Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Technische
Universität
Berlin

berlin

Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs

Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, Volker Markl

DaMoN '19, Amsterdam, The Netherlands

Hash aggregation on GPUs

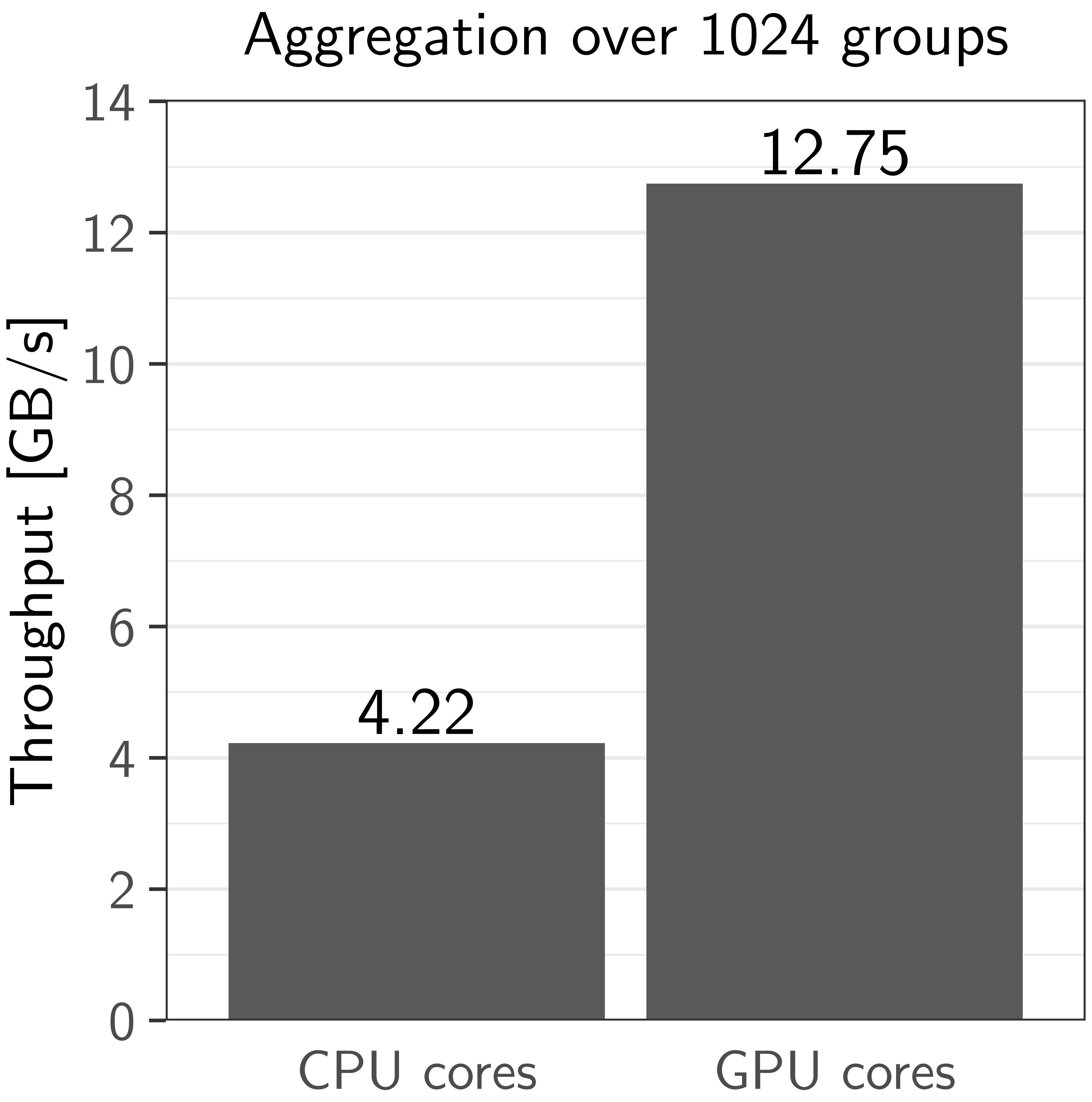
Hash aggregation:

- Used to implement `GROUP BY` and `DISTINCT`.
- Can be significantly accelerated on the GPU.

Example:

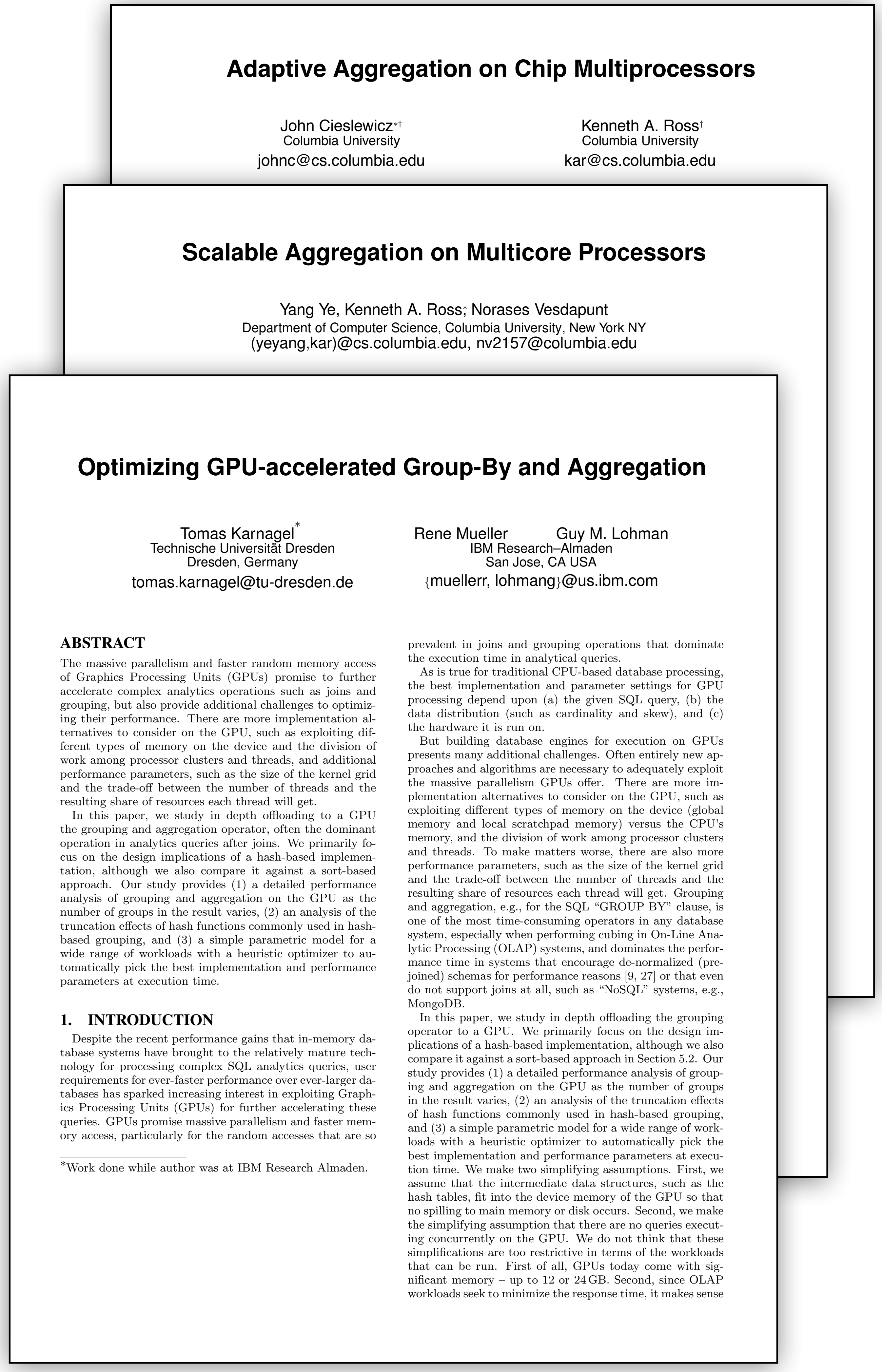
- Query:
  `SELECT G, sum(A) FROM R GROUP BY G`
- Processor: AMD A10-7850K APU.
  - CPU and GPU cores integrated on the same die.

➡ **Aggregation on GPU cores 1.6× – 4.8× faster across different group cardinalities.**



Previous work

- Hash aggregation extensively studied on CPUs.
- Only a single in-depth study on GPUs: Karnagel et al., *Optimizing GPU-accelerated Group-By and Aggregation*, ADMS@VLDB, 2015
  - Evaluated influence of **parallelization strategies** and **thread configuration** based on group cardinality.
  - Formulated heuristics based on a **single NVIDIA Kepler GPU.**



Adaptive Aggregation on Chip Multiprocessors

John Cieslewicz*†
Columbia University
johnc@cs.columbia.edu

Kenneth A. Ross†
Columbia University
kar@cs.columbia.edu

Scalable Aggregation on Multicore Processors

Yang Ye, Kenneth A. Ross; Norases Vesdapunt
Department of Computer Science, Columbia University, New York NY
(yeyang,kar)@cs.columbia.edu, nv2157@columbia.edu

Optimizing GPU-accelerated Group-By and Aggregation

Tomas Karnagel[*]
Technische Universität Dresden
Dresden, Germany
tomas.karnagel@tu-dresden.de

Rene Mueller Guy M. Lohman
IBM Research–Almaden
San Jose, CA USA
{muellerr, lohmang}@us.ibm.com

ABSTRACT

The massive parallelism and faster random memory access of Graphics Processing Units (GPUs) promise to further accelerate complex analytics operations such as joins and grouping, but also provide additional challenges to optimizing their performance. There are more implementation alternatives to consider on the GPU, such as exploiting different types of memory on the device and the division of work among processor clusters and threads, and additional performance parameters, such as the size of the kernel grid and the trade-off between the number of threads and the resulting share of resources each thread will get.

In this paper, we study in depth offloading to a GPU the grouping and aggregation operator, often the dominant operation in analytics queries after joins. We primarily focus on the design implications of a hash-based implementation, although we also compare it against a sort-based approach. Our study provides (1) a detailed performance analysis of grouping and aggregation on the GPU as the number of groups in the result varies, (2) an analysis of the truncation effects of hash functions commonly used in hash-based grouping, and (3) a simple parametric model for a wide range of workloads with a heuristic optimizer to automatically pick the best implementation and performance parameters at execution time.

1. INTRODUCTION

Despite the recent performance gains that in-memory database systems have brought to the relatively mature technology for processing complex SQL analytics queries, user requirements for ever-faster performance over ever-larger databases has sparked increasing interest in exploiting Graphics Processing Units (GPUs) for further accelerating these queries. GPUs promise massive parallelism and faster memory access, particularly for the random accesses that are so prevalent in joins and grouping operations that dominate the execution time in analytical queries.

As is true for traditional CPU-based database processing, the best implementation and parameter settings for GPU processing depend upon (a) the given SQL query, (b) the data distribution (such as cardinality and skew), and (c) the hardware it is run on.

But building database engines for execution on GPUs presents many additional challenges. Often entirely new approaches and algorithms are necessary to adequately exploit the massive parallelism GPUs offer. There are more implementation alternatives to consider on the GPU, such as exploiting different types of memory on the device (global memory and local scratchpad memory) versus the CPU's memory, and the division of work among processor clusters and threads. To make matters worse, there are also more performance parameters, such as the size of the kernel grid and the trade-off between the number of threads and the resulting share of resources each thread will get. Grouping and aggregation, e.g., for the SQL "GROUP BY" clause, is one of the most time-consuming operators in any database system, especially when performing cubing in On-Line Analytic Processing (OLAP) systems, and dominates the performance time in systems that encourage de-normalized (pre-joined) schemas for performance reasons [9, 27] or that even do not support joins at all, such as "NoSQL" systems, e.g., MongoDB.

In this paper, we study in depth offloading the grouping operator to a GPU. We primarily focus on the design implications of a hash-based implementation, although we also compare it against a sort-based approach in Section 5.2. Our study provides (1) a detailed performance analysis of grouping and aggregation on the GPU as the number of groups in the result varies, (2) an analysis of the truncation effects of hash functions commonly used in hash-based grouping, and (3) a simple parametric model for a wide range of workloads with a heuristic optimizer to automatically pick the best implementation and performance parameters at execution time. We make two simplifying assumptions. First, we assume that the intermediate data structures, such as the hash tables, fit into the device memory of the GPU so that no spilling to main memory or disk occurs. Second, we make the simplifying assumption that there are no queries executing concurrently on the GPU. We do not think that these simplifications are too restrictive in terms of the workloads that can be run. First of all, GPUs today come with significant memory – up to 12 or 24 GB. Second, since OLAP workloads seek to minimize the response time, it makes sense

[*] Work done while author was at IBM Research Almaden.

Do these heuristics yield good performance on other GPUs?

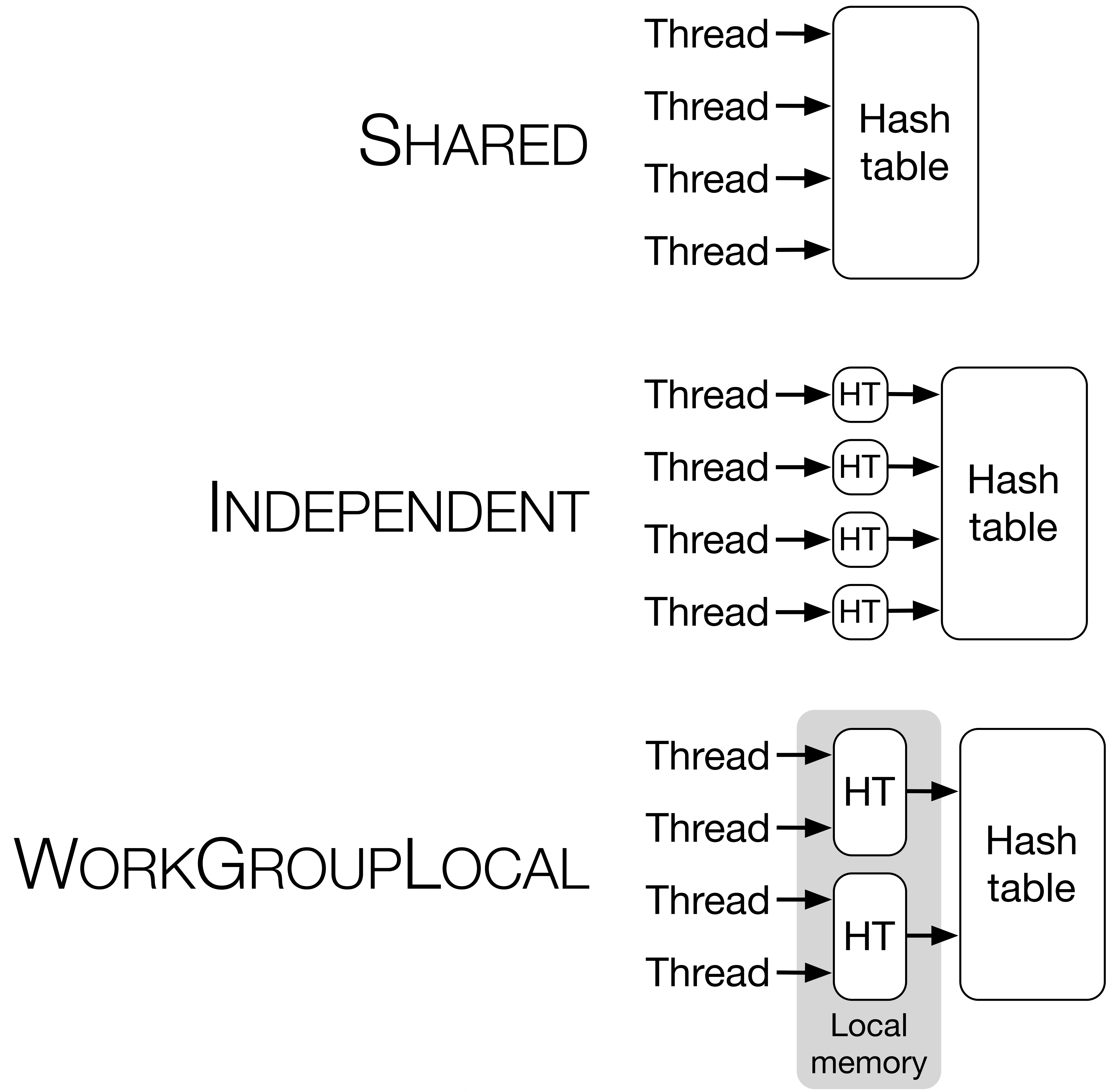
Part 1

Performance analysis of hash aggregation on various GPUs

Tested GPUs

GPU	Microarchitecture	Integration
Tesla K40m	Kepler	PCIe 3.0
GeForce GTX 980	Maxwell	PCIe 3.0
GeForce GTX 1080	Pascal	PCIe 3.0
Tesla V100	Volta	NVLink 2.0
A10-7850K	Graphics Core Next 2nd Gen.	on die
Radeon R9 Fury	Graphics Core Next 3rd Gen.	PCIe 3.0

Parallelization strategies

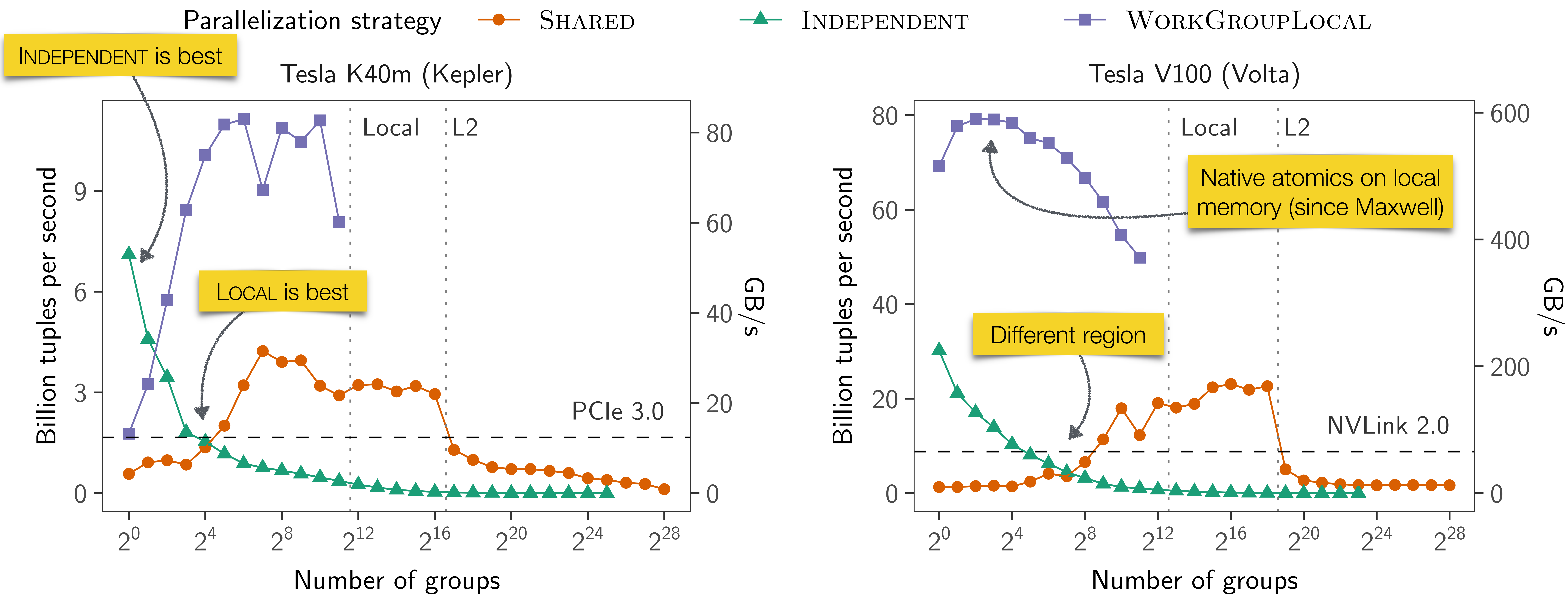


- Concurrent accesses to same hash bucket resolved with atomics.
- SHARED and INDEPENDENT also commonly used on CPUs.
- WORKGROUPLOCAL uses fast local GPU memory.
- Fastest strategy is data and query dependent (amount of contention and cache efficiency).

How do these parallelization strategies perform on different GPUs?

Parallelization strategies

```
SELECT G, sum(A)
FROM R GROUP BY G;
```



Independent is best

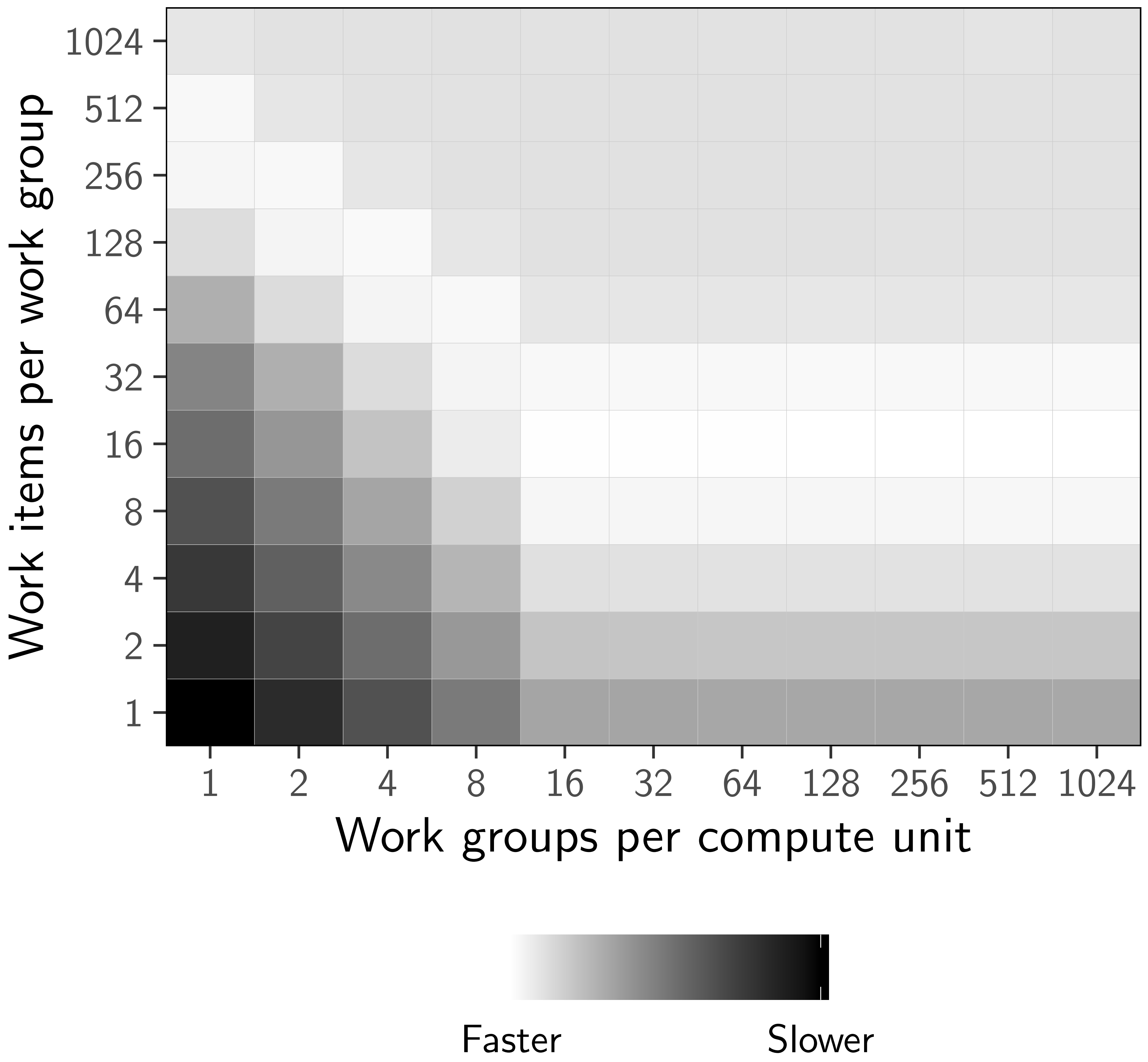
Local is best

Native atomics on local memory (since Maxwell)

Different region

Independent aggregation not competitive on newer GPUs that implement fast atomics on local memory.

Thread configurations



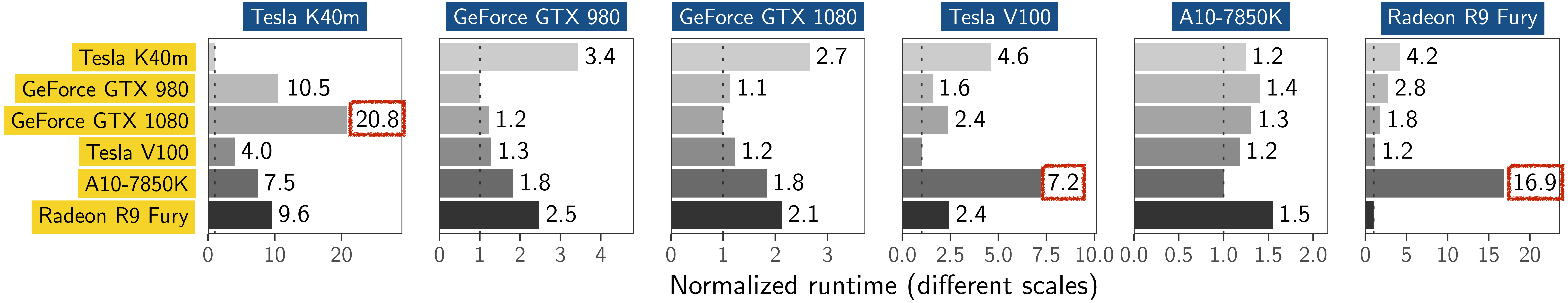
- Number of threads determined by two variables (OpenCL terminology).
  - *Work groups per compute unit*
  - *Work items per work group*
- Performance influenced by hardware and kernel properties.
  - Warp size, register file, TLB cache, ...
  - Number of registers used in kernel, local memory usage, ...

Can we find optimal thread configurations across GPUs?

Performance penalty

```
SELECT G, sum(A)
FROM R GROUP BY G;
```

- When a thread configuration optimized for a specific GPU (rows) is executed on another GPU (columns).



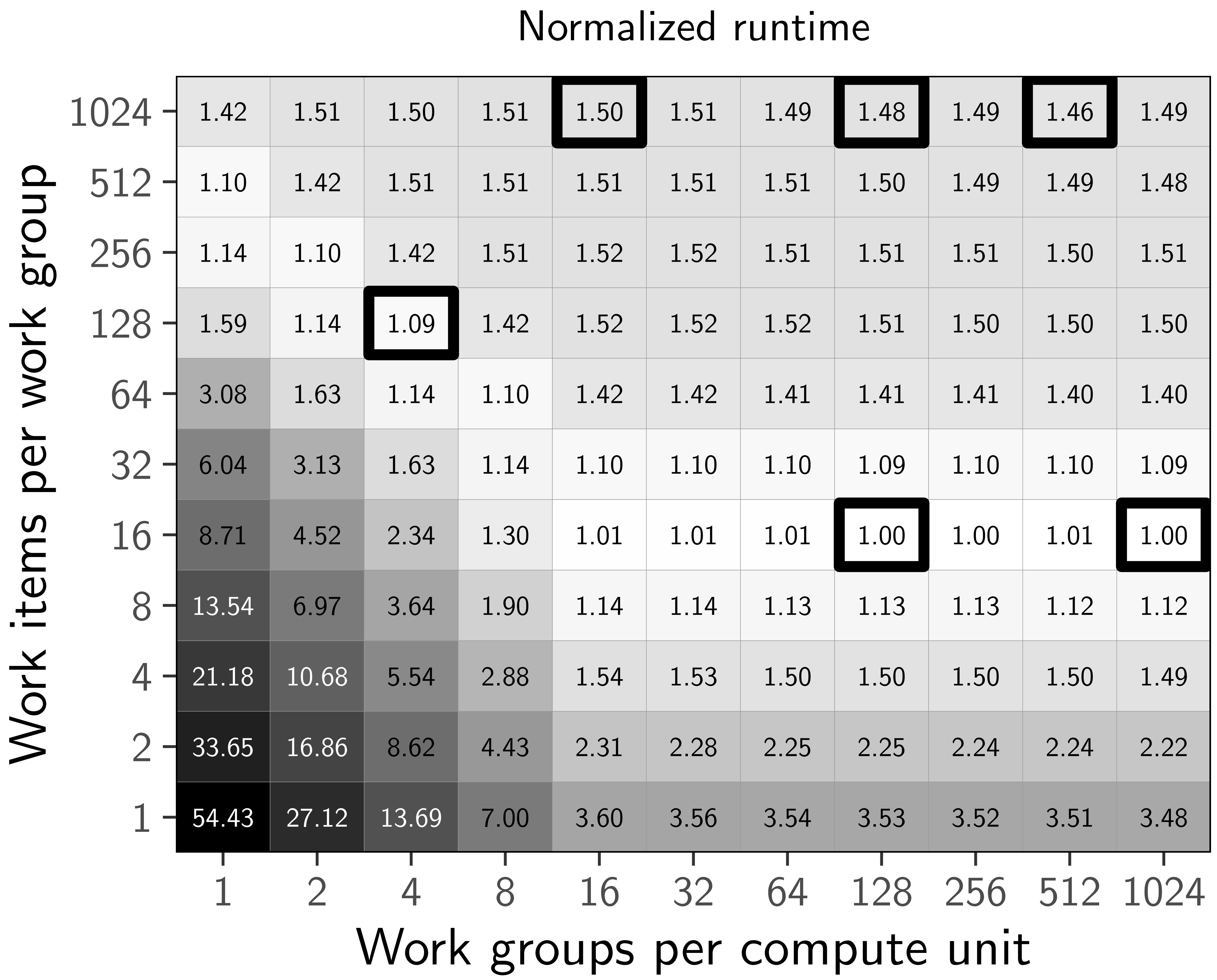
The optimal thread configuration is highly GPU-dependent.

Full evaluation takes hours. How can we find fast thread configurations efficiently?

Part 2

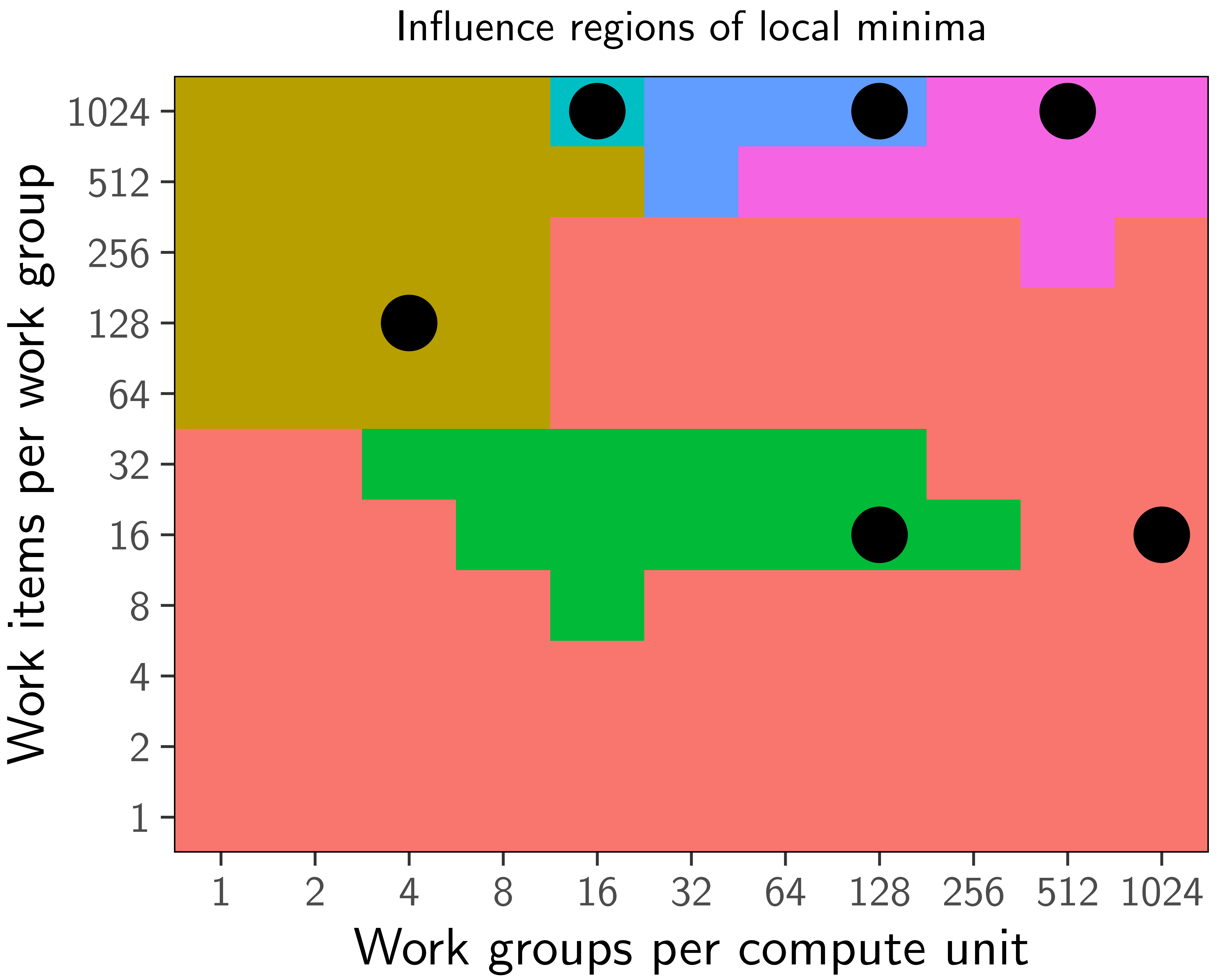
Finding fast thread configurations

Thread configuration search space



Normalized runtime

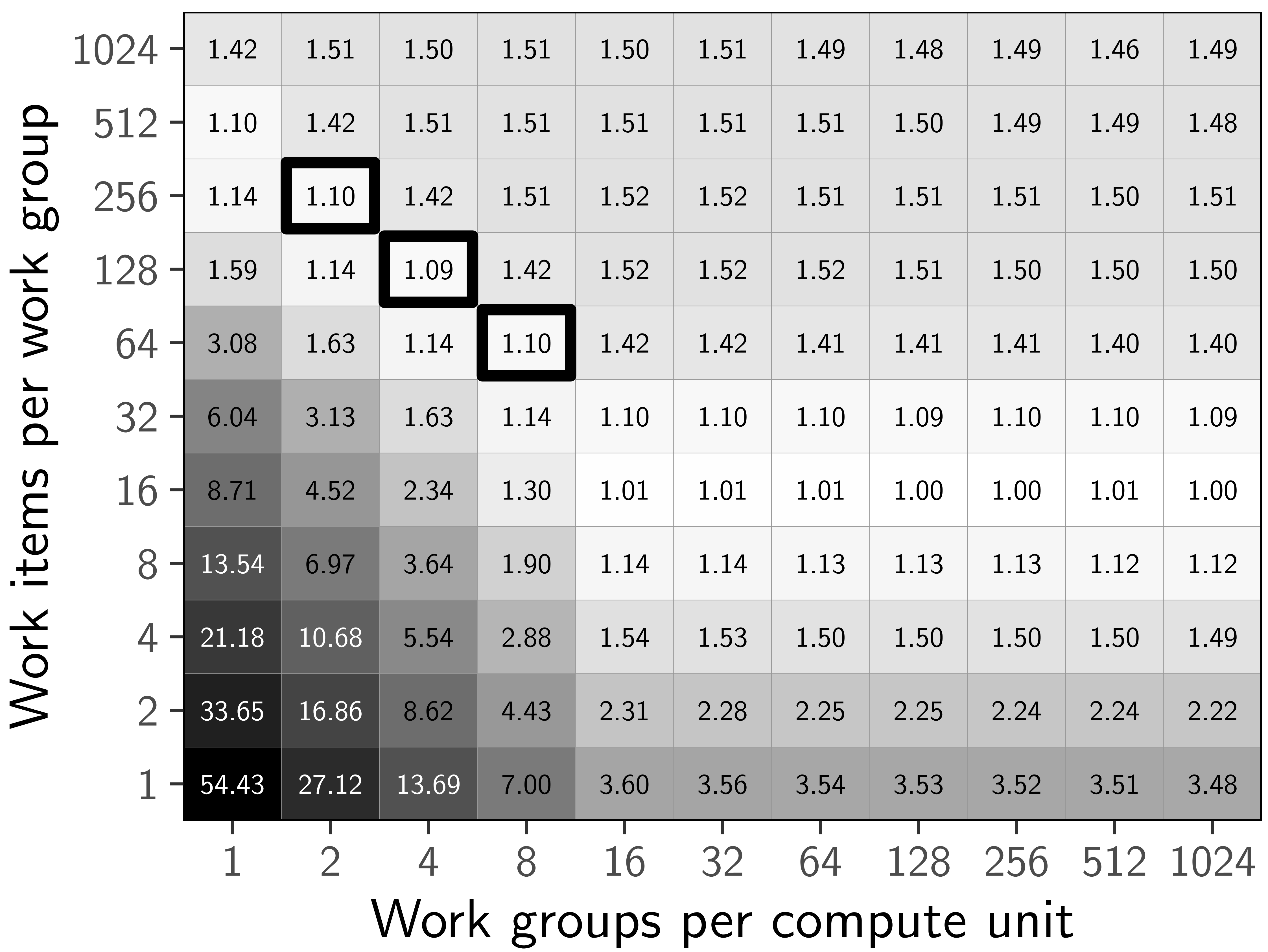
	1	2	4	8	16	32	64	128	256	512	1024
1024	1.42	1.51	1.50	1.51	1.50	1.51	1.49	1.48	1.49	1.46	1.49
512	1.10	1.42	1.51	1.51	1.51	1.51	1.51	1.50	1.49	1.49	1.48
256	1.14	1.10	1.42	1.51	1.52	1.52	1.51	1.51	1.51	1.50	1.51
128	1.59	1.14	1.09	1.42	1.52	1.52	1.52	1.51	1.50	1.50	1.50
64	3.08	1.63	1.14	1.10	1.42	1.42	1.41	1.41	1.41	1.40	1.40
32	6.04	3.13	1.63	1.14	1.10	1.10	1.10	1.09	1.10	1.10	1.09
16	8.71	4.52	2.34	1.30	1.01	1.01	1.01	1.00	1.00	1.01	1.00
8	13.54	6.97	3.64	1.90	1.14	1.14	1.13	1.13	1.13	1.12	1.12
4	21.18	10.68	5.54	2.88	1.54	1.53	1.50	1.50	1.50	1.50	1.49
2	33.65	16.86	8.62	4.43	2.31	2.28	2.25	2.25	2.24	2.24	2.22
1	54.43	27.12	13.69	7.00	3.60	3.56	3.54	3.53	3.52	3.51	3.48



Influence regions of local minima

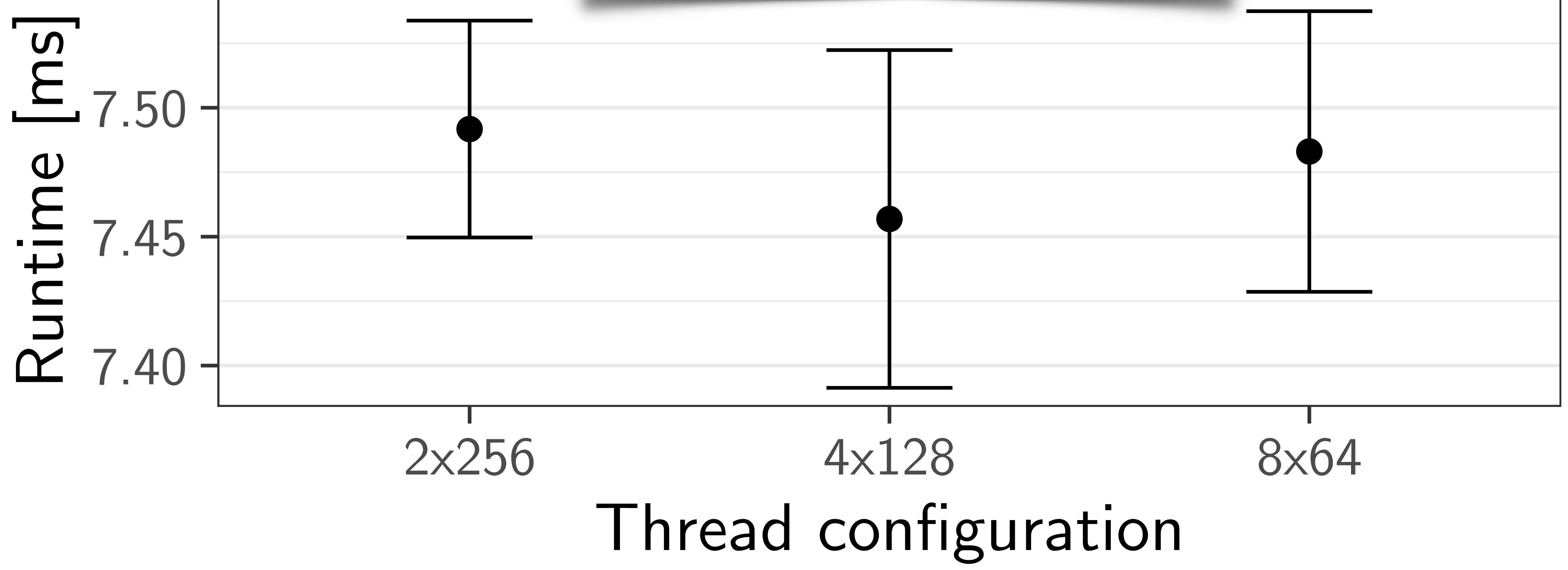
- Search space: Tesla K40m, SHARED aggregation, ca. 8 million groups.
- Search space appears convex but has multiple local minima.

Performance plateaus



	1	2	4	8	16	32	64	128	256	512	1024
1024	1.42	1.51	1.50	1.51	1.50	1.51	1.49	1.48	1.49	1.46	1.49
512	1.10	1.42	1.51	1.51	1.51	1.51	1.51	1.50	1.49	1.49	1.48
256	1.14	1.10	1.42	1.51	1.52	1.52	1.51	1.51	1.51	1.50	1.51
128	1.59	1.14	1.09	1.42	1.52	1.52	1.52	1.51	1.50	1.50	1.50
64	3.08	1.63	1.14	1.10	1.42	1.42	1.41	1.41	1.41	1.40	1.40
32	6.04	3.13	1.63	1.14	1.10	1.10	1.10	1.09	1.10	1.10	1.09
16	8.71	4.52	2.34	1.30	1.01	1.01	1.01	1.00	1.00	1.01	1.00
8	13.54	6.97	3.64	1.90	1.14	1.14	1.13	1.13	1.13	1.12	1.12
4	21.18	10.68	5.54	2.88	1.54	1.53	1.50	1.50	1.50	1.50	1.49
2	33.65	16.86	8.62	4.43	2.31	2.28	2.25	2.25	2.24	2.24	2.22
1	54.43	27.12	13.69	7.00	3.60	3.56	3.54	3.53	3.52	3.51	3.48

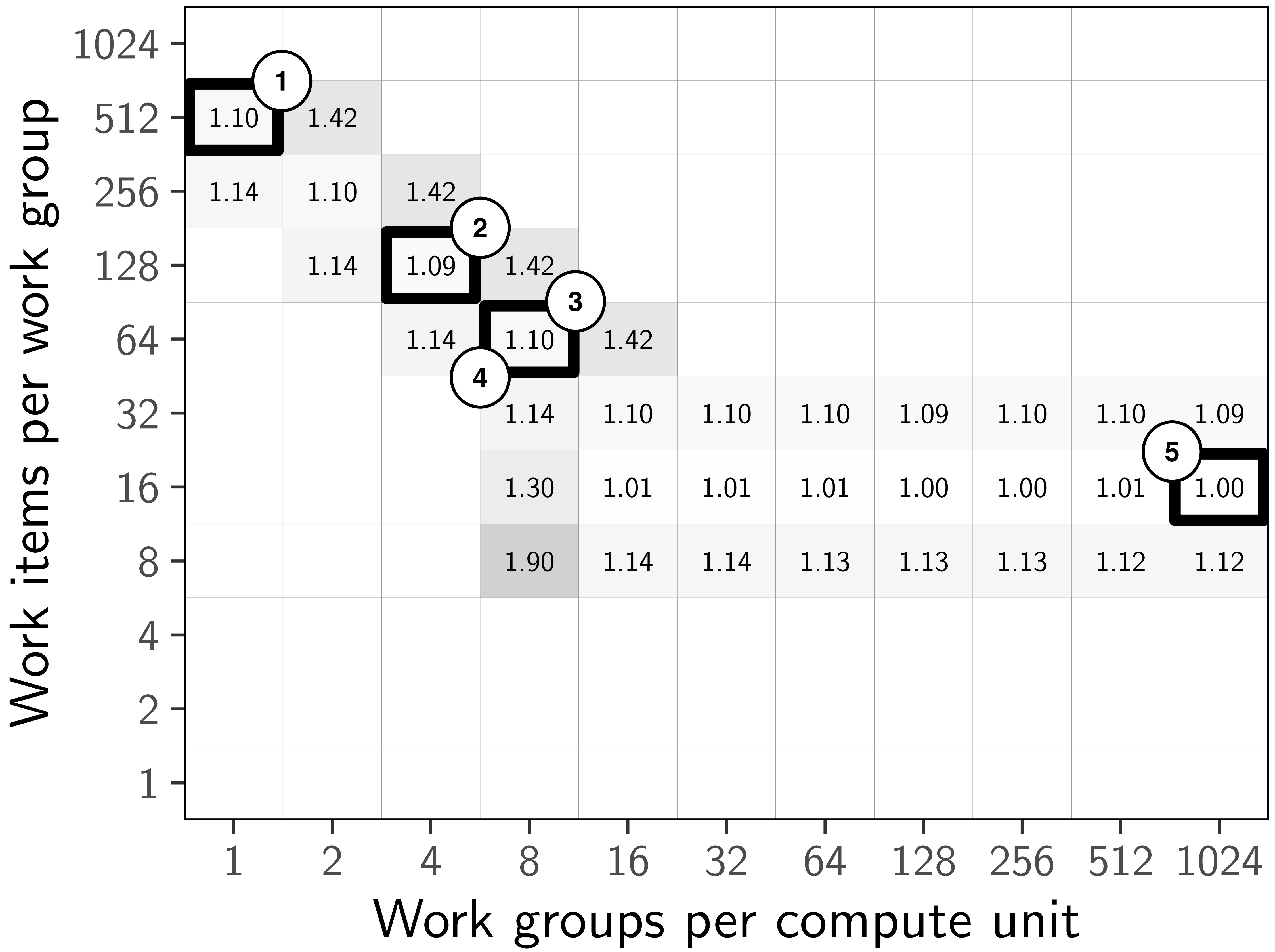
Absolute runtimes differ by less than 0.5%



- **Performance plateau**: Runtimes of two adjacent thread configurations differ by less than a small delta.
- **Nearly convex**: Single local minimum if we account for runtime variation.

Thread configuration search spaces are nearly convex.

Finding fast thread configurations



	1	2	4	8	16	32	64	128	256	512	1024
1024											
512	1.10	1.42									
256	1.14	1.10	1.42								
128		1.14	1.09	1.42							
64			1.14	1.10	1.42						
32				1.14	1.10	1.10	1.10	1.09	1.10	1.10	1.09
16				1.30	1.01	1.01	1.01	1.00	1.00	1.01	1.00
8				1.90	1.14	1.14	1.13	1.13	1.13	1.12	1.12
4											
2											
1											

① Start with initial thread configuration.

② Follow gradient in search space to local minimum.

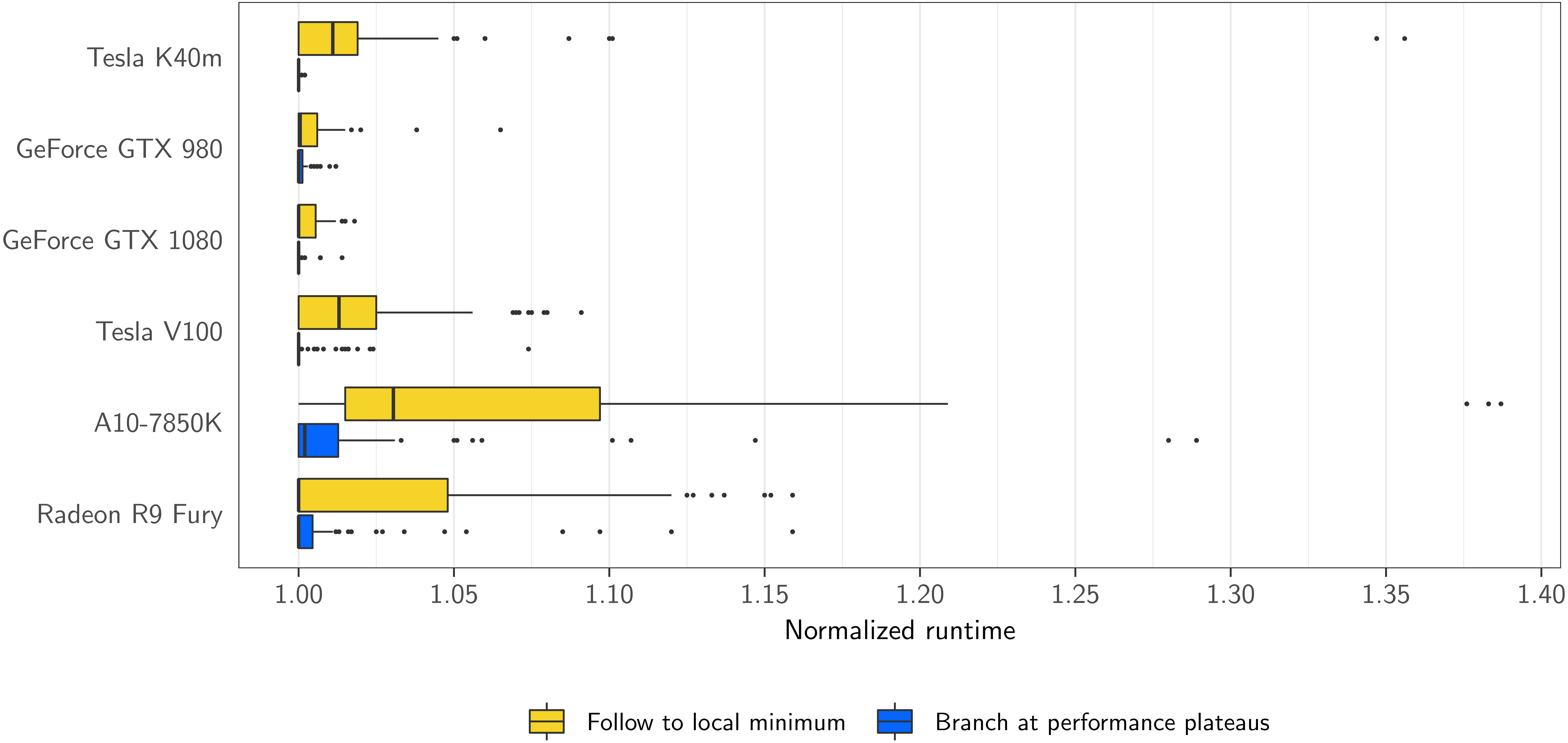
③ Branch search path at performance plateaus.

④ Prune branches that are slower than the fastest thread configuration found so far.

⑤ Stop at minimum when there are no more branches.

Approach: Follow gradient and branch search path at performance plateaus.

Runtime of found implementation



Summary

1. INDEPENDENT aggregation is not competitive on newer GPUs that implement fast atomics on local memory. Use WORKGROUPLOCAL aggregation instead.
2. The optimal thread configuration is highly dependent on the executing GPU.

Heuristics derived from analyzing a single GPU are not generalizable to other GPUs.

3. Thread configuration search spaces are *nearly convex*, i.e., they have a single local minimum when we account for runtime variation.

Follow gradient and branch at performance plateaus to find fast thread configurations.

4. NVIDIA GPUs exhibit a low degree of runtime variation. AMD GPUs exhibit a high degree of runtime variation.
5. Aggregation performance is limited by global GPU memory latency (and not transfer bandwidth) when the hash table exceeds the L2 cache.