal{F})$.
  - $\mathcal{F}_{\mathrm{MIMT}}$ preserves the (in)finity of $\mathcal{F}$'s support.
- Thus, we can proceed by case analysis:
  - If $\mathrm{supp}(\mathcal{F})$ is finite, $\mathrm{supp}(\frac{\mathcal{F}_{\mathrm{SIMT}}}{\mathcal{F}_{\mathrm{MIMT}}}) = \{\frac{a}{b} \mid a \in \mathrm{supp}(\mathcal{F}_{\mathrm{SIMT}}),\ b \in \mathrm{supp}(\mathcal{F}_{\mathrm{MIMT}}),\ b \neq 0\} \in \mathbb{Q}$.
  - If $\mathrm{supp}(\mathcal{F})$ is infinite, the same still works theoretically but it becomes a real pain in the behind, so we find truncated distributions $\mathcal{F}'_{\mathrm{SIMT}}$ with $\mathrm{supp}(\mathcal{F}'_{\mathrm{SIMT}}) = \{0, \ldots, m\}$ s.t. $\mathrm{CDF}(m) \geq 0.999$ or some other threshold. Then proceed as above.

Overhead

- Elephant in the room: numerator and denominator are obviously not independent.
- Introduces error, and allows impossible scenarios (e.g. $h < 1$).
- Solutions:
  - Quantify error and accept if sufficiently small.
  - Rewrite equation for overhead to eliminate dependence, i.e. using distributivity of maximum over division.

$$\frac{\max\{X_0, X_1, \ldots, X_n\}}{\sum_{i=0}^{n} X_i} = \max\left\{\frac{X_0}{\sum_{i=0}^{n} X_i}, \frac{X_1}{\sum_{i=0}^{n} X_i}, \ldots, \frac{X_n}{\sum_{i=0}^{n} X_i}\right\}$$
$$= \max\left\{\frac{X_0}{X_0 + \sum(\{X_0, X_1, \ldots, X_n\} \setminus \{X_0\})}, \ldots\right\}$$

- Work in progress, help welcome!

Validation

- Validation strategy: try this model for a simple mini-app that calculates matrix powers[2].
- Note that this fits our requirements exactly: $p : M \to M = \cdot$ and $p \in \mathcal{O}(1)$.
- Reduce, as much as possible, overhead from memory and external factors.
- We use 8×8 matrices because:
  1. If p is too computationally inexpensive, we suffer from far more noise.
  2. Seems pretty common in what we're doing.
- Powers distributed binomially, geometrically, and Poissonially (?).

[2]Because square matrices under multiplication form a monoid, A^n can be computed in $\mathcal{O}(\log_2 n)$ time, but that defeats the purpose of the exercise, so we use a $\mathcal{O}(n)$ implementation.

Validation

- Launch threads in blocks of 32.
- Reduce measurement error due to context switching by artificially reducing occupancy:
  - Kernel comes with shared memory “hogging” parameter which limits SM capacity.
- Optionally configurable seed for some degree of determinism.
- Mini-app executed on my own GTX 1660 Ti (not noiseless, preliminary measurement).
- Tested for 1, 2, 4, 8, 16, and 32 threads (more than that is meaningless on an NVIDIA GPU).
- 131072 samples gathered for each configuration (combination of distribution and thread count).

Validation

Validation step	Affected by	Purpose
Statistical	Quality of our statistical model.	Allow a concise, preferably closed form expression for the expected overhead of an arbitrary application.
Sampled	Underlying probability distribution.	Provides an empirical estimate of the overhead distribution through repeated sampling, but is not found analytically.
Measured	Additional sources of overhead on the GPU.	End-goal of modelling process, achieving agreement with this is the holy grail.

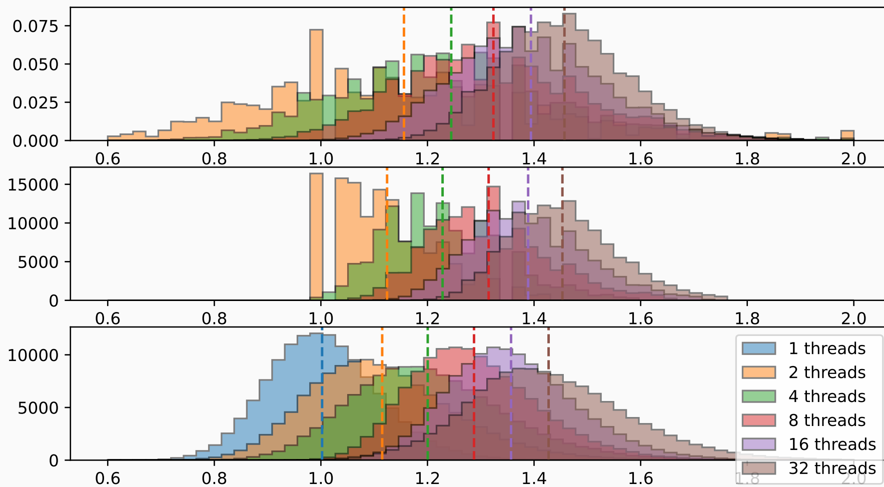
“All models are wrong, but some are useful”

– George Box

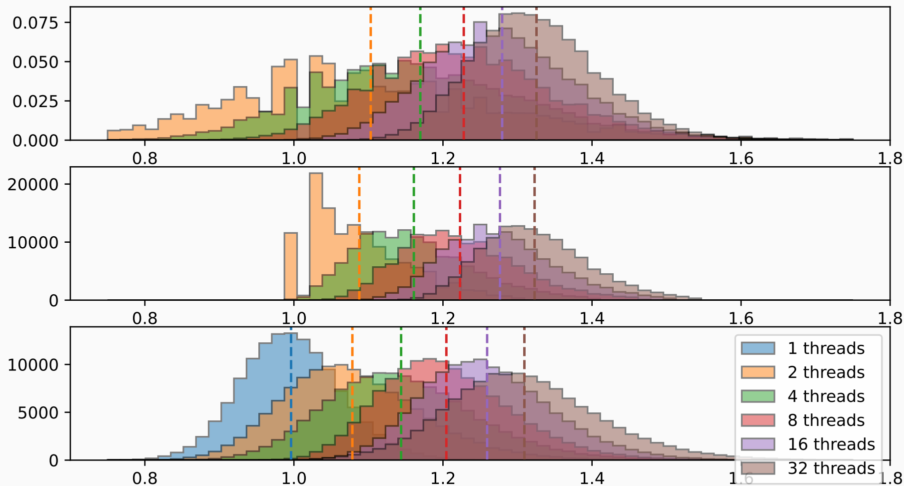
Management of Expectations

- Complete agreement between all three sources of data is – right now – out of scope.
- Shape of estimated distributions is not correct due to previously noted issues.
    - However, mean values (which are really what we're looking for here) should be more robust.
- Overhead on the GPU that we do not understand yet.
- Also: estimated distributions will look rough because we're histogramming the rational numbers.

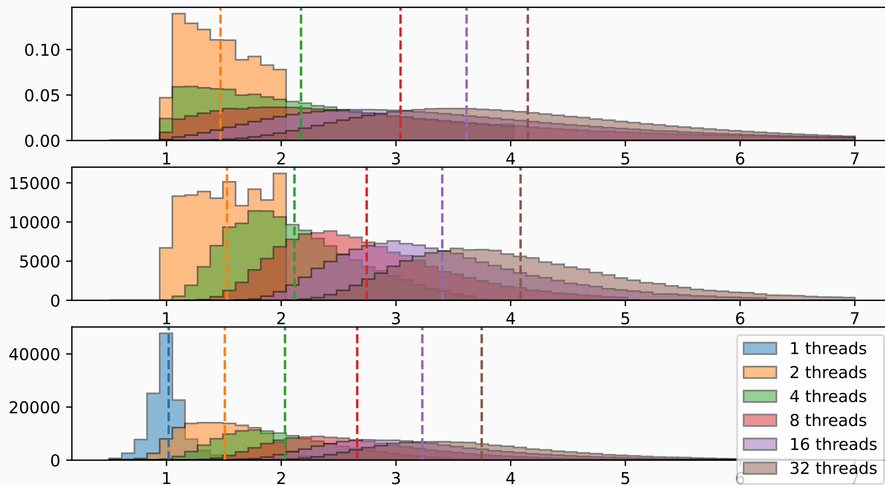
Results



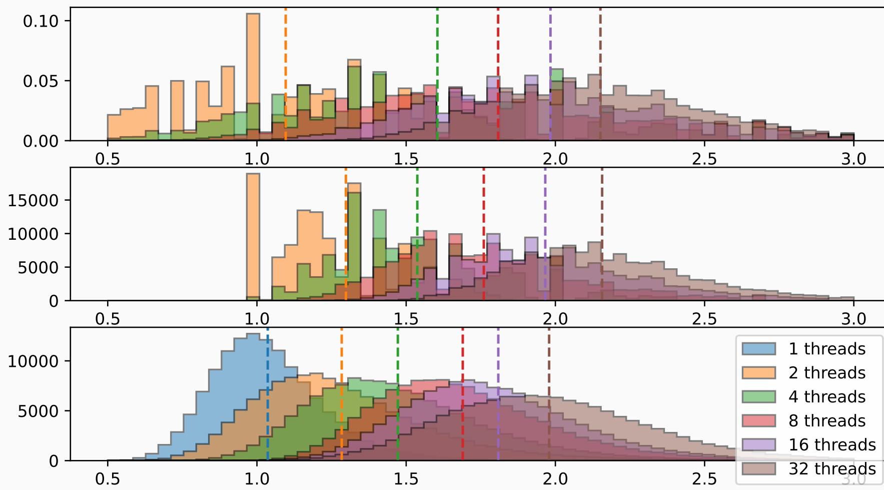
Results



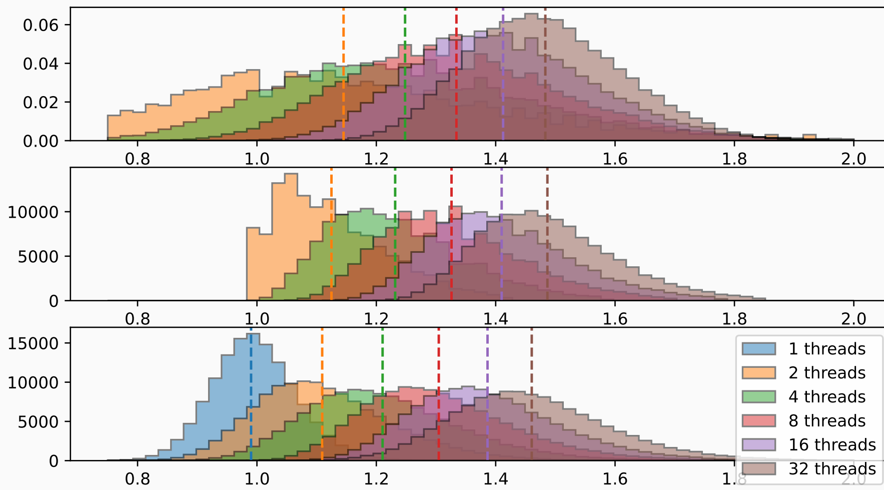
Results



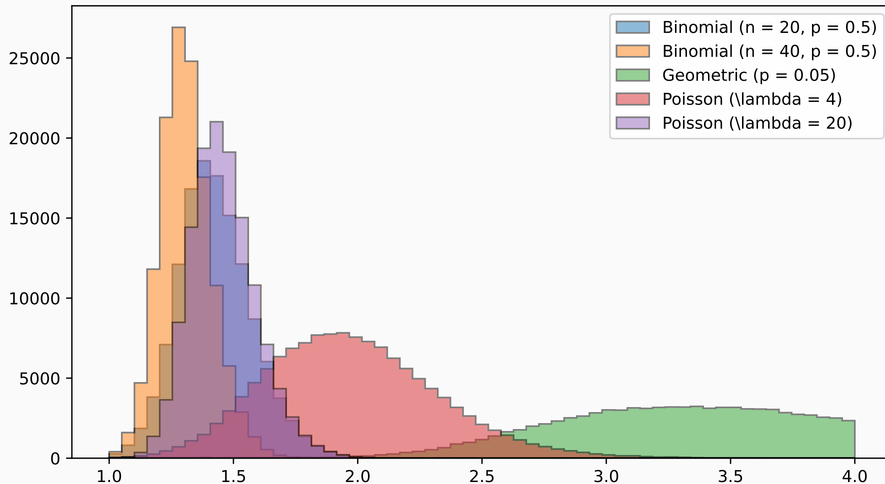
Results



Results



Bonus plot



Conclusions

- Performance models can give powerful insight into non-functional properties of programs.
  - Including those that do not exist yet.
- Presented today: a performance model for the predicted overhead of running a parallel computation in a lock-stepped SIMT machine.
- Computed distributions are not quite right, but promising results for some distributions, and somewhat decent correspondence between intermediate validation step and true measurements.