Optimized LU-decomposition with Full Pivot for Small Batched Matrices

S3069

Ian Wainwright

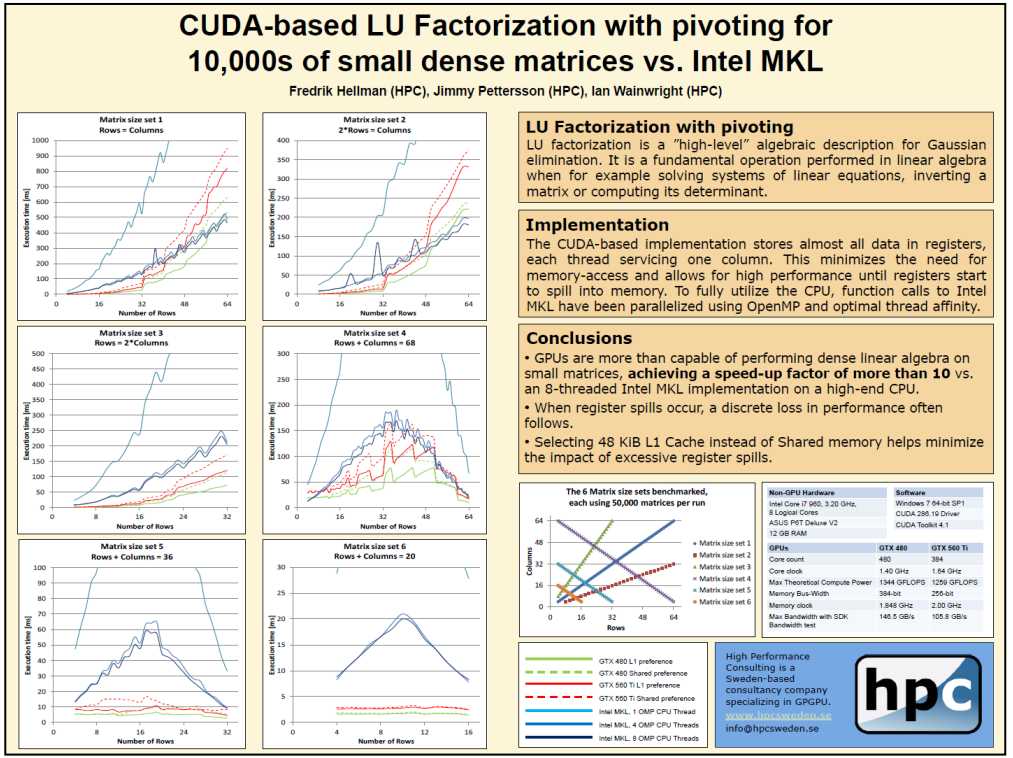
High Performance Consulting Sweden

ian.wainwright@hpcsweden.se

Background

Based on work for GTC 2012: 10x speed-up vs multi-threaded Intel MKL on an 4 core (8HT).

http://www.hpcsweden.se/files/CUDA-based_LU_Factorization.pdf



Aim

To investigate various techniques for batched matrix computations.

The motivation behind this work is to use our previous work on LU-decomposition as a use case to investigate optimization techniques for Kepler.

Outline

1. LU-decomposition
2. Implementations
3. Performance gains
4. Conclusions

LU-decomposition

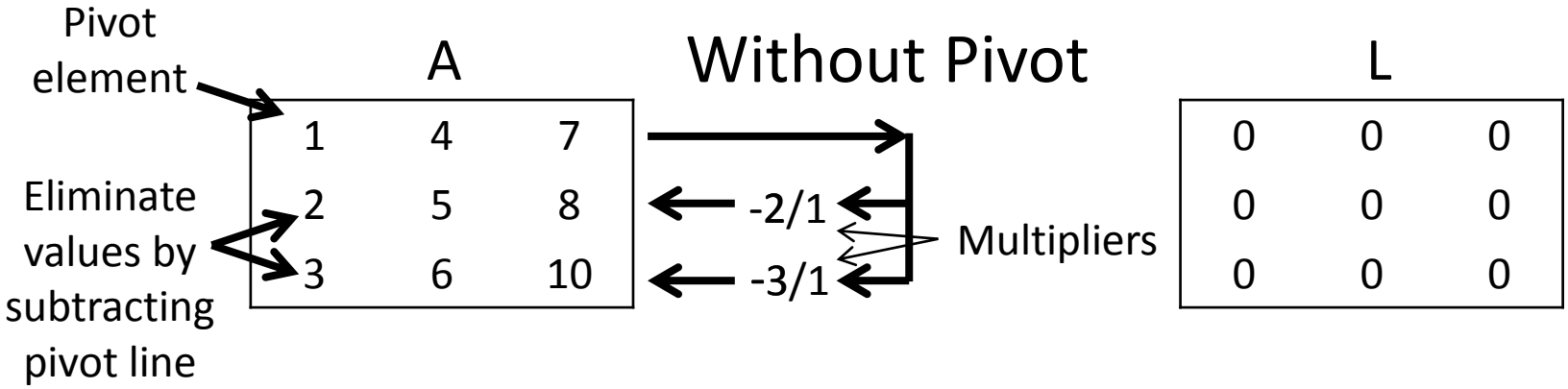
1. LU-decomposition
2. Implementations
3. Performance gains
4. Conclusions

LU-decomposition

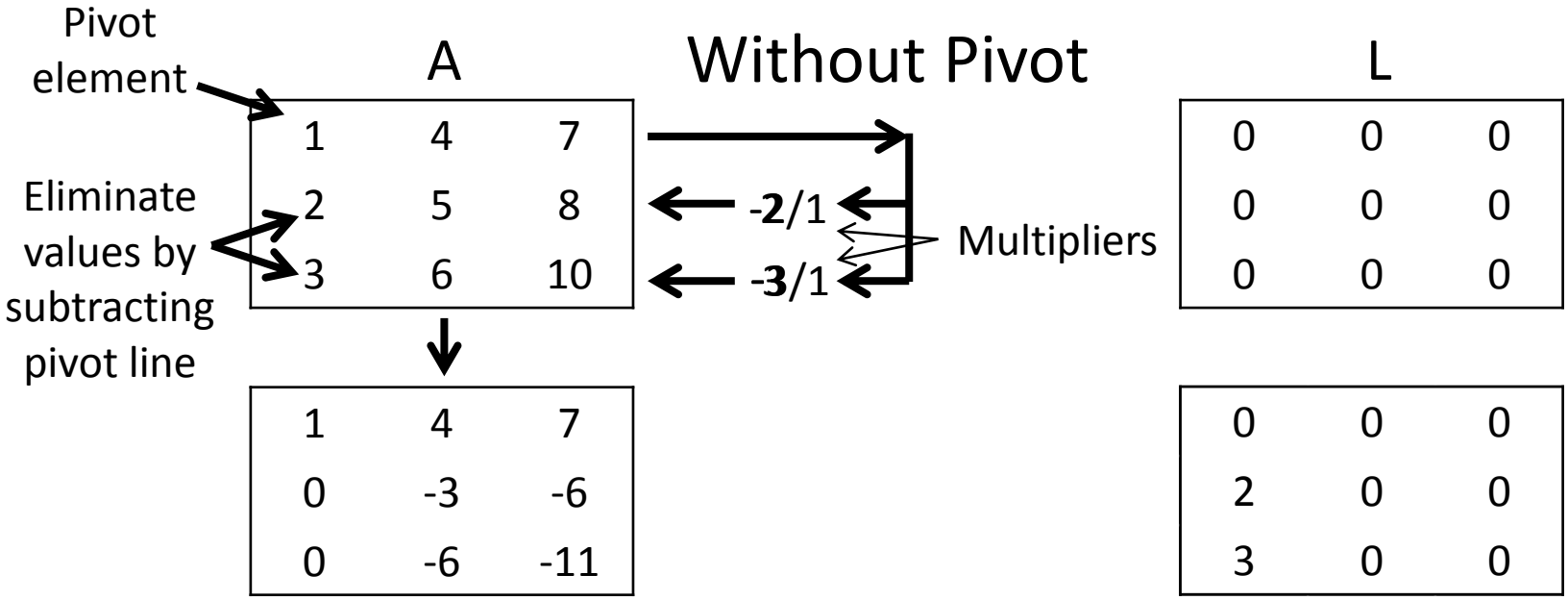
The idea is to transform A where `Ax=b` to an equivalent triangular system such that `A=LU`

$$
\underset{A}{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}} = \underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}} * \underset{U}{\begin{bmatrix} u11 & u12 & u13 & u14 \\ 0 & u22 & u23 & u24 \\ 0 & 0 & u33 & u34 \\ 0 & 0 & 0 & u44 \end{bmatrix}}
$$

LU-decomposition



LU-decomposition



LU-decomposition

Without Pivot

A

1	4	7
2	5	8
3	6	10

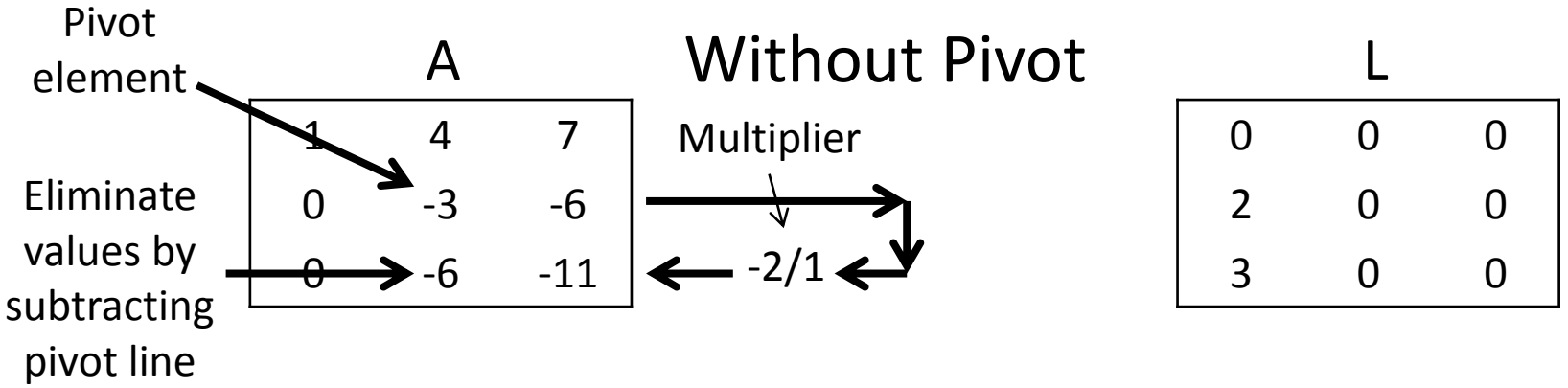
L

0	0	0
0	0	0
0	0	0

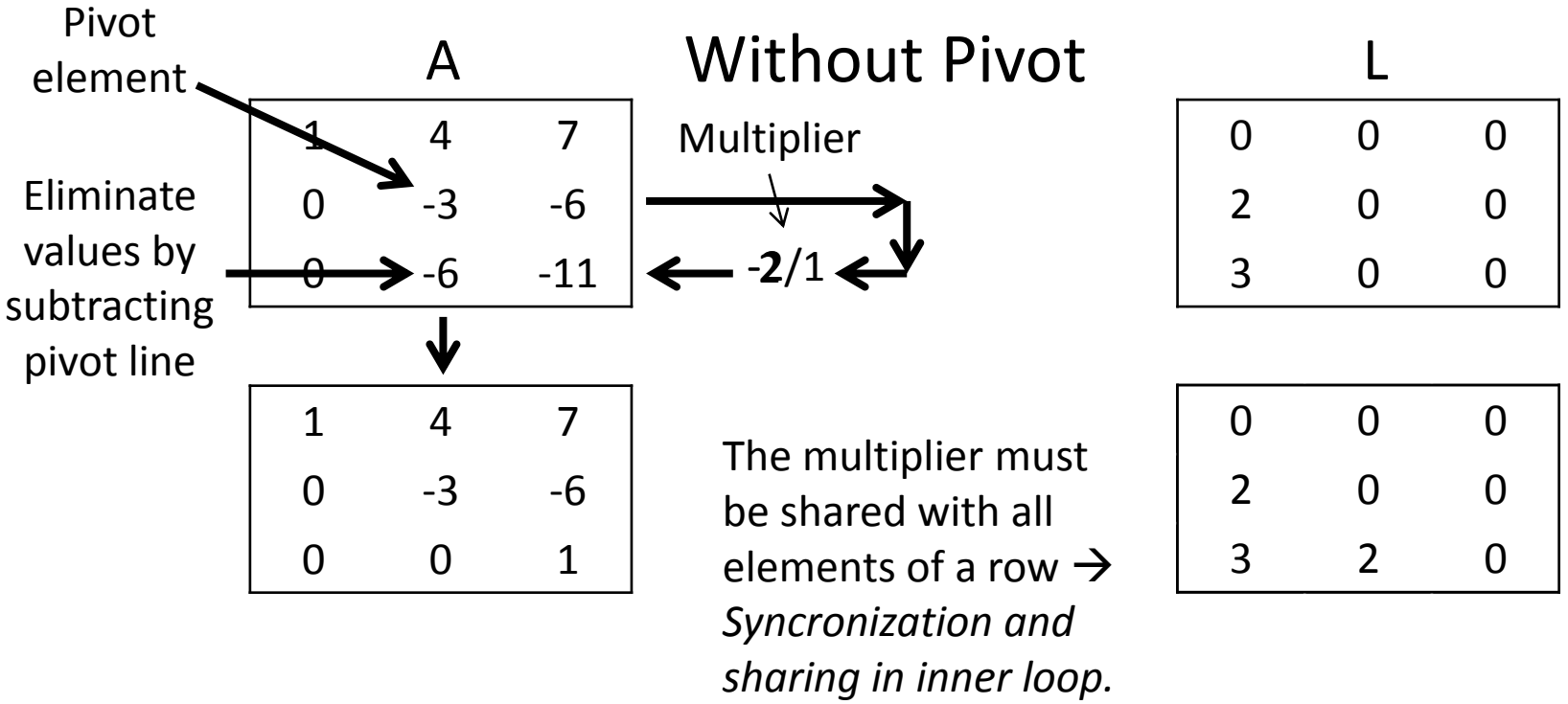
1	4	7
0	-3	-6
0	-6	-11

0	0	0
2	0	0
3	0	0

LU-decomposition



LU-decomposition



LU-decomposition

Without Pivot

A

1	4	7
0	-3	-6
0	-6	-11

L

0	0	0
2	0	0
3	0	0

1	4	7
0	-3	-6
0	0	1

0	0	0
2	0	0
3	2	0

LU-decomposition

Without Pivot

A

1	4	7
0	-3	-6
0	0	1

L

0	0	0
2	0	0
3	2	0

LU-decomposition

Without Pivot

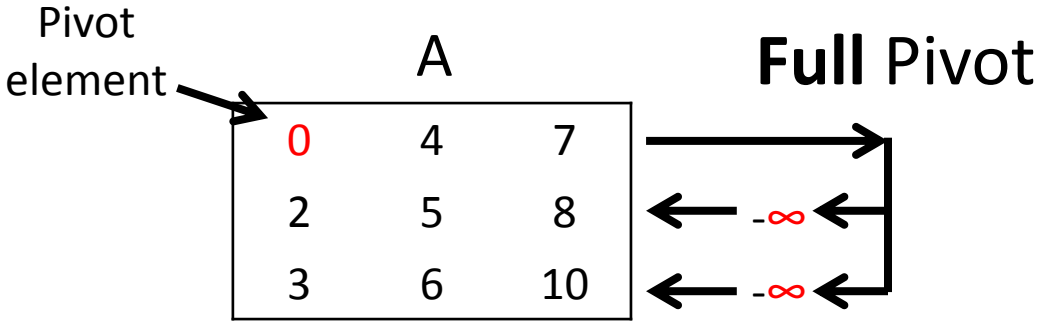
U

1	4	7
0	-3	-6
0	0	1

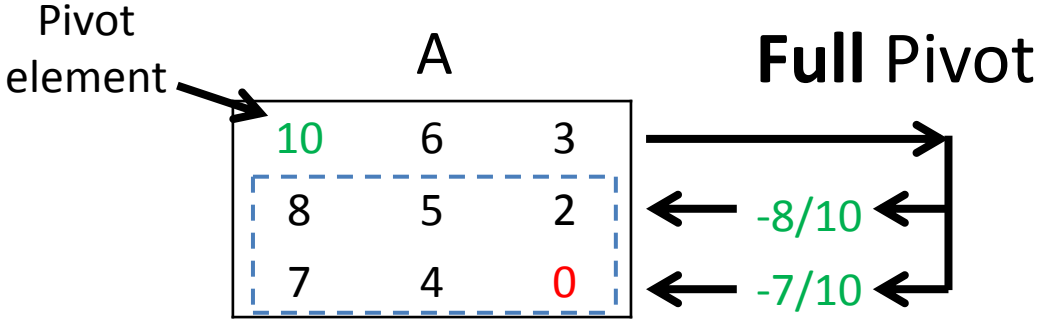
L

1	0	0
2	1	0
3	2	1

LU-decomposition



LU-decomposition



Solution: Perform Pivot

1. Find *largest* value in bottom submatrix.
2. Swap rows and columns to make largest value the pivot element.
3. Keep track of row and column pivot in each step.
4. Then perform operations over rows as usual.

LU-decomposition with full pivot is stable

LU-decomposition

Pivot element

A

Full Pivot

Find largest value→
Synchronization and **data-sharing**

Necessitates max-reduction in each column iteration.

Solution: Perform Pivot

1. **Find *largest* value in bottom submatrix**.
2. Swap rows and columns to make largest value the pivot element.
3. Keep track of row and column pivot in each step.
4. Then perform operations over rows as usual.

LU-decomposition with full pivot is stable

Implementations

Problem size:

- Matrices of at most 32 rows or columns of any shape, i.e. both rectangular and square.
- Batches of 10 000 matrices.
- Find **L** and **U** for matrix **A** such that **PAQ=LU** with full pivoting, where **P** and **Q** are the pivot vectors.

Implementations

Mapping of problem:

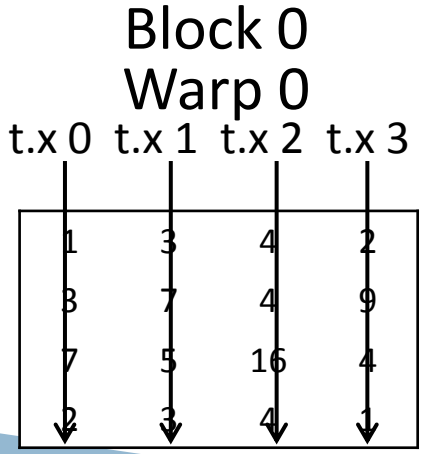
- Map one matrix to a warp.
- Map a column to a thread.
- One or more matrices per block.

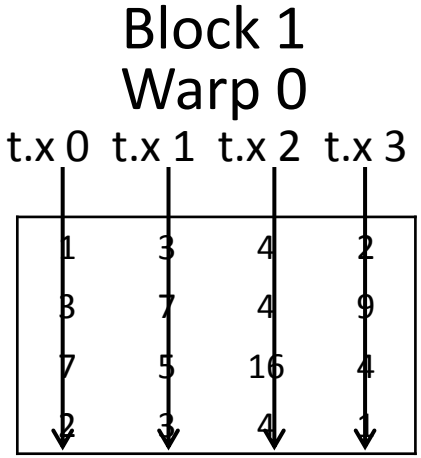
Implementations

Mapping of problem:

- Map one matrix to a warp.
- Map a column to a thread.
- **One** or more matrices per block.

One matrix per block





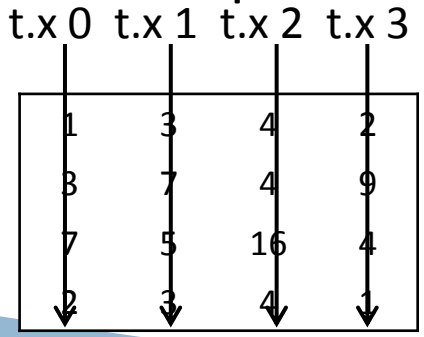
Implementations

Mapping of problem:

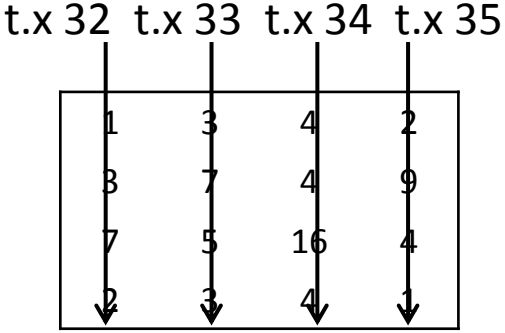
- Map one matrix to a warp.
- Map a column to a thread.
- One or **more** matrices per block.

Two matrix per block

Block 0
Warp 0



Block 0
Warp 1



Implementations

Mapping of problem:

- **Map one matrix to a warp**.
- Map a colu
- One or **mor**

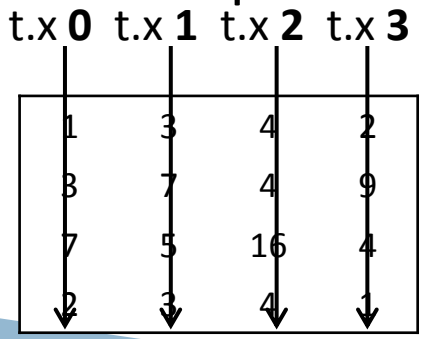
Two matrix per block

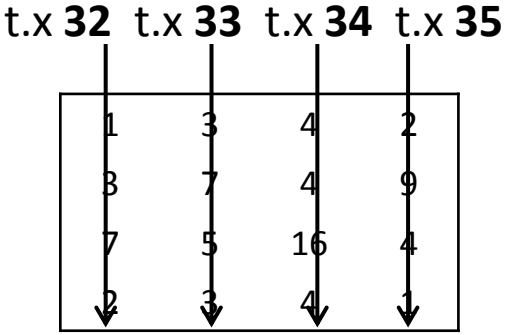
The number of matrices per block affects the occupancy.

Block 0

Warp 0

Warp 1





Implementations

Aim

To investigate various techniques for batched matrix computations.

Multiple matrices per block:

1. 1 matrix per block.
2. 2 matrices per block.
3. 4 matrices per block.

Occupancy

Synchronization and data sharing:

1. Shared-mem with __syncthreads().
2. Shared-mem using warp-synchronous programming.
3. Warp shuffle.

Synchronization

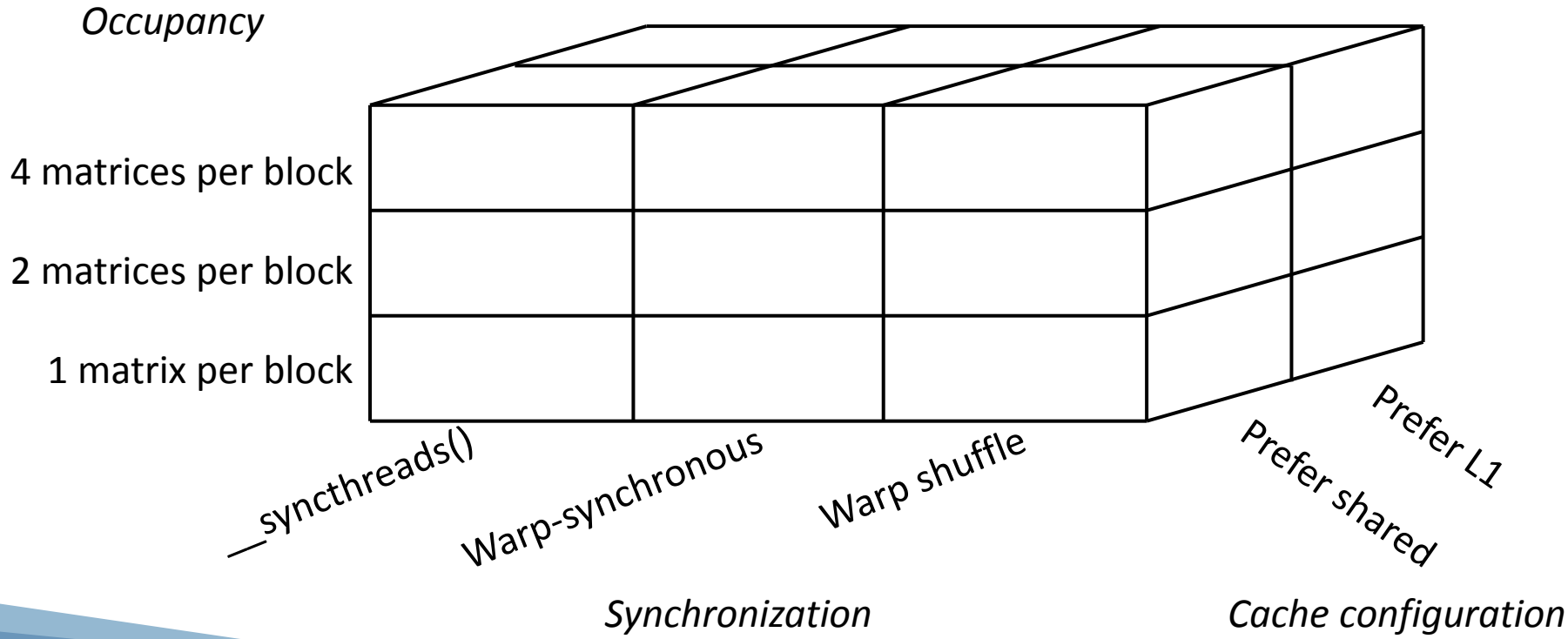
Cache-config:

1. Prefer shared.
2. Prefer L1.

Cache configuration

and their interaction

Implementations



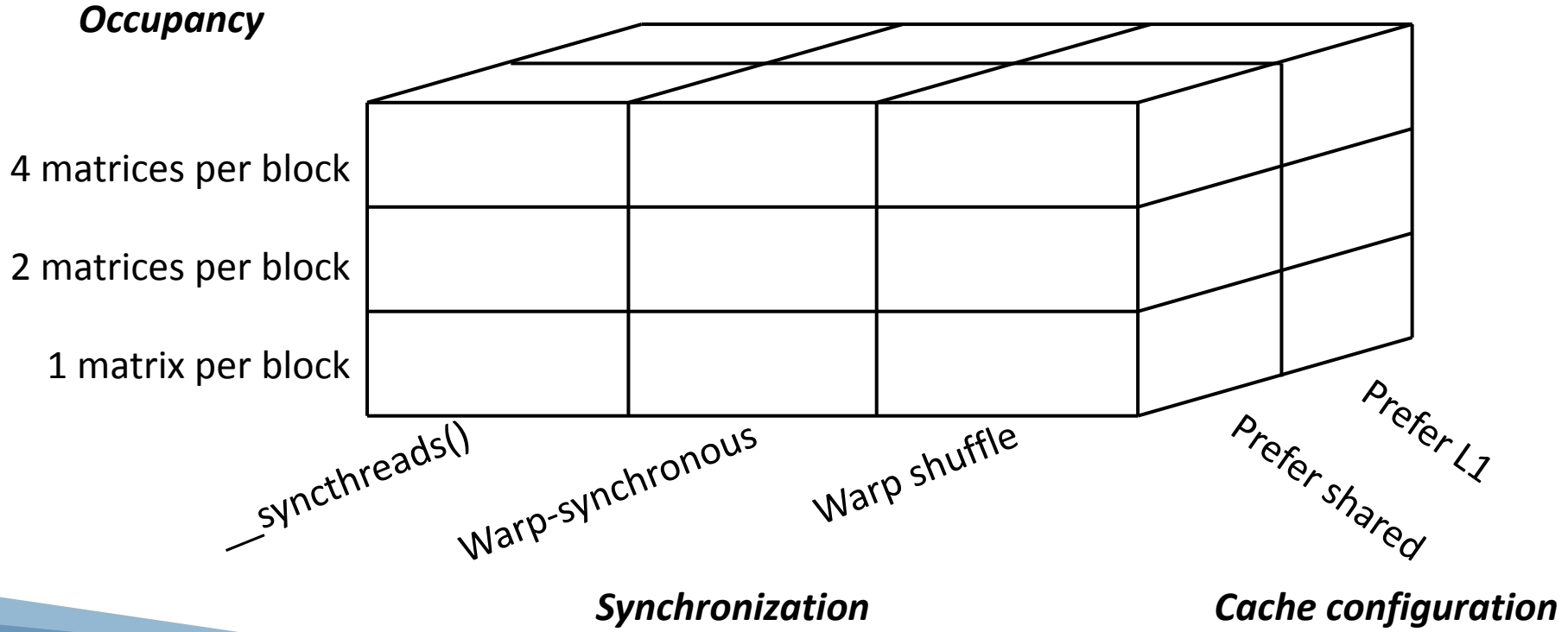
Performance gains

All benchmarks were performed on a standard GTX 680.

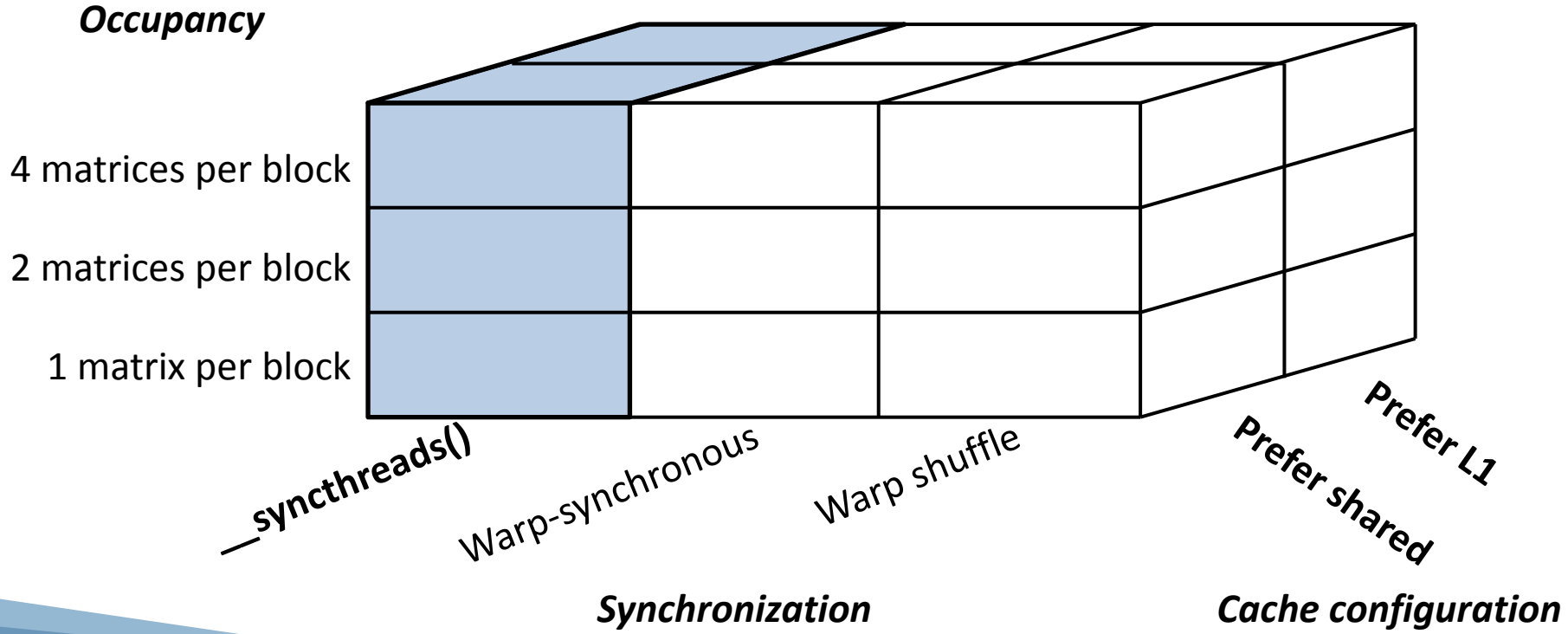
All numbers are based on execution time for a batch of 10 000 matrices for various sizes. Execution time is in ***ms***.

To save time, we will only be looking at performance numbers for square matrices.

Performance gains



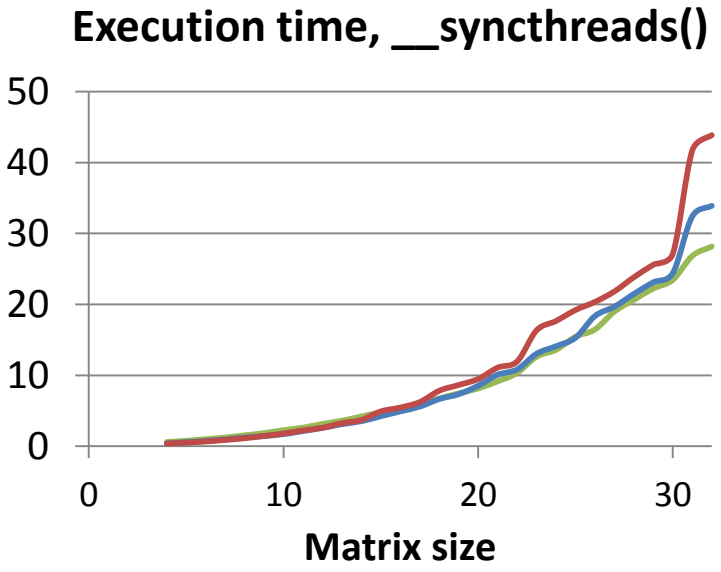
Performance gains



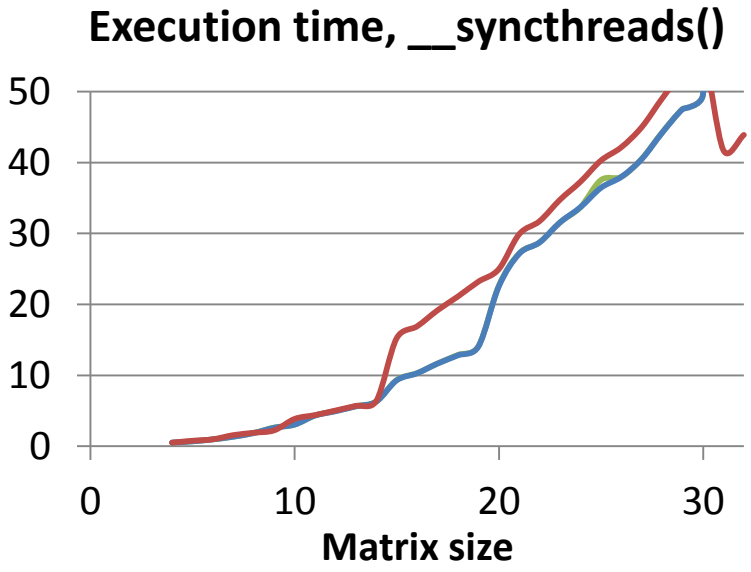
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. **Syncthreads()**
2. Warp-synchronous
3. Warp shuffle



Prefer shared

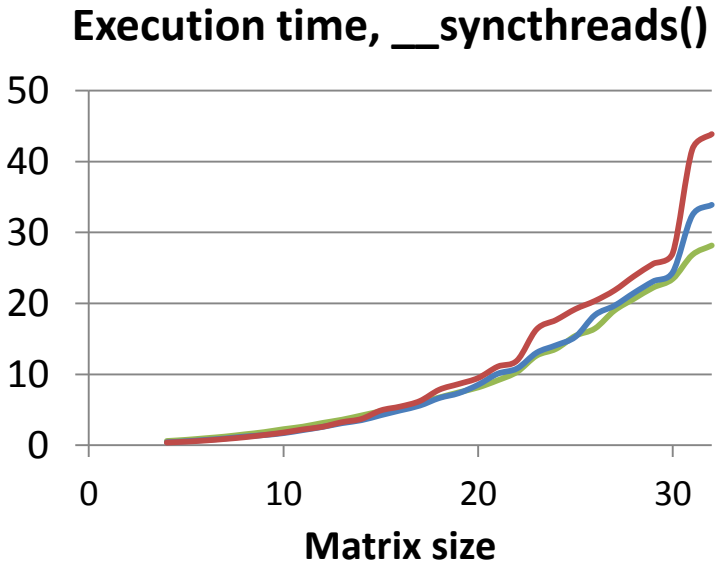


Prefer L1

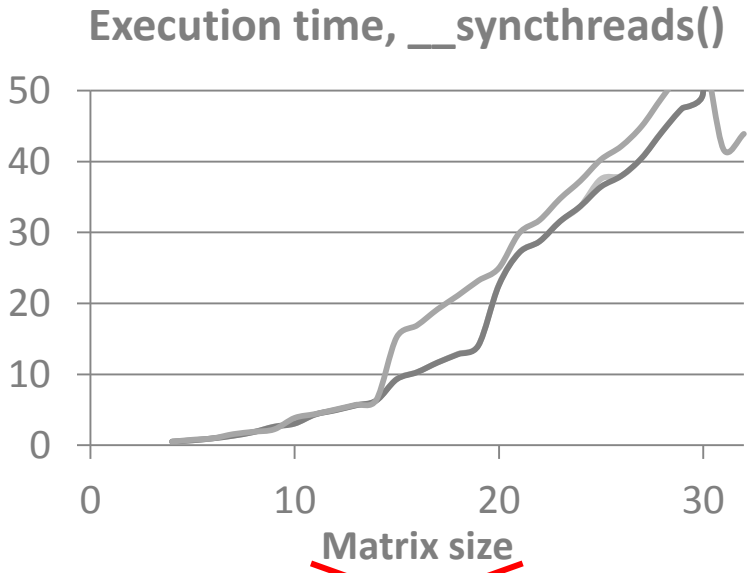
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. **Syncthreads()**
2. Warp-synchronous
3. Warp shuffle

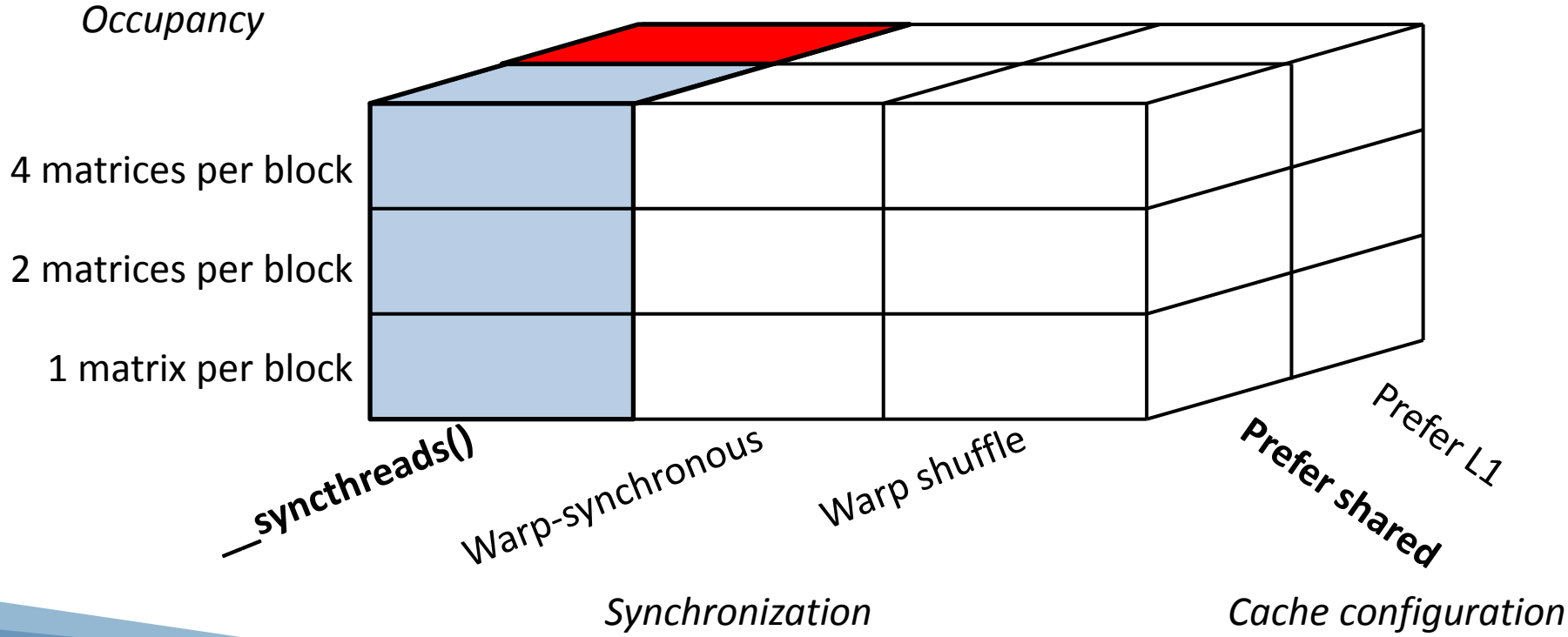


Prefer shared

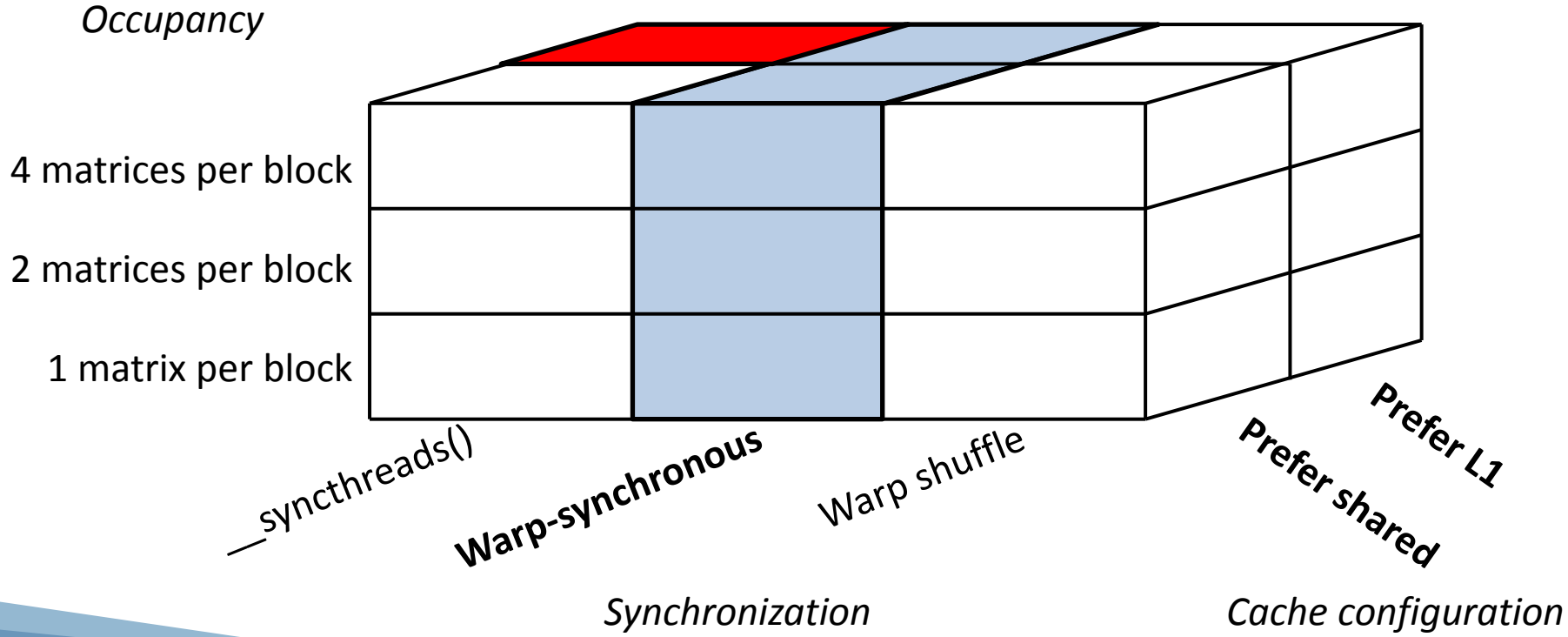


~~Prefer L1~~

Performance gains



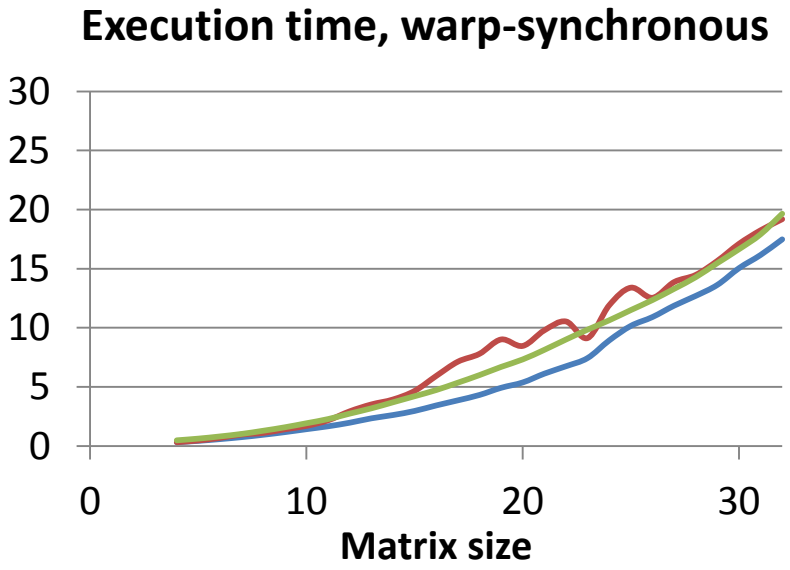
Performance gains



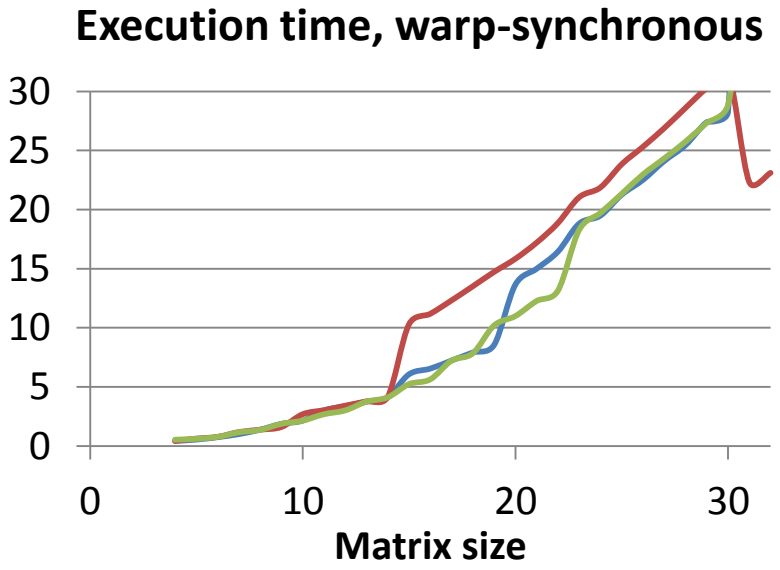
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. **Warp-synchronous**
3. Warp shuffle



Prefer shared

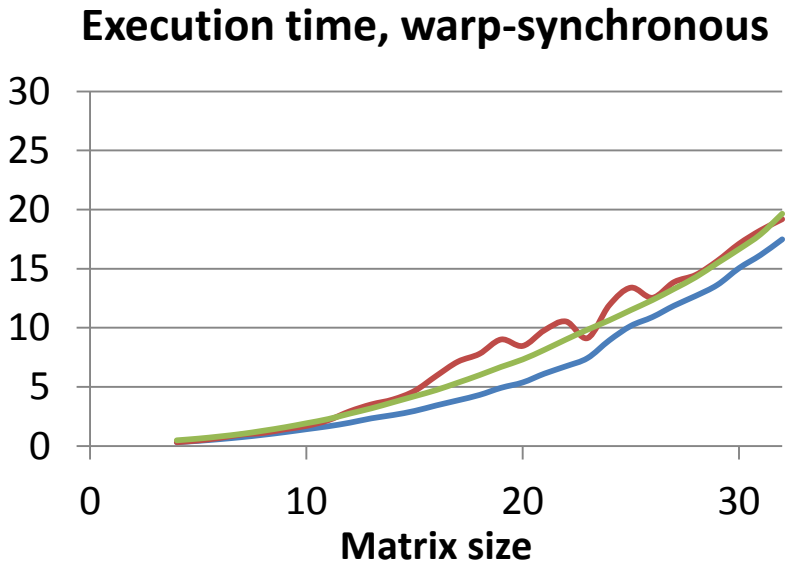


Prefer L1

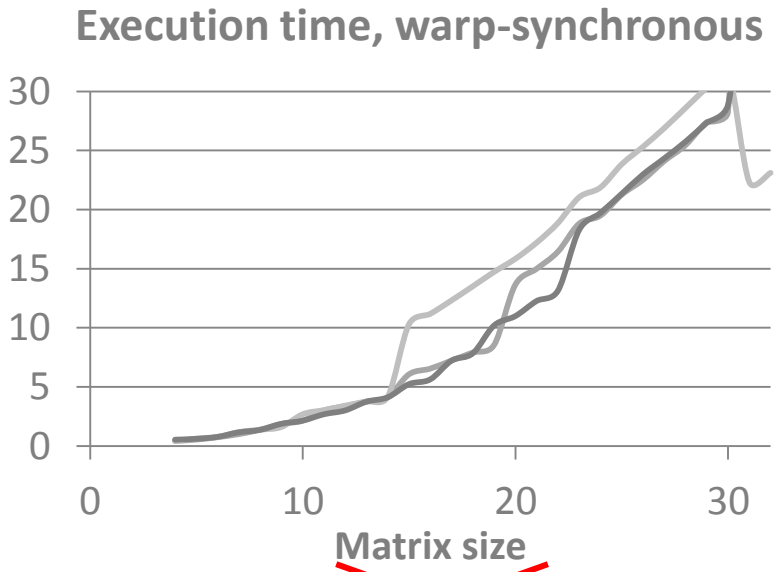
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. **Warp-synchronous**
3. Warp shuffle

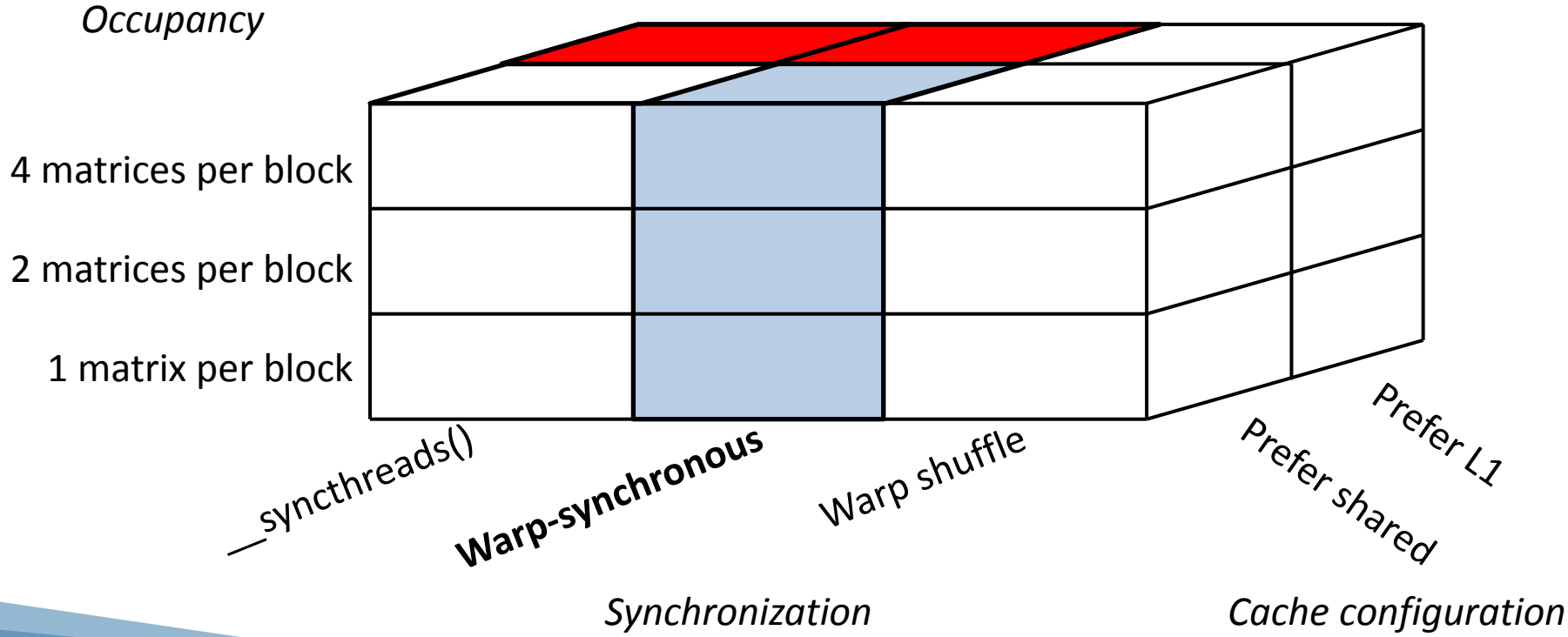


Prefer shared

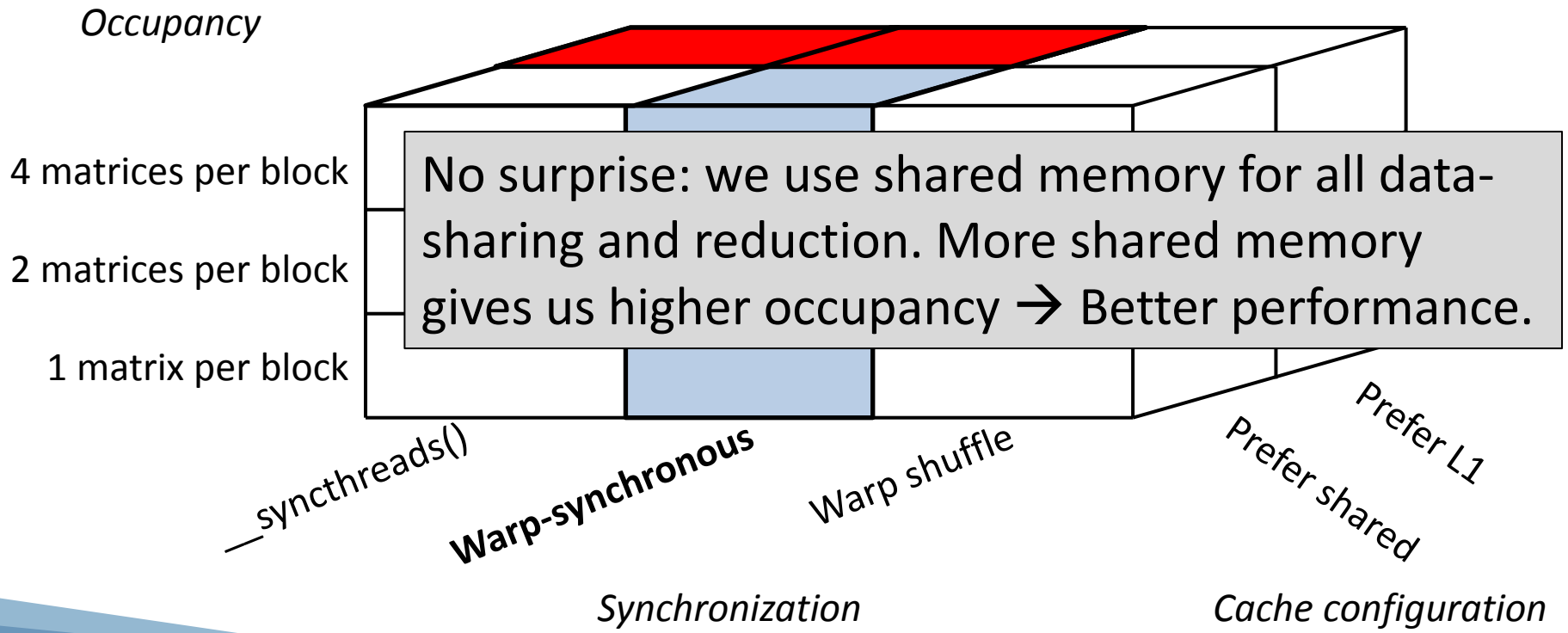


~~Prefer L1~~

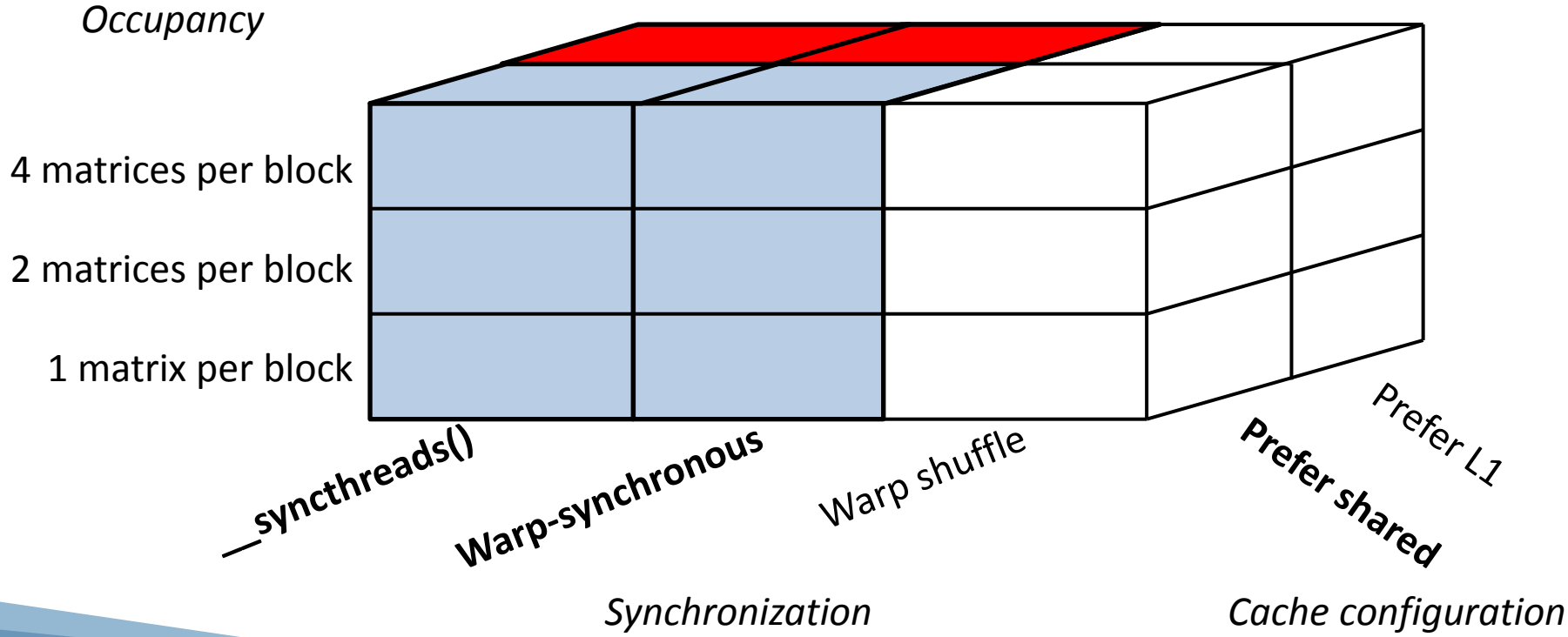
Performance gains



Performance gains



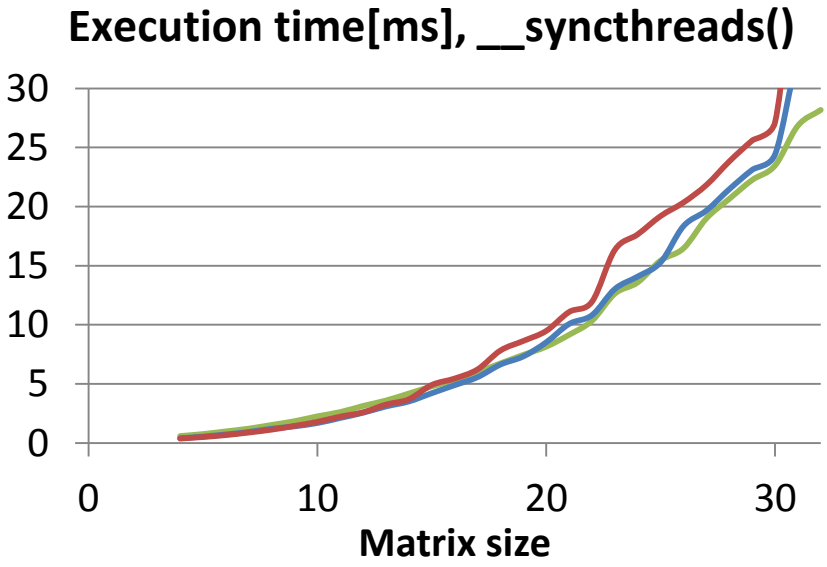
Performance gains

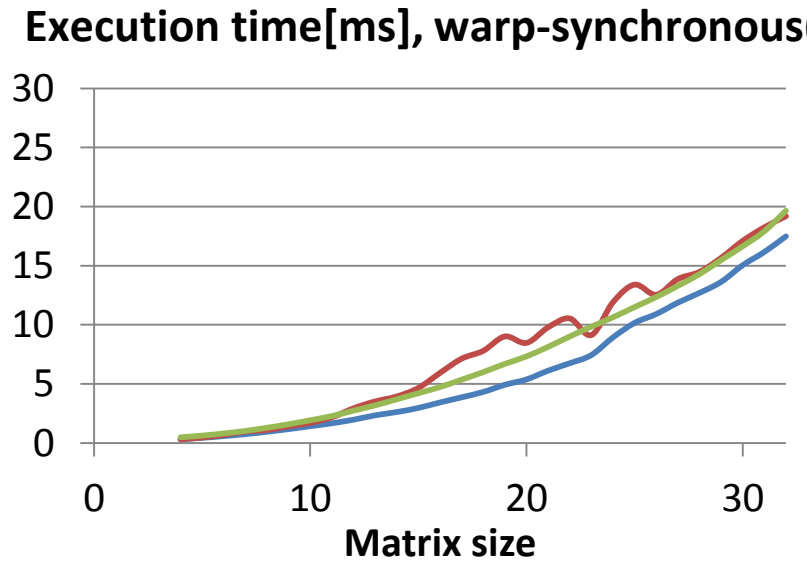


1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. Warp-synchronous
3. Warp shuffle





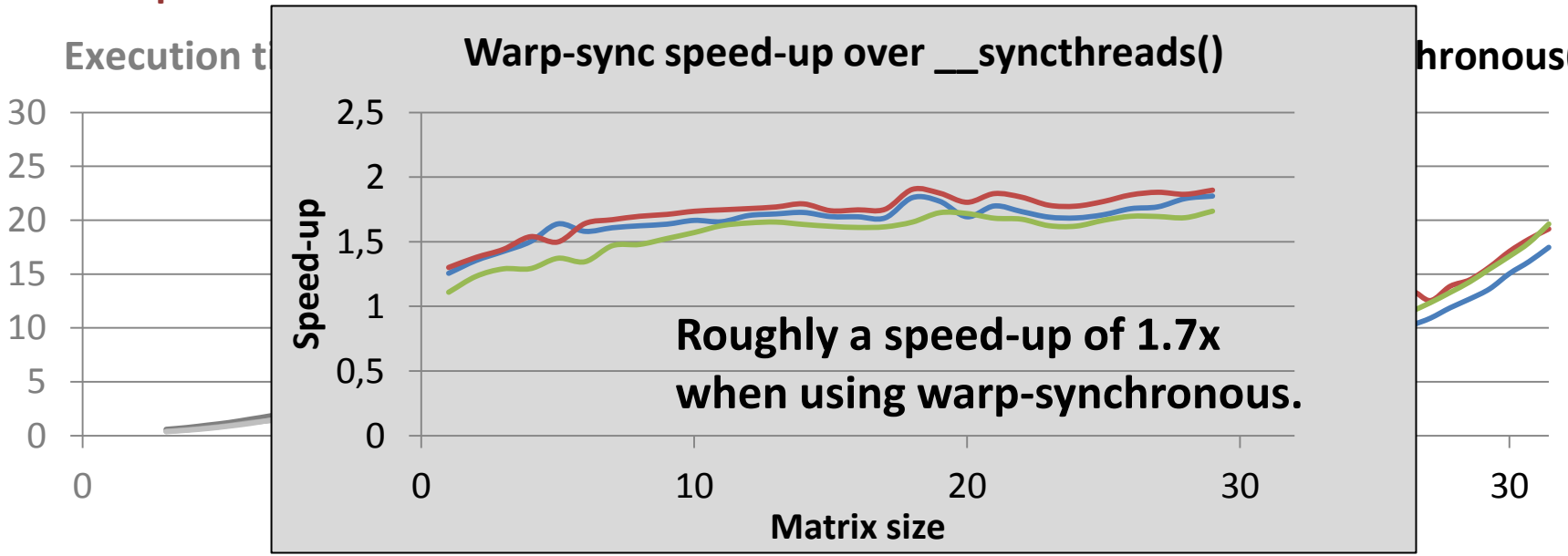
__syncthreads()

Warp-synchronous

1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

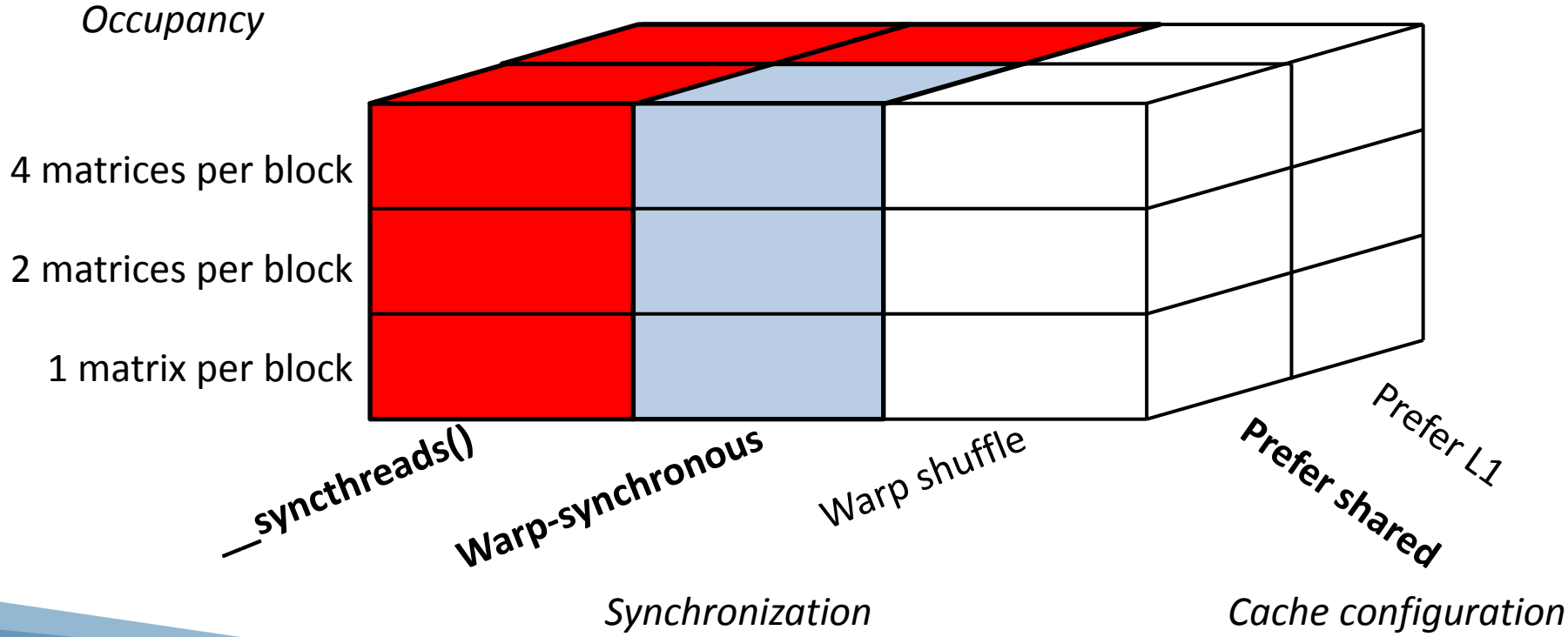
1. Syncthreads()
2. Warp-synchronous
3. Warp shuffle



~~__syncthreads()~~

Warp-synchronous

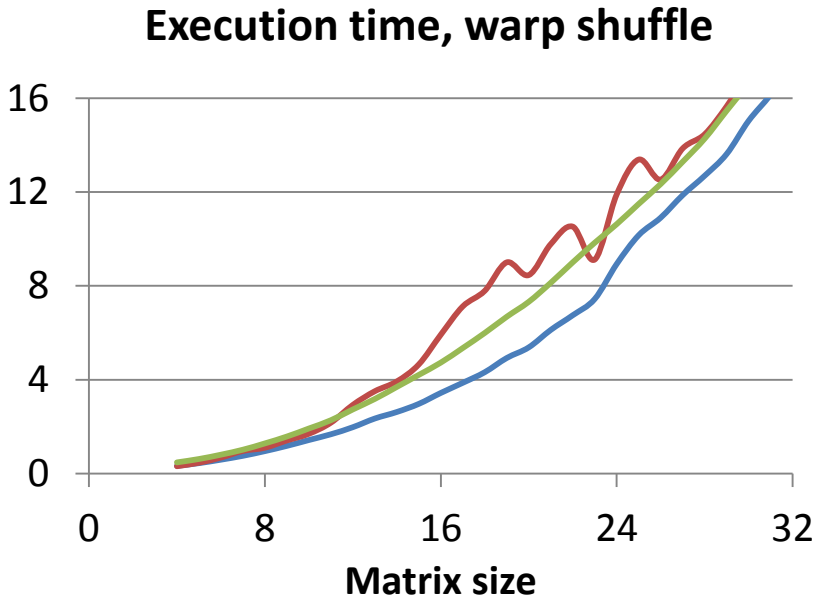
Performance gains



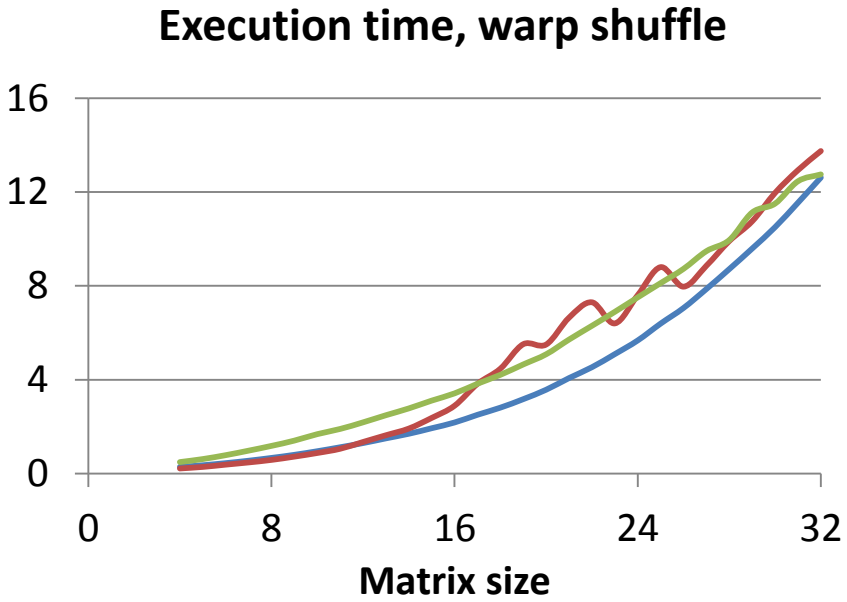
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. Warp-synchronous
3. **Warp shuffle**



Prefer shared



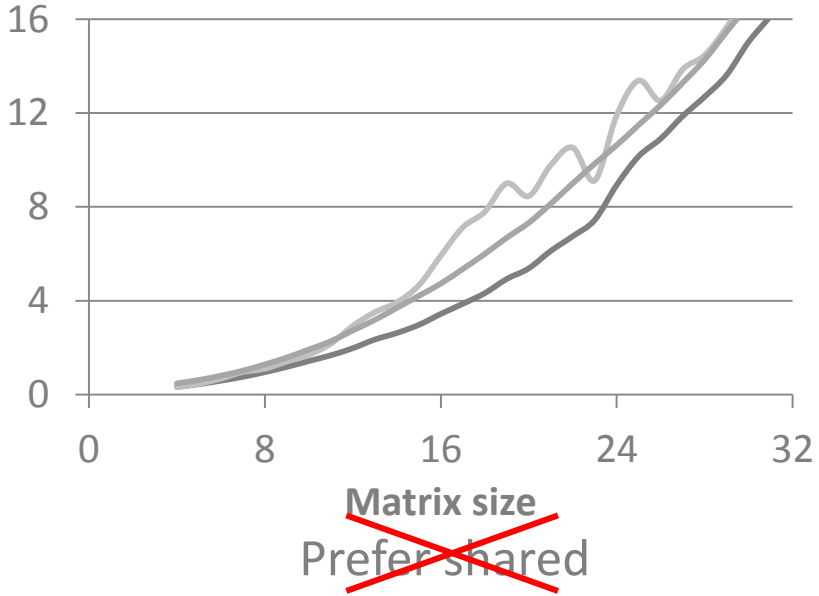
Prefer L1

1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

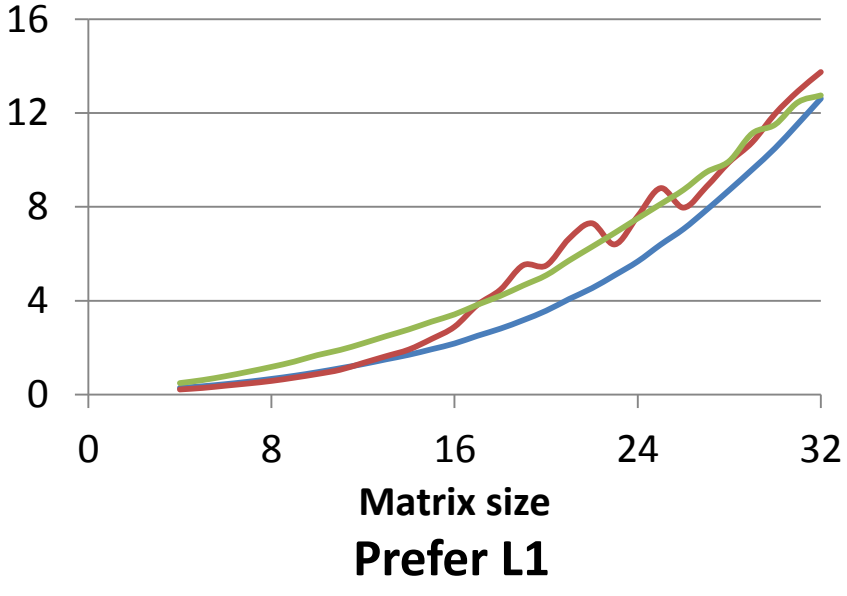
1. Syncthreads()
2. Warp-synchronous
3. **Warp shuffle**

Execution time, warp shuffle



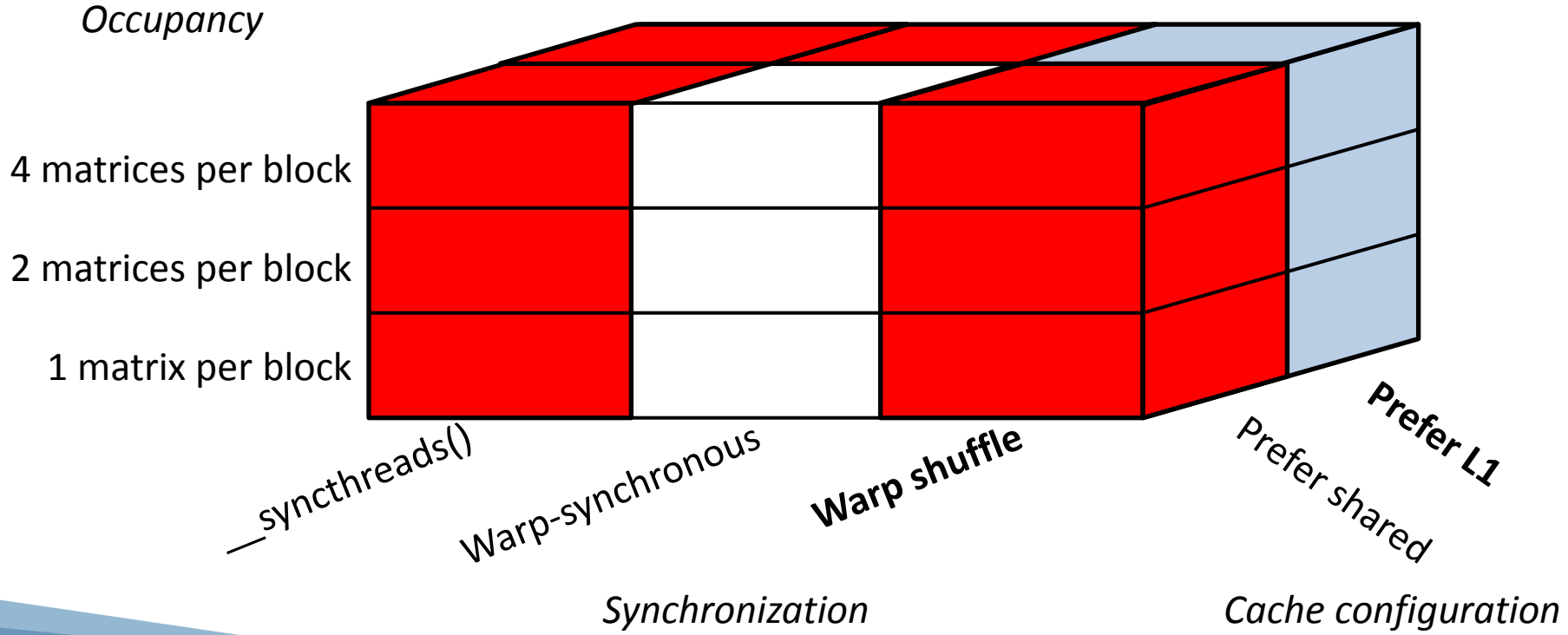
~~Prefer shared~~

Execution time, warp shuffle

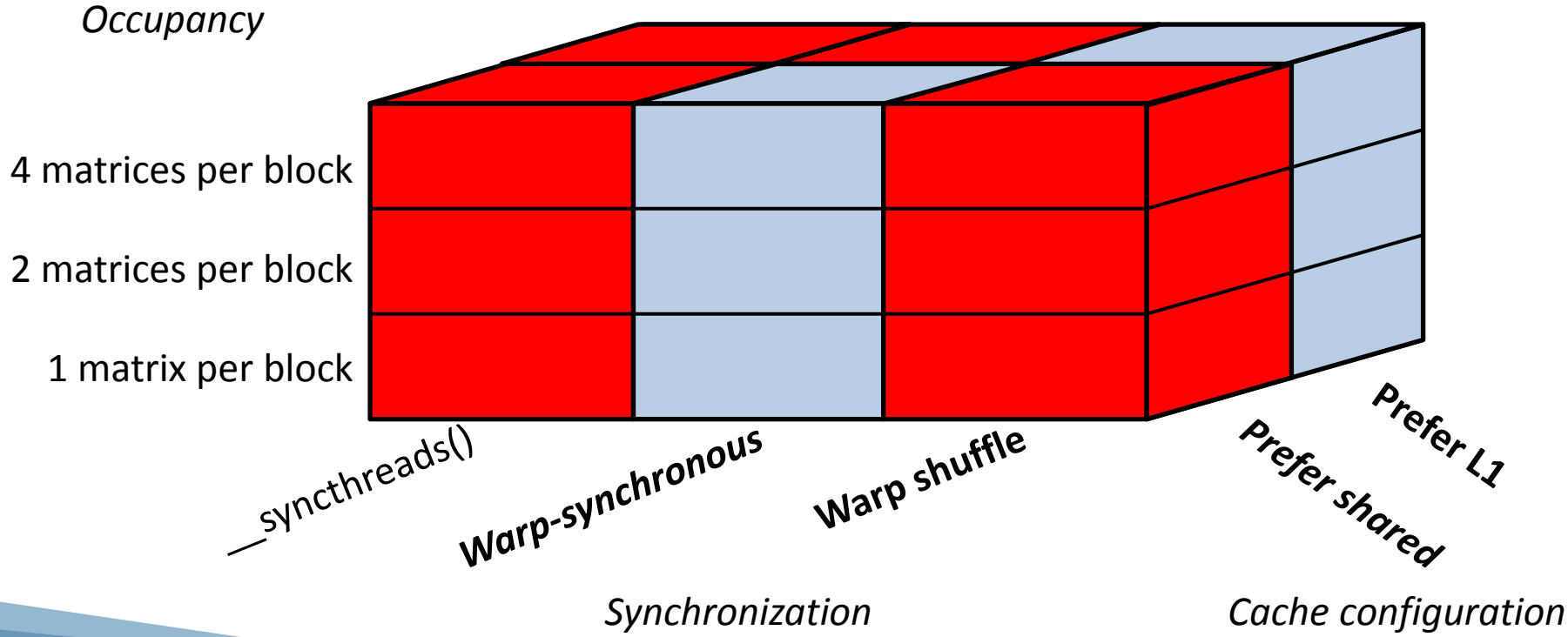


Prefer L1

Performance gains



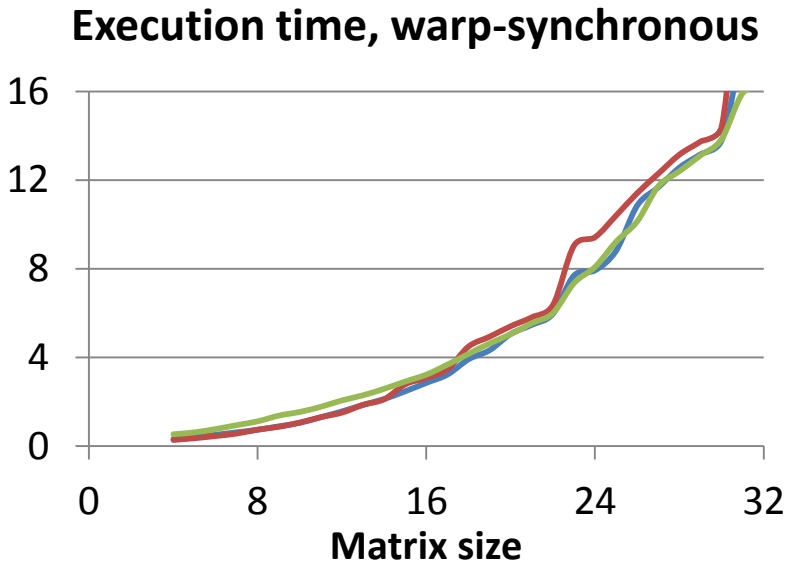
Performance gains



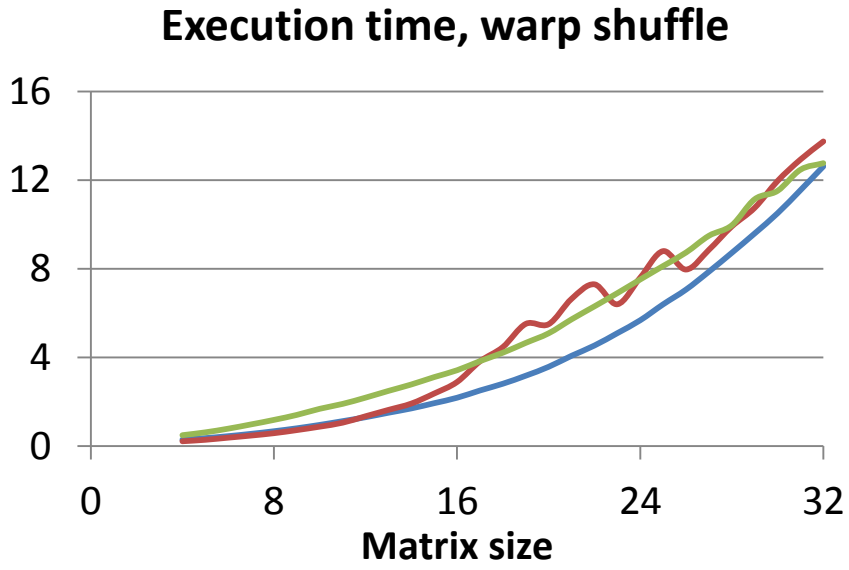
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. **Warp-synchronous**
3. **Warp shuffle**



Prefer shared

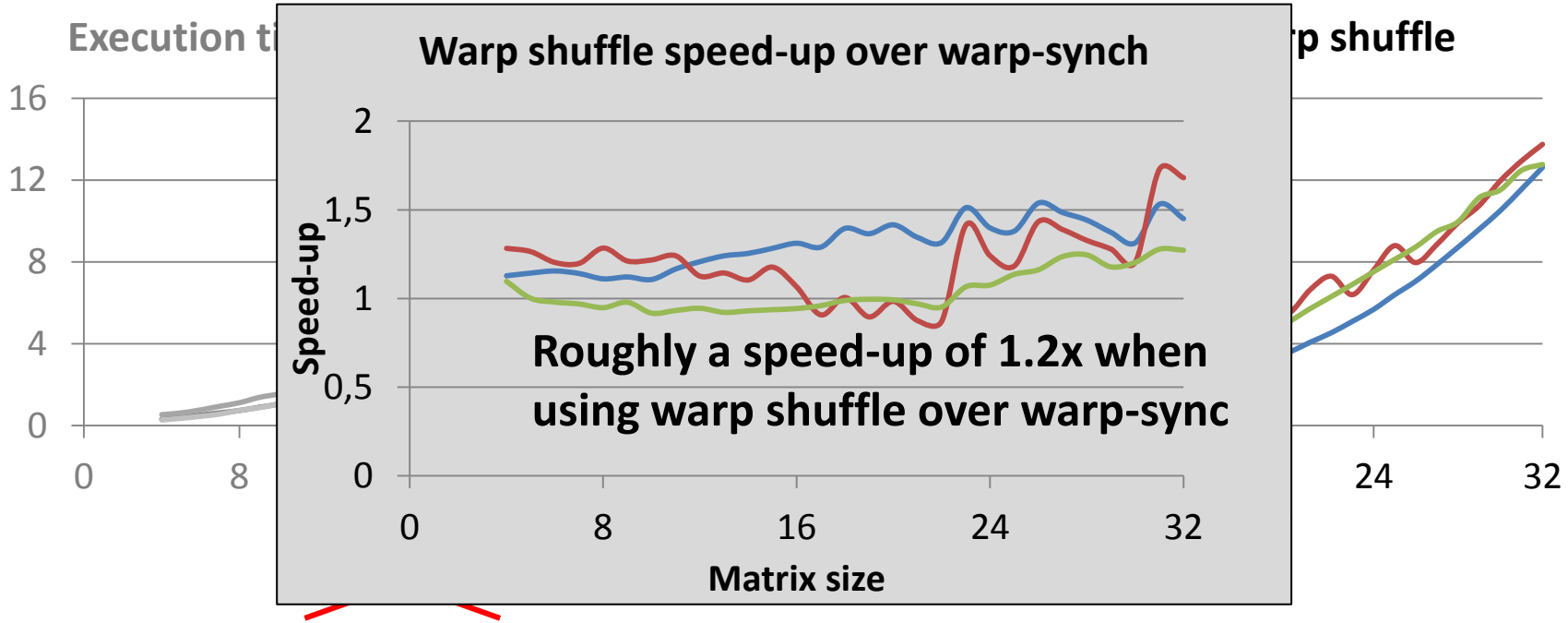


Prefer L1

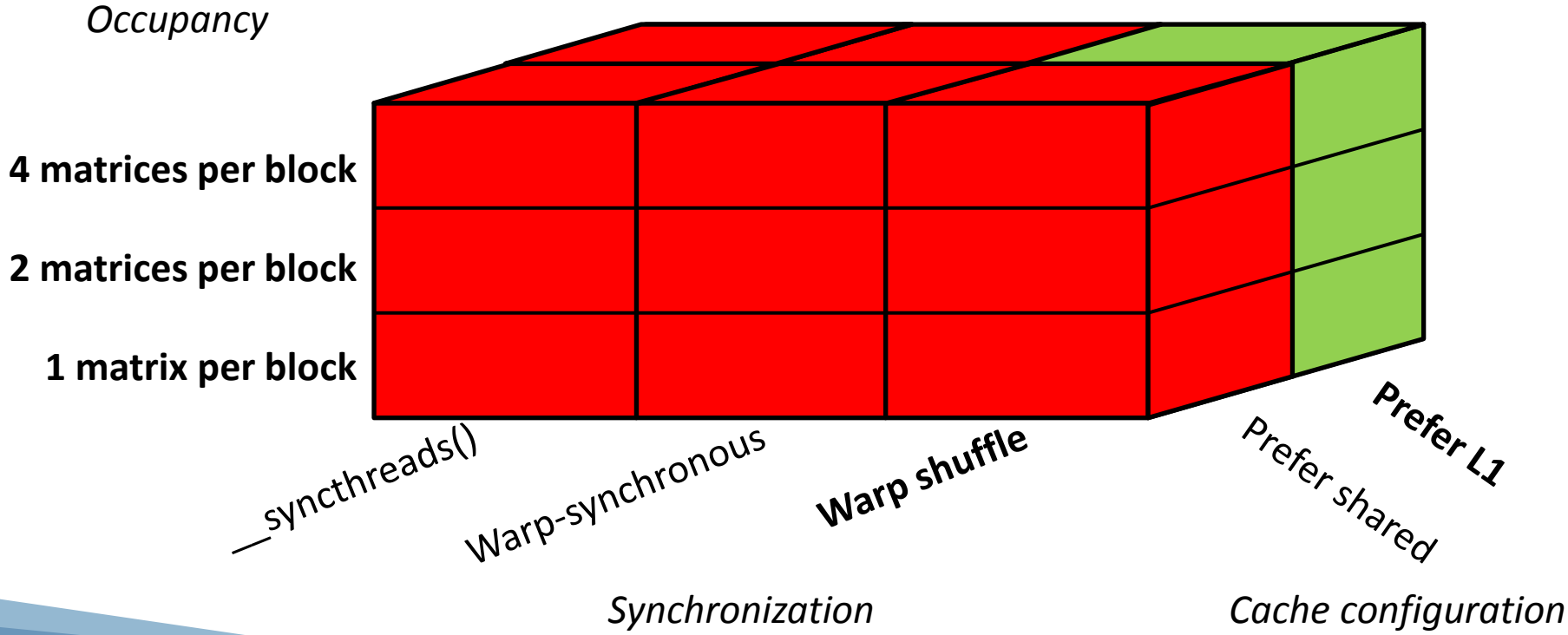
1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. **Warp-synchronous**
3. **Warp shuffle**



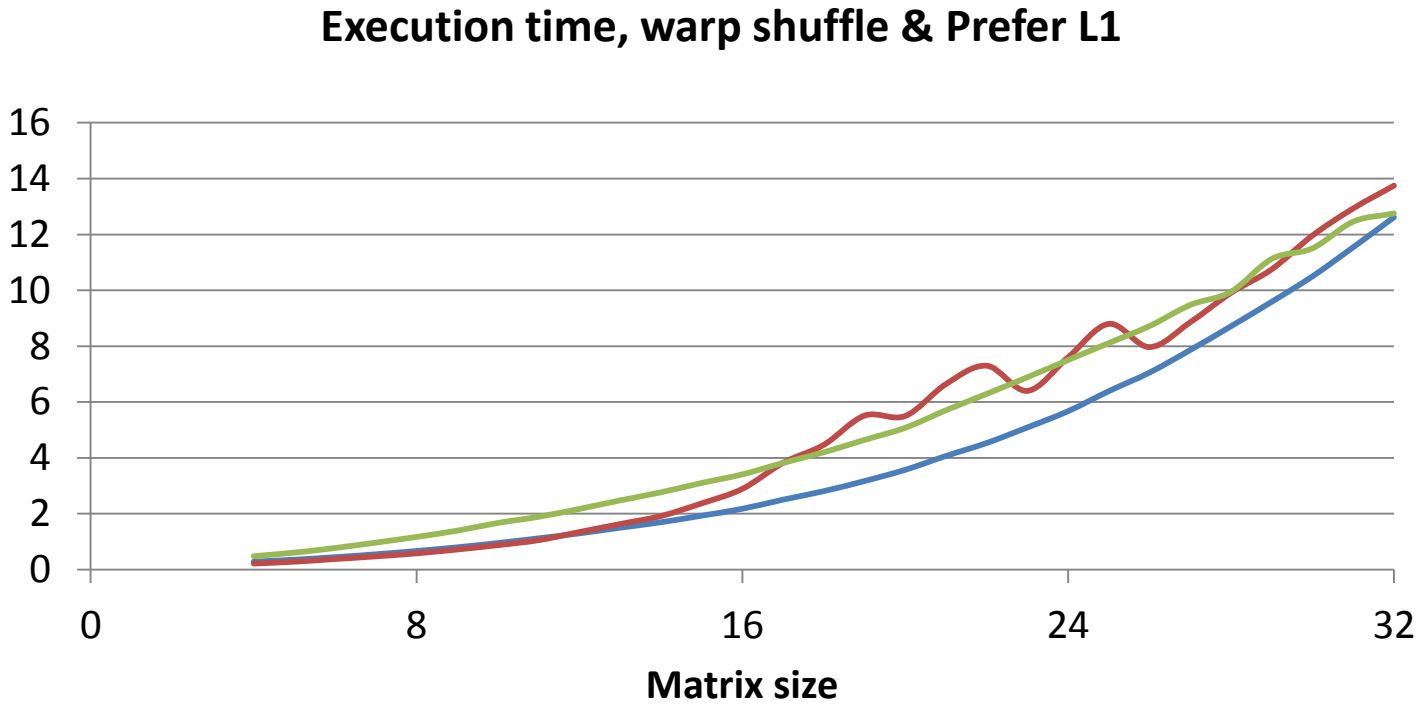
Performance gains



1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

1. Syncthreads()
2. **Warp-synchronous**
3. **Warp shuffle**



1 matrix per block
2 matrices per block
4 matrices per block

Performance gains

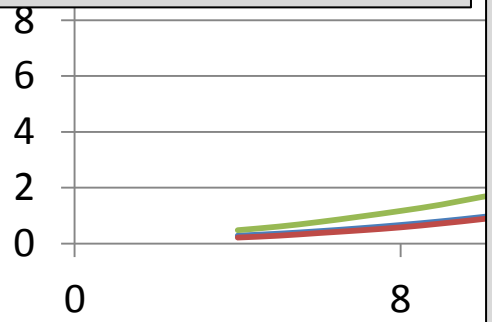
1. Syncthreads()
2. **Warp-synchronous**
3. **Warp shuffle**

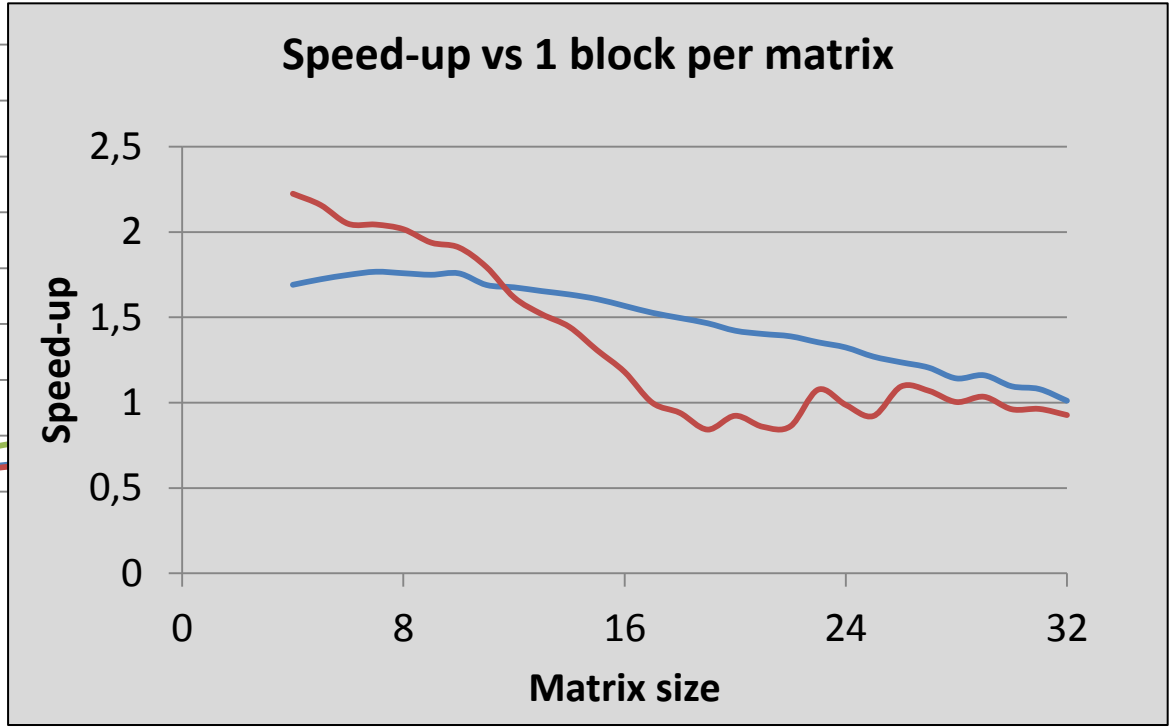
2 matrices per block is most consistent.

warp shuffle & Prefer L1

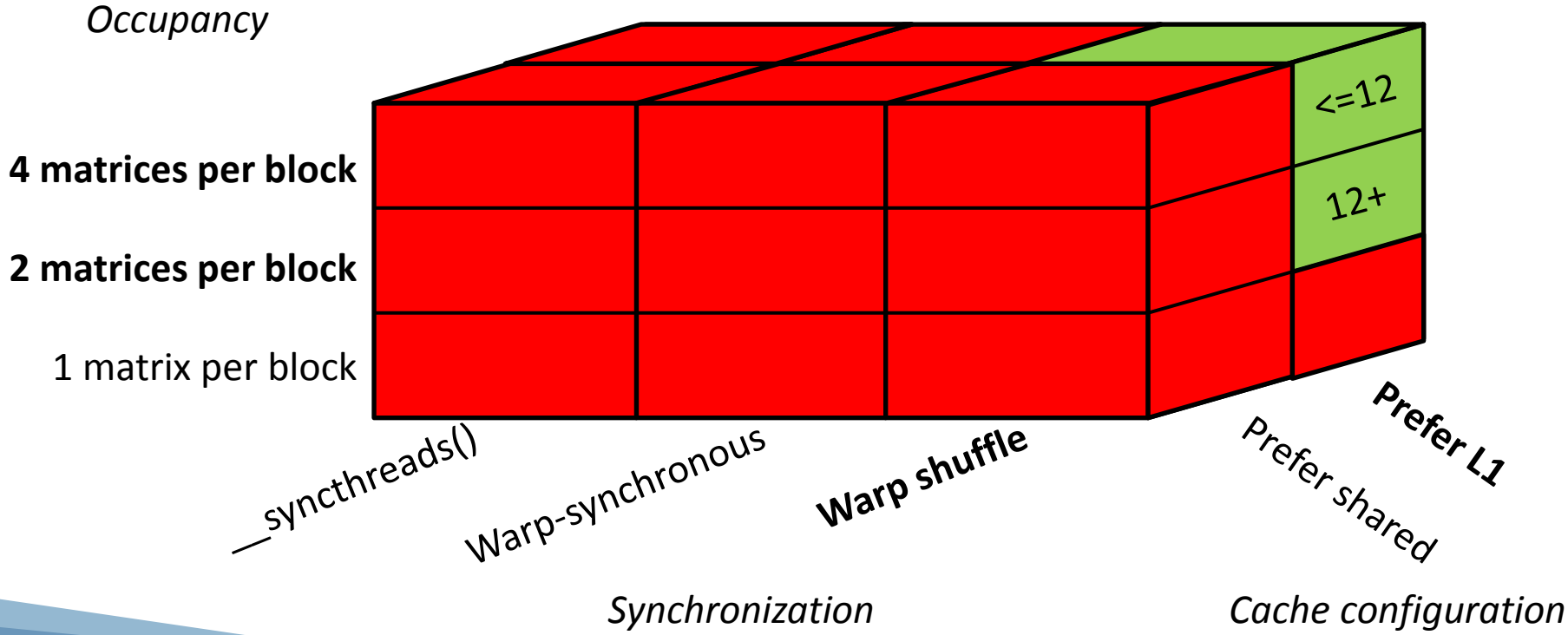
We can gain 2x by mapping 2 or 4 matrices to each block.

On average we gain **1.5x**.





Performance gains



Conclusions

1. Warp-synchronous is **1.7x faster** than __syncthreads().
2. Warp shuffle is **1.2x faster** than warp-synchronous.
3. 2 matrices per block is **1.5x faster** than 1 matrix per block.
4. Optimal cache config varies with implementation:
   Prefer shared if you use shared mem.
   Prefer L1 if you use warp shuffle.

Questions?

Ian Wainwright

High Performance Consulting Sweden

ian.wainwright@hpcsweden.se