



Optimized LU-decomposition with Full Pivot for Small Batched Matrices

S3069

Ian Wainwright

High Performance Consulting Sweden

ian.wainwright@hpcsweden.se



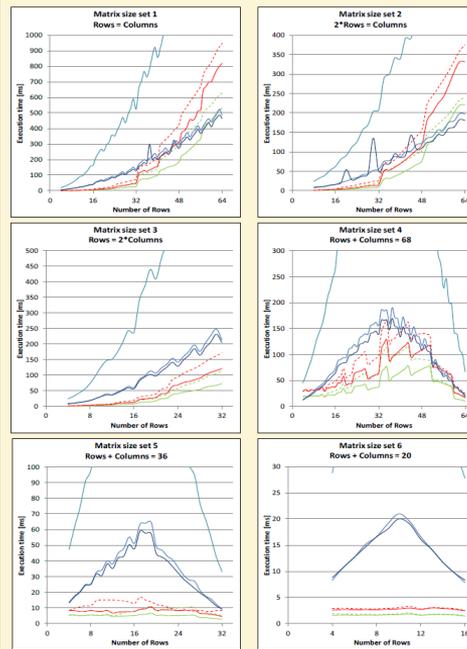
Background

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](http://www.hpcsweden.se/files/CUDA-based%20LU%20Factorization.pdf)

CUDA-based LU Factorization with pivoting for 10,000s of small dense matrices vs. Intel MKL

Fredrik Hellman (HPC), Jimmy Pettersson (HPC), Ian Wainwright (HPC)



LU Factorization with pivoting

LU factorization is a "high-level" algebraic description for Gaussian elimination. It is a fundamental operation performed in linear algebra when for example solving systems of linear equations, inverting a matrix or computing its determinant.

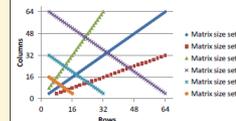
Implementation

The CUDA-based implementation stores almost all data in registers, each thread servicing one column. This minimizes the need for memory-access and allows for high performance until registers start to spill into memory. To fully utilize the CPU, function calls to Intel MKL have been parallelized using OpenMP and optimal thread affinity.

Conclusions

- GPUs are more than capable of performing dense linear algebra on small matrices, **achieving a speed-up factor of more than 10** vs. an 8-threaded Intel MKL implementation on a high-end CPU.
- When register spills occur, a discrete loss in performance often follows.
- Selecting 48 KIB L1 Cache instead of Shared memory helps minimize the impact of excessive register spills.

The 6 Matrix size sets benchmarked, each using 50,000 matrices per run



Non-GPU Hardware	Software
Intel Core i7 960, 3.20 GHz, 8 Logical Cores	Windows 7 64-bit SP1 CUDA 5.06 32-bit driver
ASUS P8T Deluxe V2	CUDA Toolkit 4.1
12 GB RAM	
GPUs	GTX 480 GTX 560 TI
Core count	480 384
Core clock	1.40 GHz 1.54 GHz
Max Theoretical Compute Power	1344 GFLOPS 1250 GFLOPS
Memory Bus Width	384-bit 256-bit
Memory clock	1.948 GHz 2.00 GHz
Max Bandwidth with SDRAM	146.5 GB/s 105.8 GB/s

--- GTX 480 L1 preference
--- GTX 480 Shared preference
--- GTX 560 TI L1 preference
--- GTX 560 TI Shared preference
--- Intel MKL 1 OMP CPU Thread
--- Intel MKL 4 OMP CPU Threads
--- Intel MKL 8 OMP CPU Threads

High Performance Consulting is a Sweden-based consultancy company specializing in GPGPU.
www.hpcsweden.se
info@hpcsweden.se





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



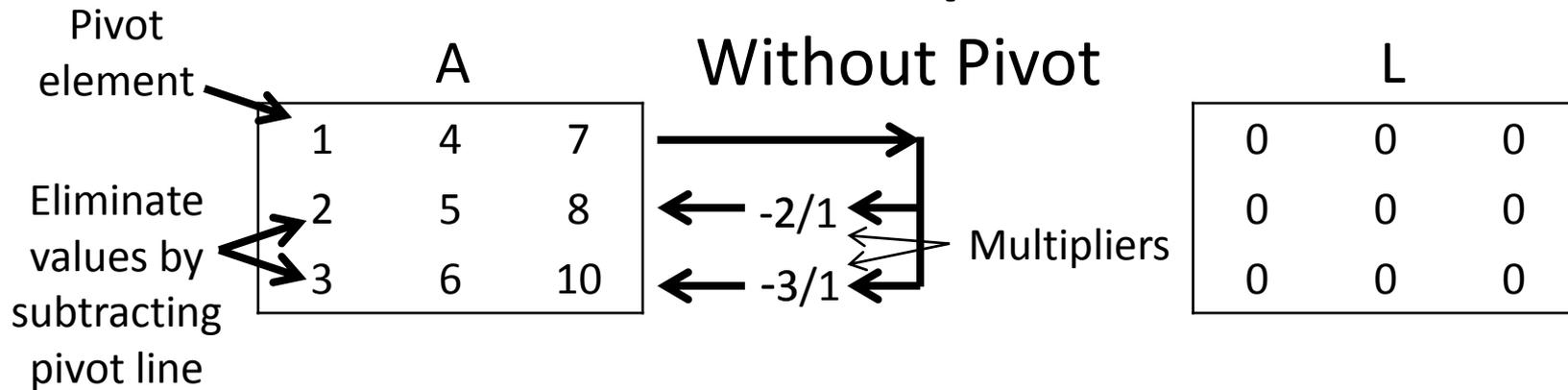
1. LU-decomposition
2. Implementations
3. Performance gains
4. Conclusions

LU-decomposition

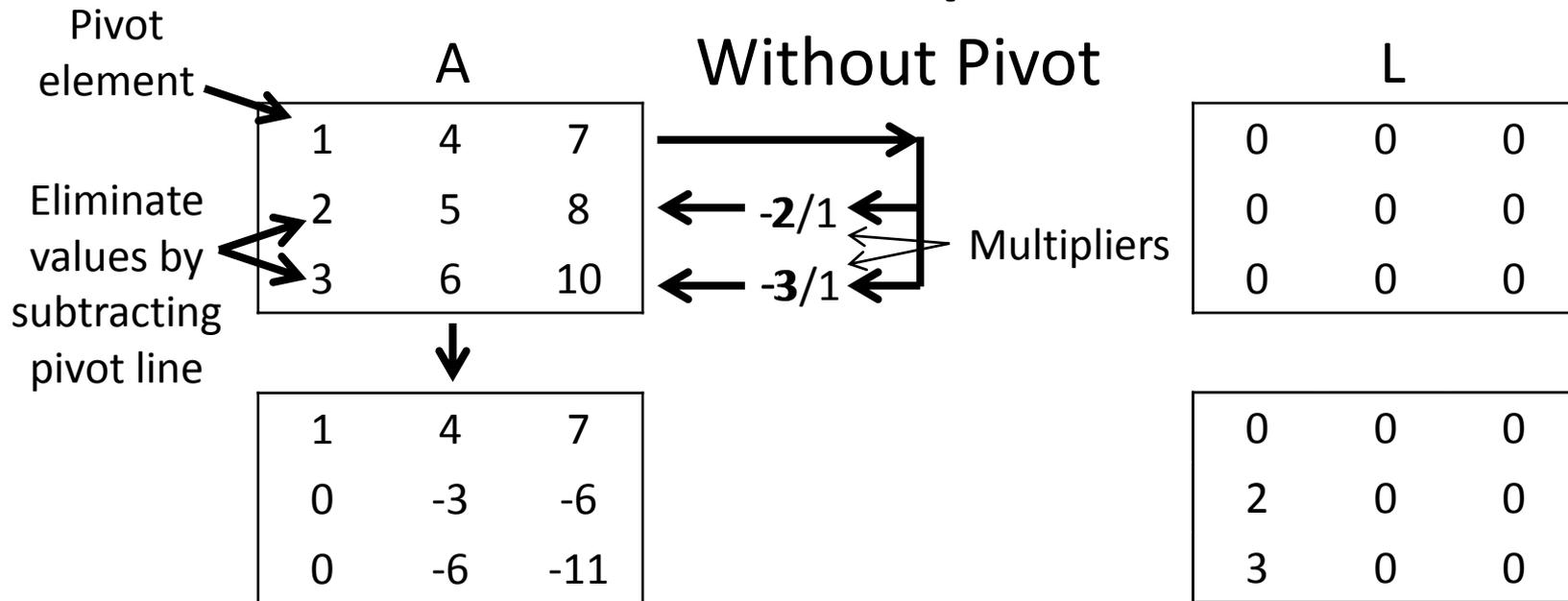
The idea is to transform A where $Ax=b$ to an equivalent triangular system such that $A=LU$

$$\begin{array}{cccc}
 & \mathbf{A} & & \\
 a_{11} & a_{12} & a_{13} & a_{14} \\
 a_{21} & a_{22} & a_{23} & a_{24} \\
 a_{31} & a_{32} & a_{33} & a_{34} \\
 a_{41} & a_{42} & a_{43} & a_{44}
 \end{array}
 =
 \begin{array}{cccc}
 & \mathbf{L} & & \\
 1 & 0 & 0 & 0 \\
 l_{21} & 1 & 0 & 0 \\
 l_{31} & l_{32} & 1 & 0 \\
 l_{41} & l_{42} & l_{43} & 1
 \end{array}
 *
 \begin{array}{cccc}
 & \mathbf{U} & & \\
 u_{11} & u_{12} & u_{13} & u_{14} \\
 0 & u_{22} & u_{23} & u_{24} \\
 0 & 0 & u_{33} & u_{34} \\
 0 & 0 & 0 & u_{44}
 \end{array}$$

LU-decomposition



LU-decomposition



LU-decomposition

A

1	4	7
2	5	8
3	6	10

Without Pivot

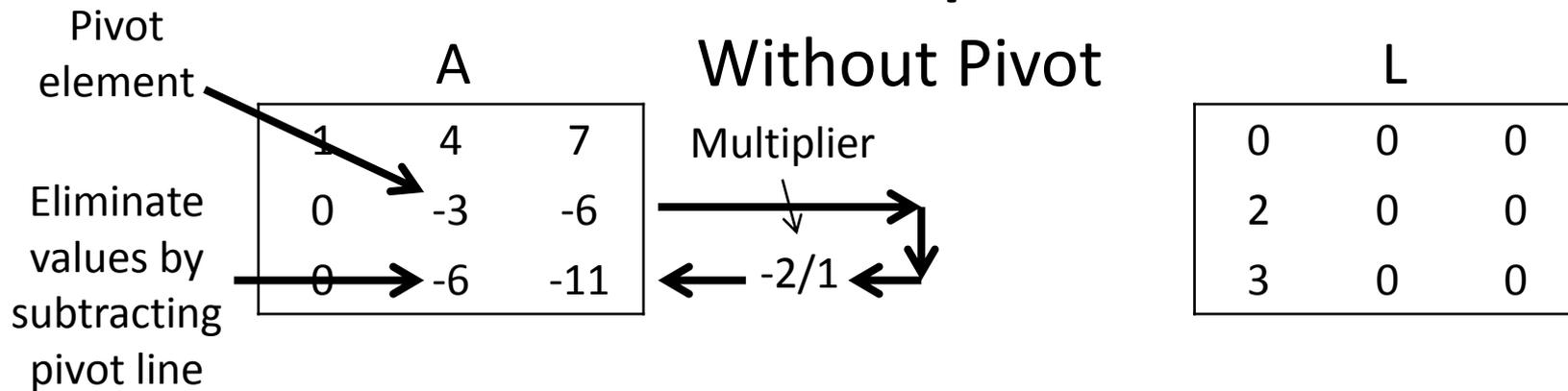
L

0	0	0
0	0	0
0	0	0

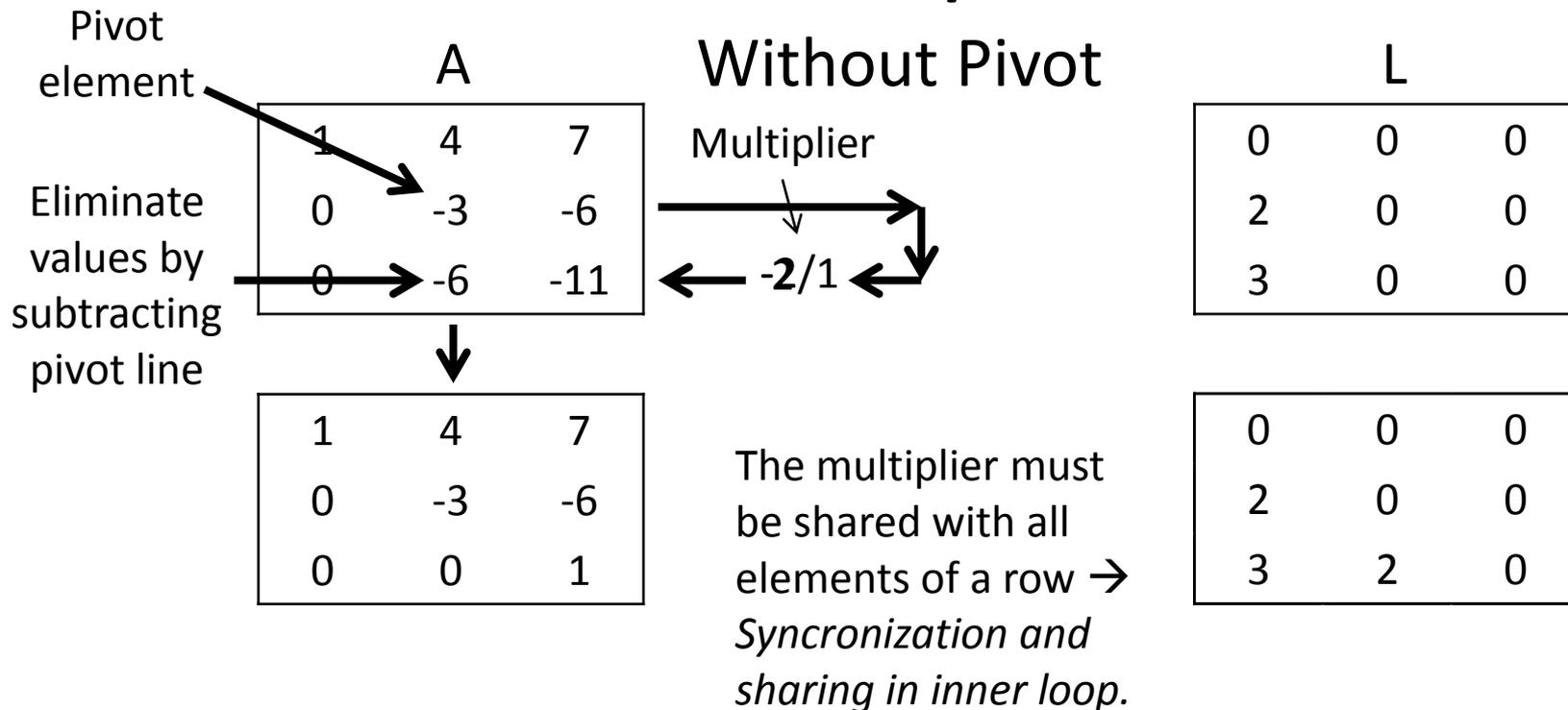
1	4	7
0	-3	-6
0	-6	-11

0	0	0
2	0	0
3	0	0

LU-decomposition



LU-decomposition



LU-decomposition

A

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

Without Pivot

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

A

1	4	7
0	-3	-6
0	0	1

Without Pivot

L

0	0	0
2	0	0
3	2	0

LU-decomposition

U

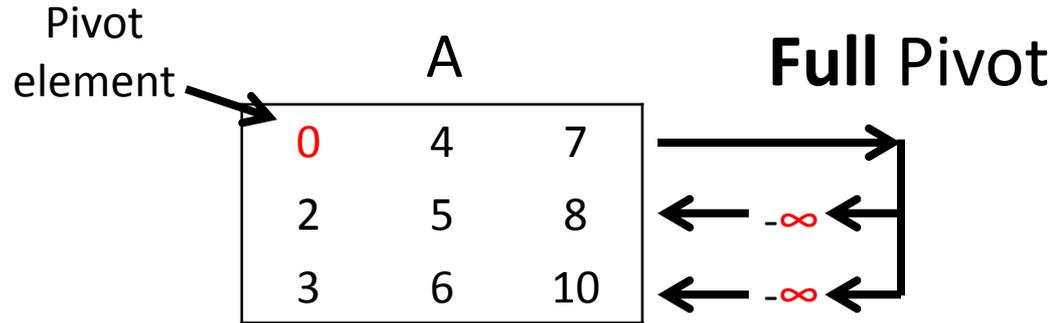
1	4	7
0	-3	-6
0	0	1

Without Pivot

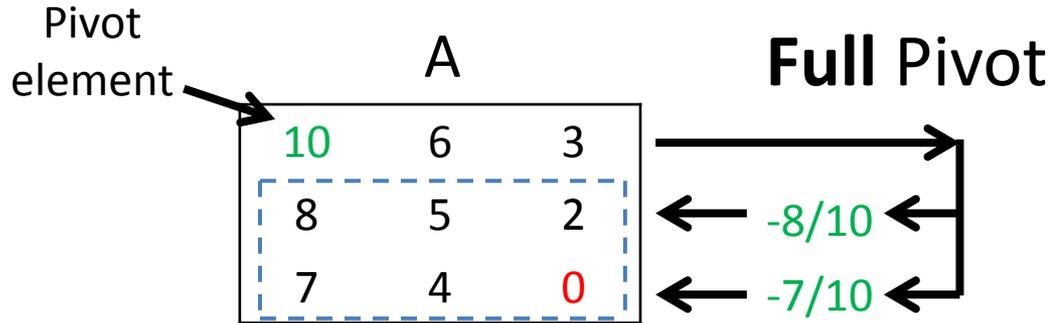
L

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

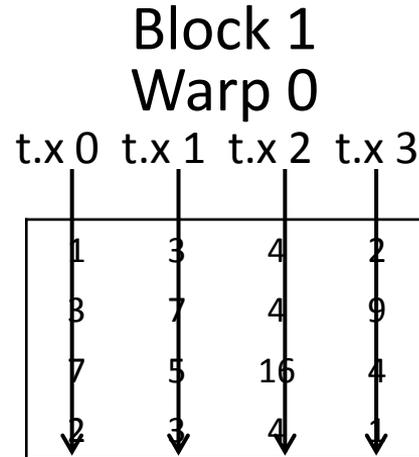
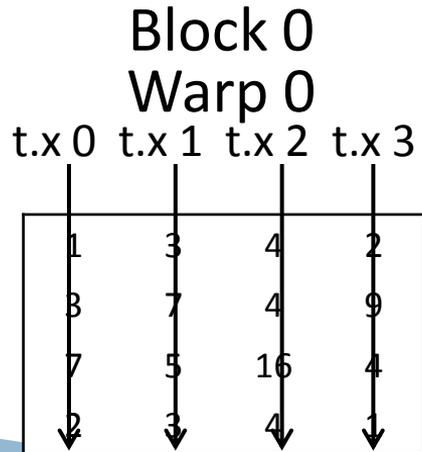
Mapping of problem:

- 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.



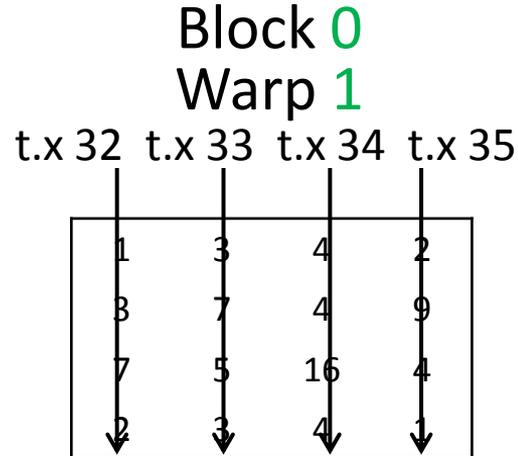
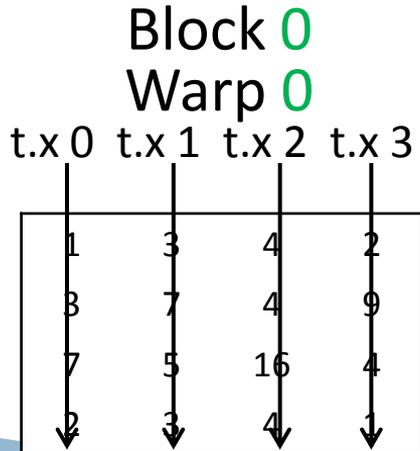
One matrix per block

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



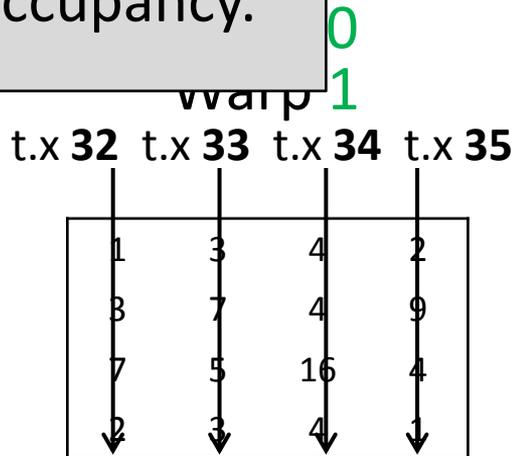
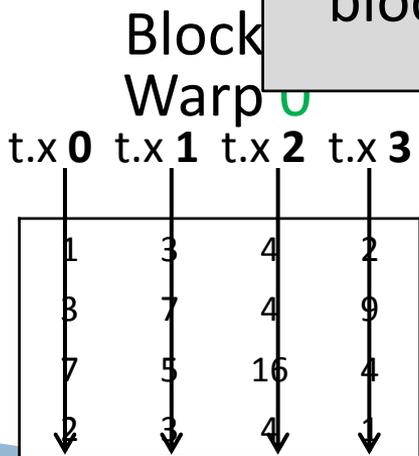
Implementations

Mapping of problem:

- Map one matrix to a warp.
- Map a column
- One or more

Two matrix per block

The number of matrices per block affects the occupancy.





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.

Synchronization and data sharing:

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

Cache-config:

1. Prefer shared.
2. Prefer L1.

Occupancy

Synchronization

Cache configuration

and their interaction

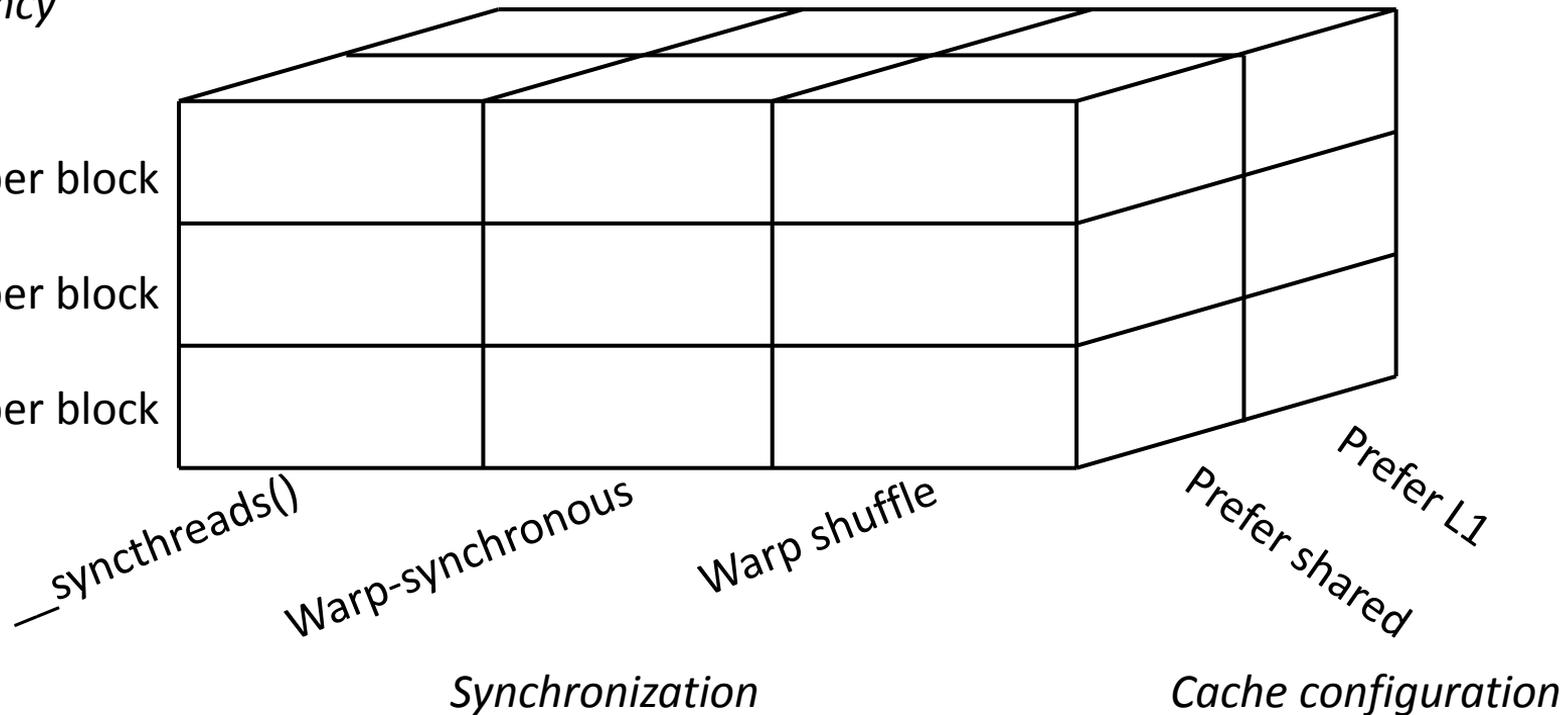
Implementations

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block





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.

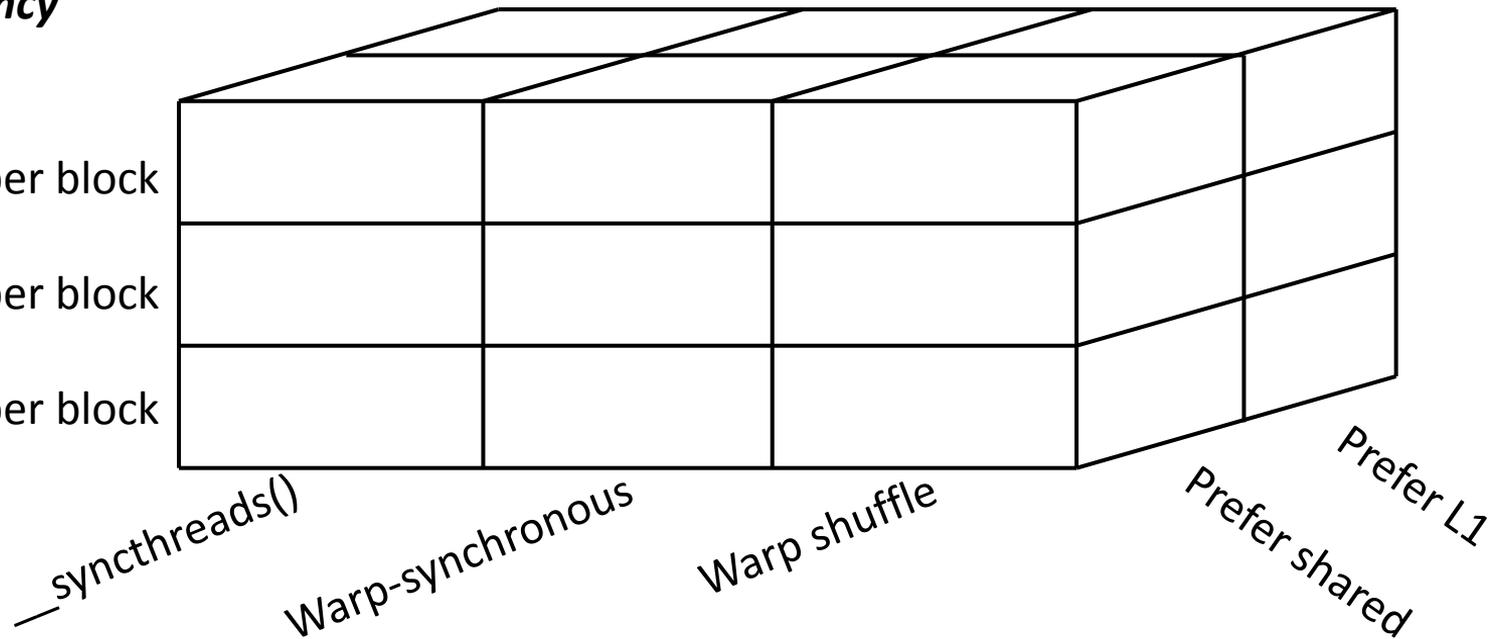
Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block

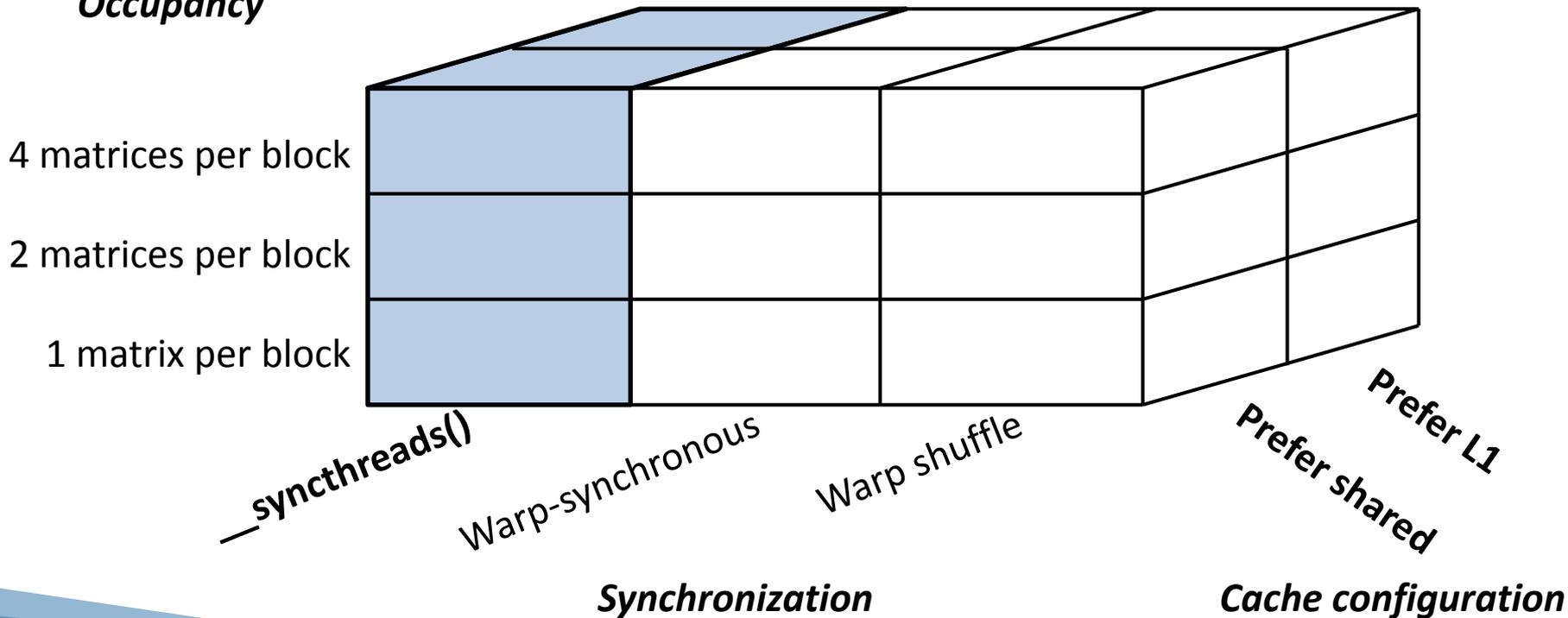


Synchronization

Cache configuration

Performance gains

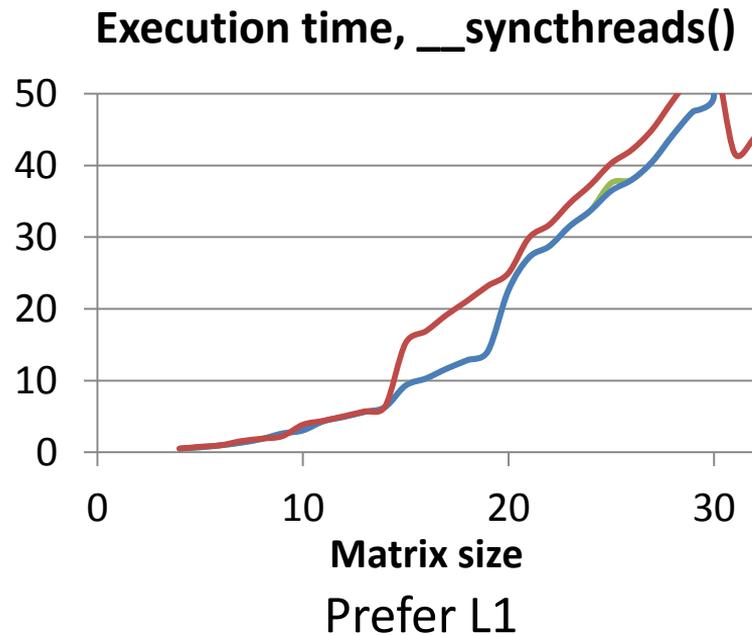
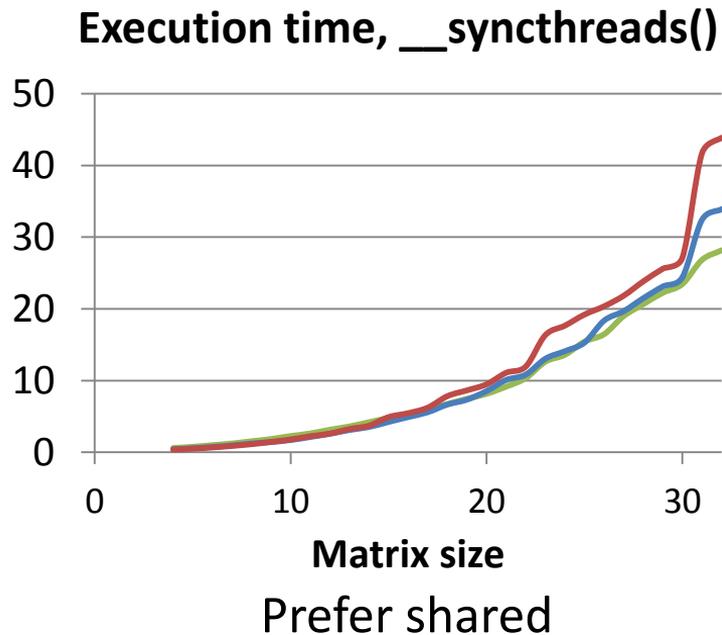
Occupancy



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

Performance gains

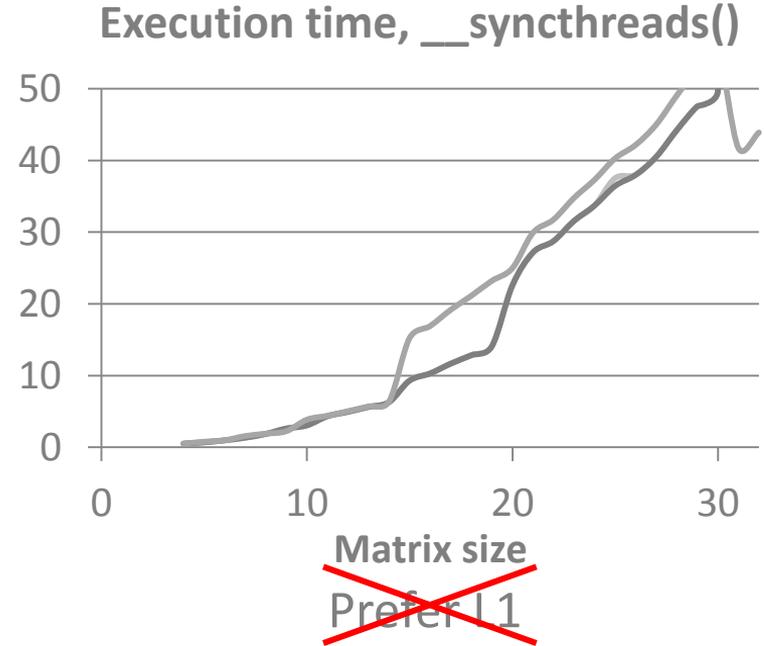
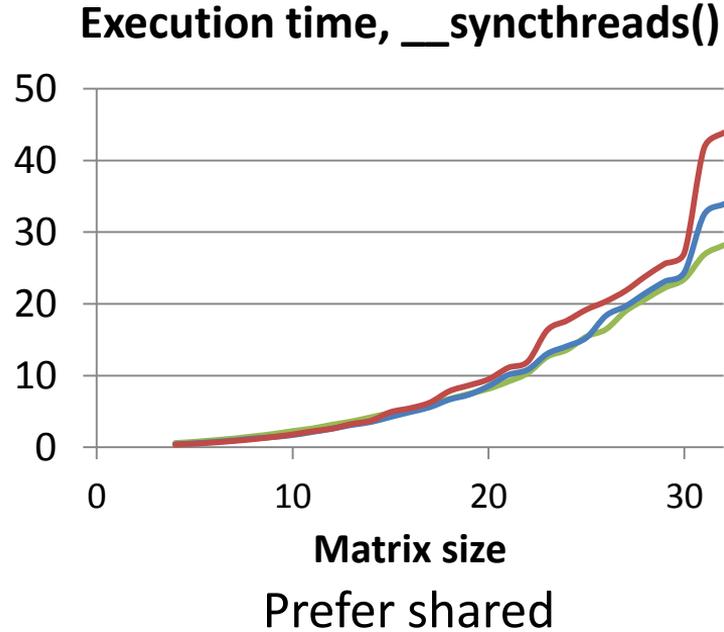
1. Syncthreads()
2. Warp-synchronous
3. Warp shuffle



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

Performance gains

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



Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block

`—syncthreads()`

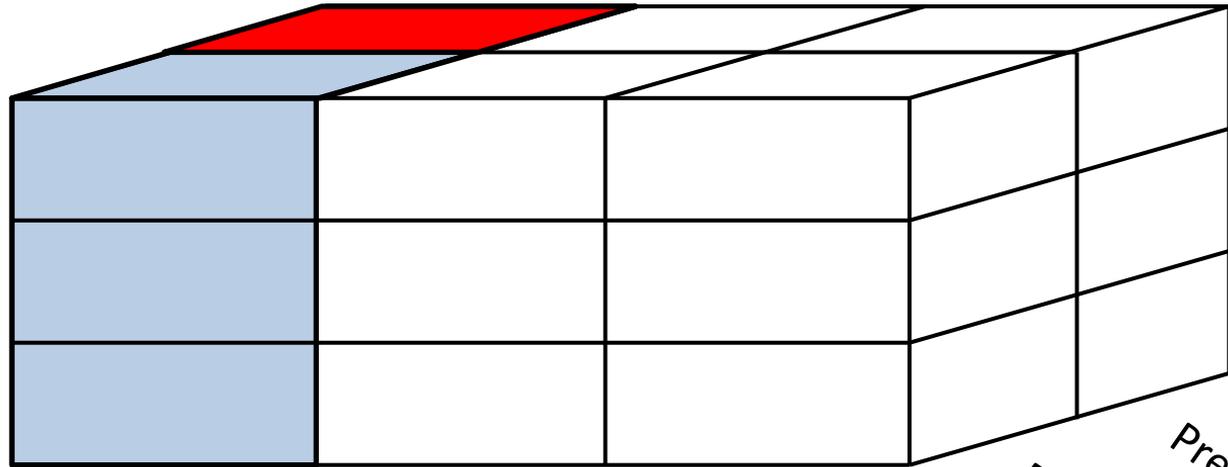
Warp-synchronous

Warp shuffle

Prefer shared
Prefer L1

Synchronization

Cache configuration



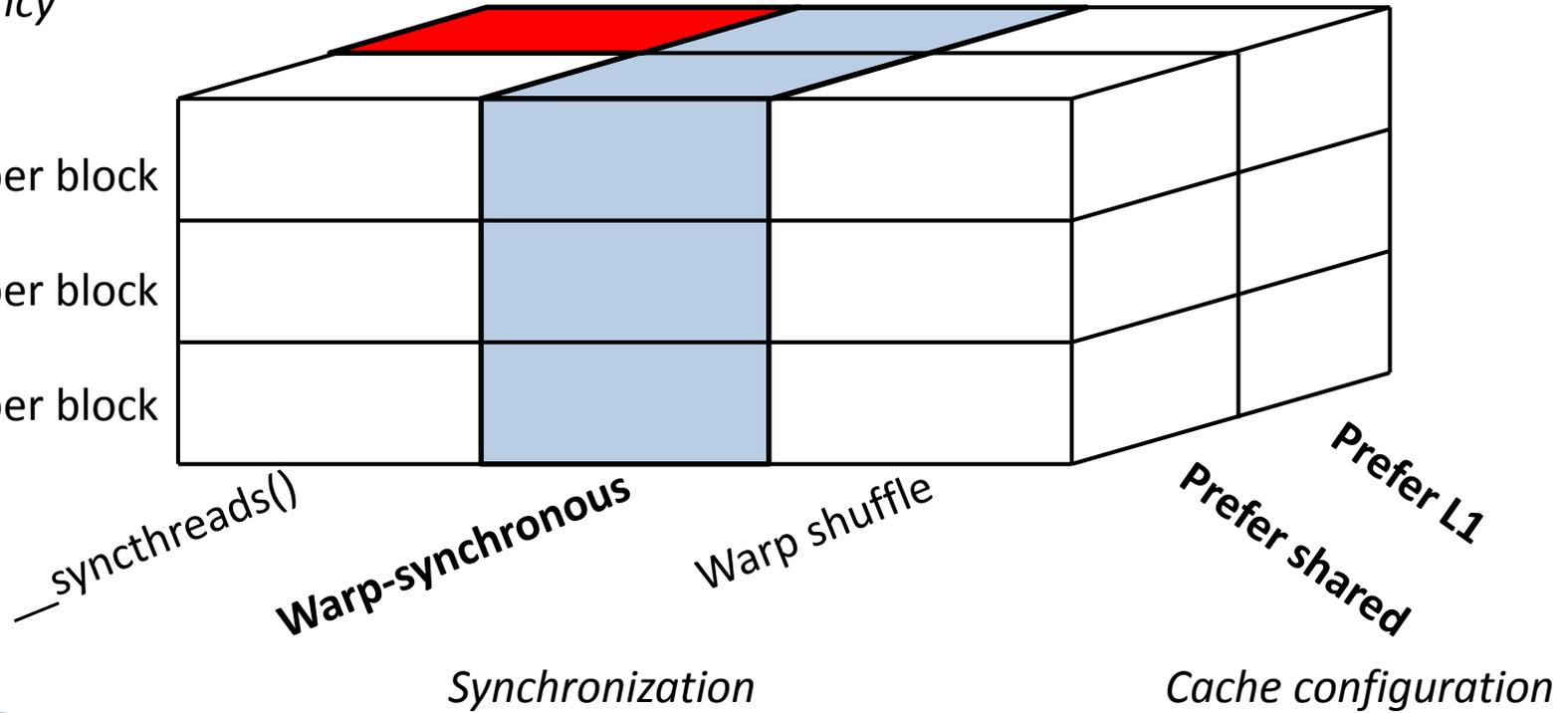
Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block

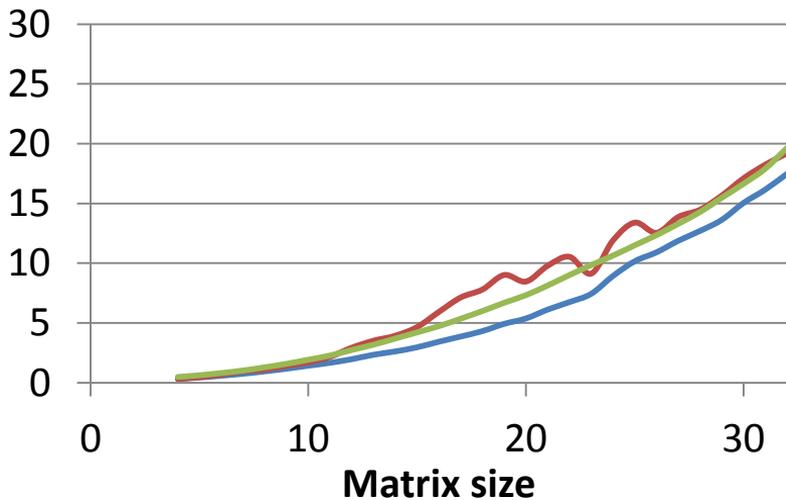


- 1 matrix per block
- 2 matrices per block
- 4 matrices per block

Performance gains

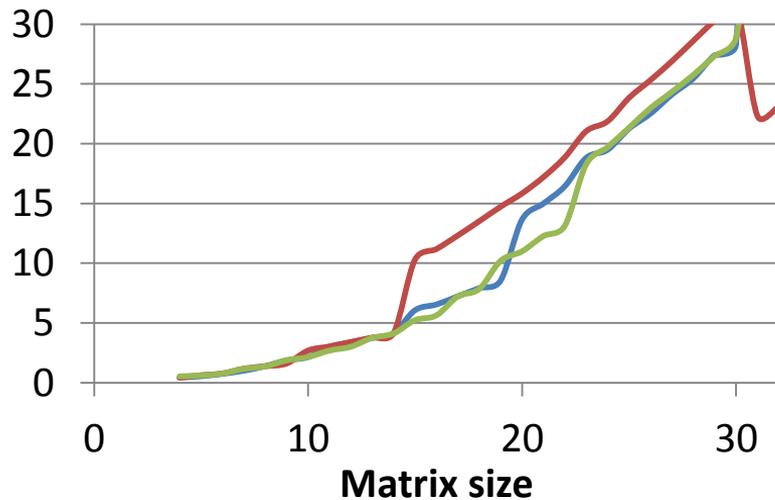
1. Syncthreads()
2. Warp-synchronous
3. Warp shuffle

Execution time, warp-synchronous



Prefer shared

Execution time, warp-synchronous



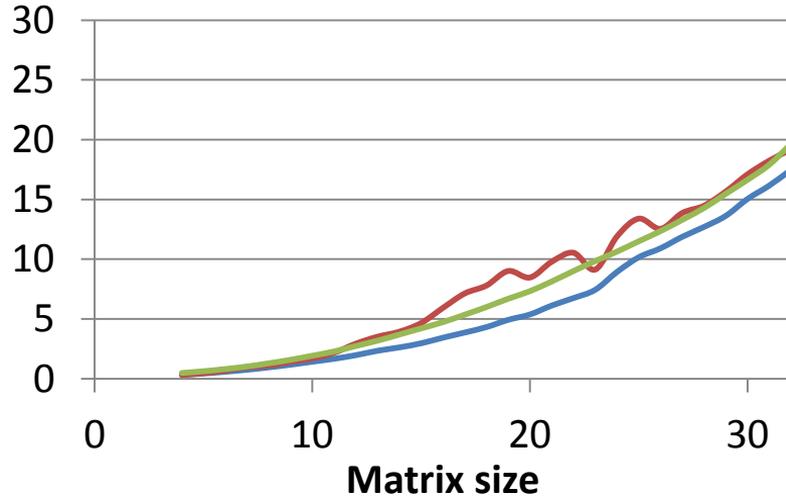
Prefer L1

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

Performance gains

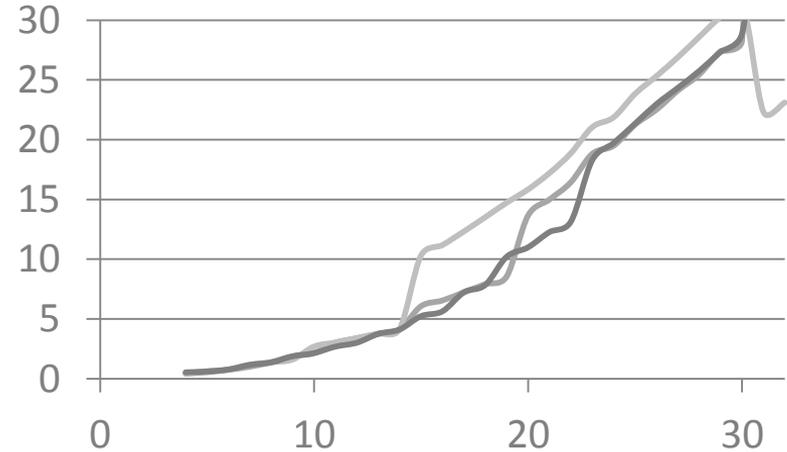
1. Synctreads()
2. **Warp-synchronous**
3. Warp shuffle

Execution time, warp-synchronous



Prefer shared

Execution time, warp-synchronous



~~Matrix size~~
~~Prefer L1~~

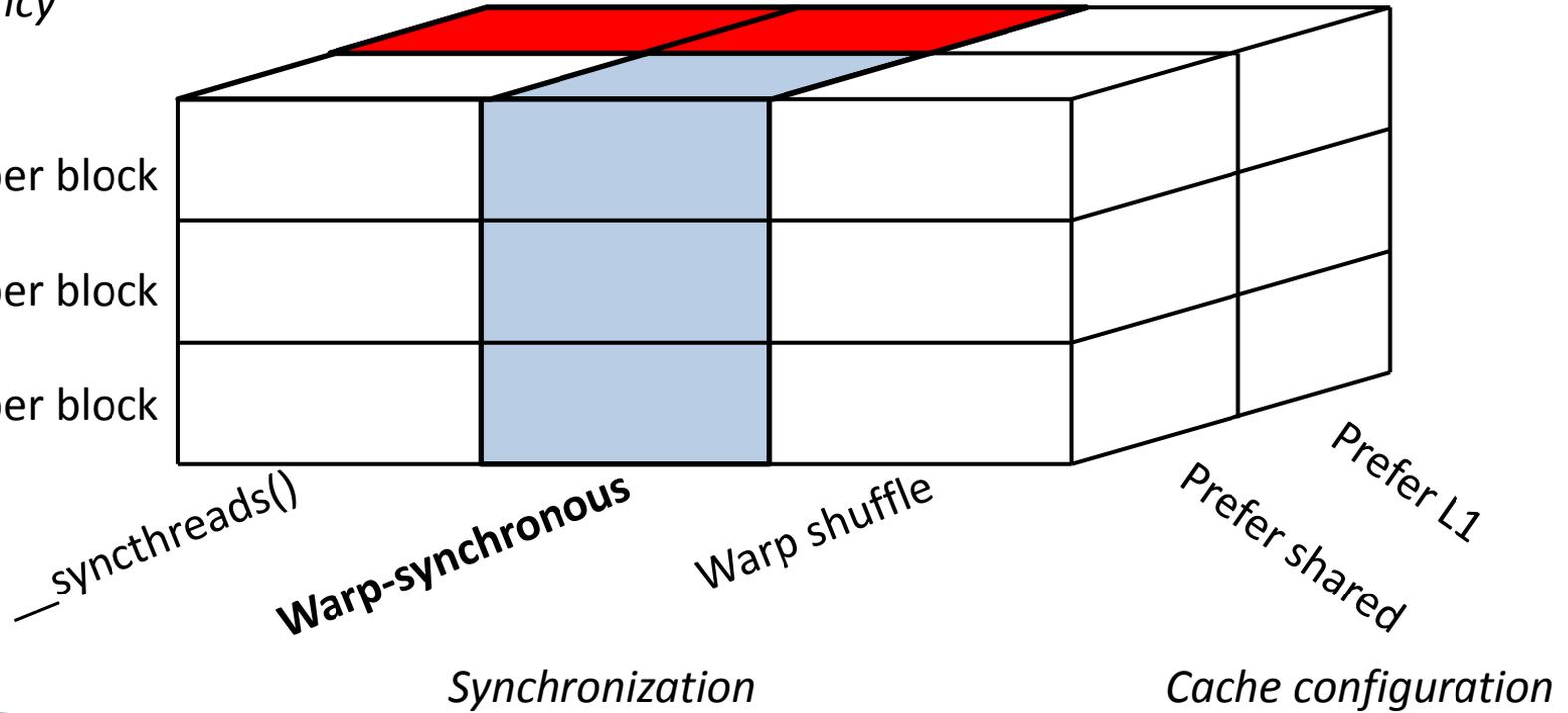
Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block



Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block

No surprise: we use shared memory for all data-sharing and reduction. More shared memory gives us higher occupancy → Better performance.

`—syncthreads()`

Warp-synchronous

Warp shuffle

Prefer shared
Prefer L1

Synchronization

Cache configuration

Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block

`—syncthreads()`

Warp-synchronous

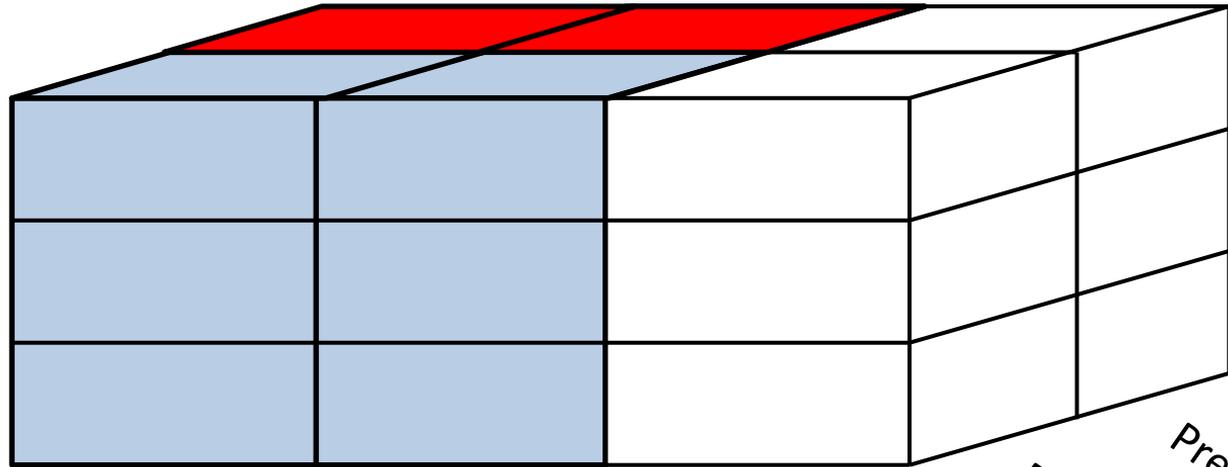
Warp shuffle

Prefer shared

Prefer L1

Synchronization

Cache configuration

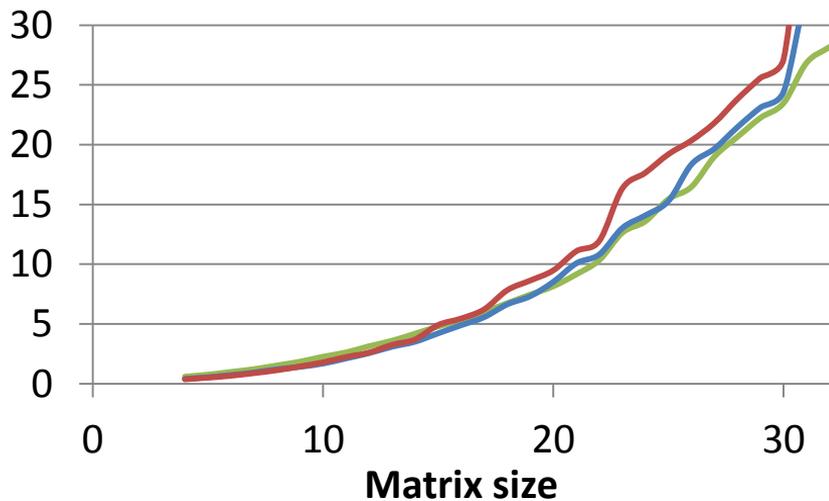


- 1 matrix per block
- 2 matrices per block
- 4 matrices per block

Performance gains

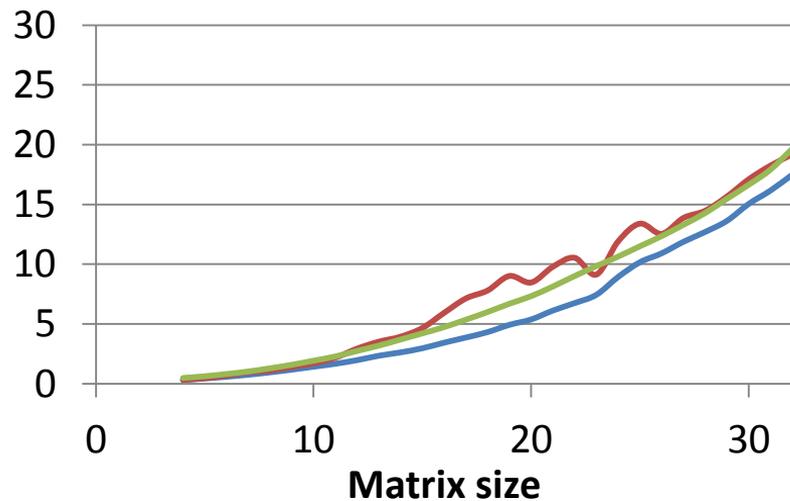
1. Synctreads()
2. Warp-synchronous
3. Warp shuffle

Execution time[ms], __synctreads()



__synctreads()

Execution time[ms], warp-synchronous()

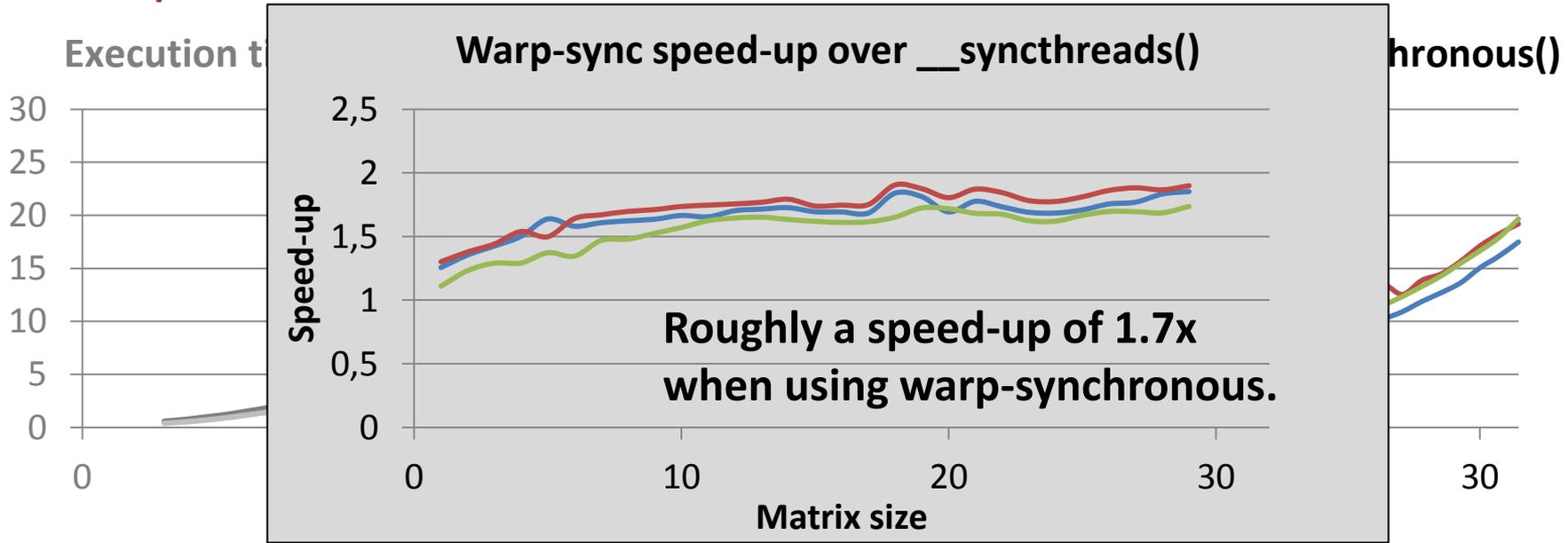


Warp-synchronous

- 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

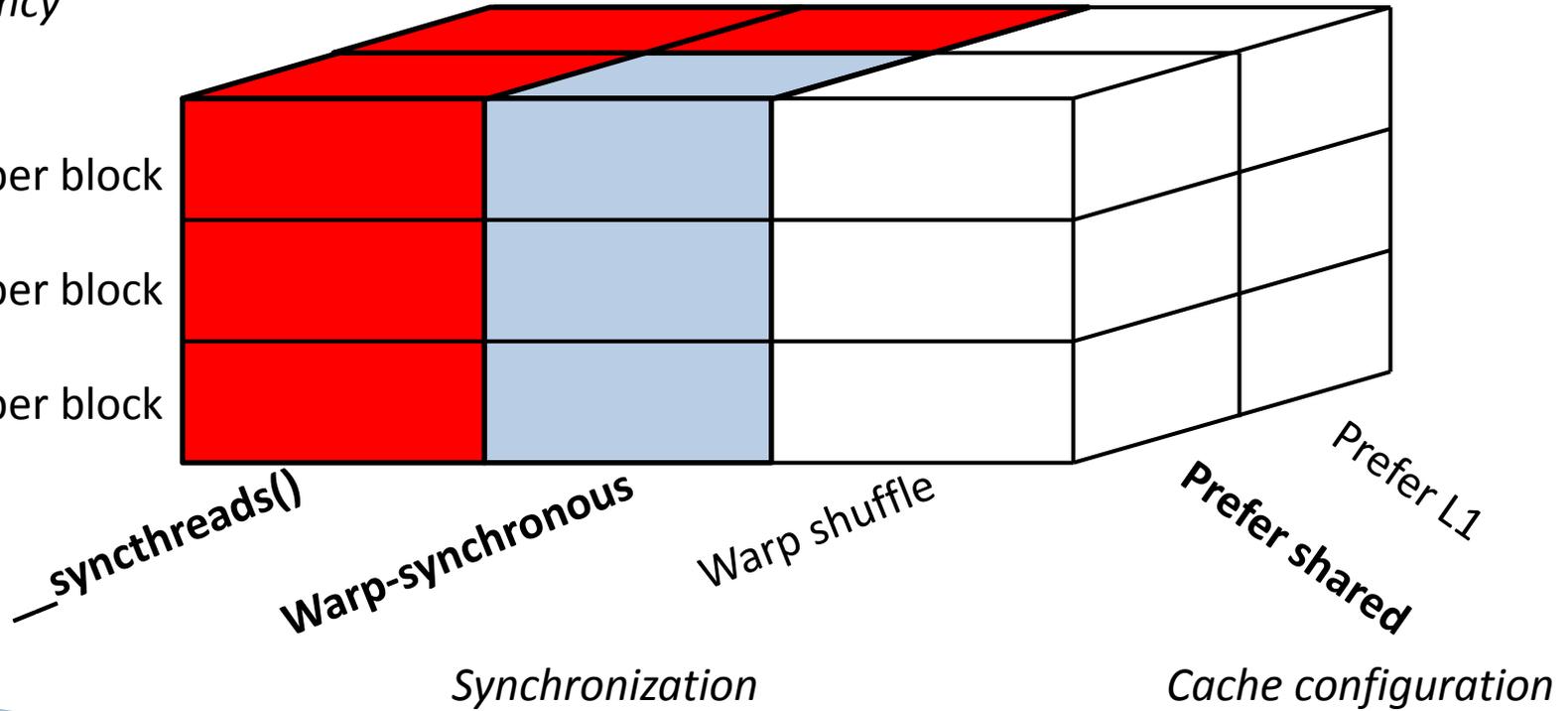
Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block



`__syncthreads()`

Synchronization

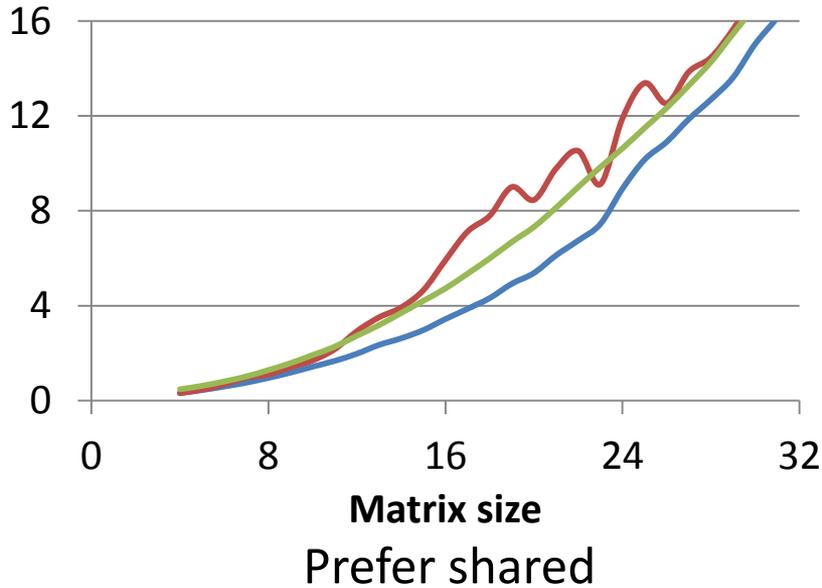
Cache configuration

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

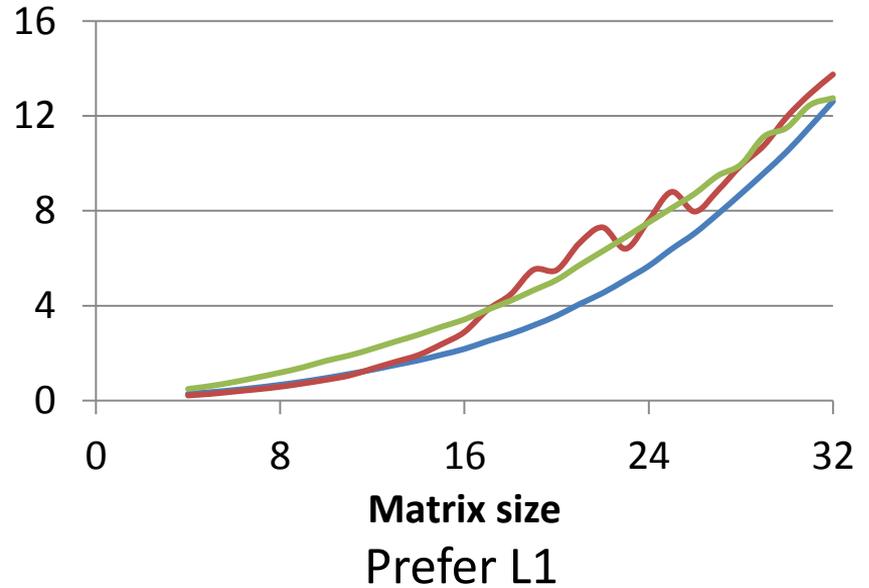
Performance gains

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

Execution time, warp shuffle



Execution time, warp shuffle

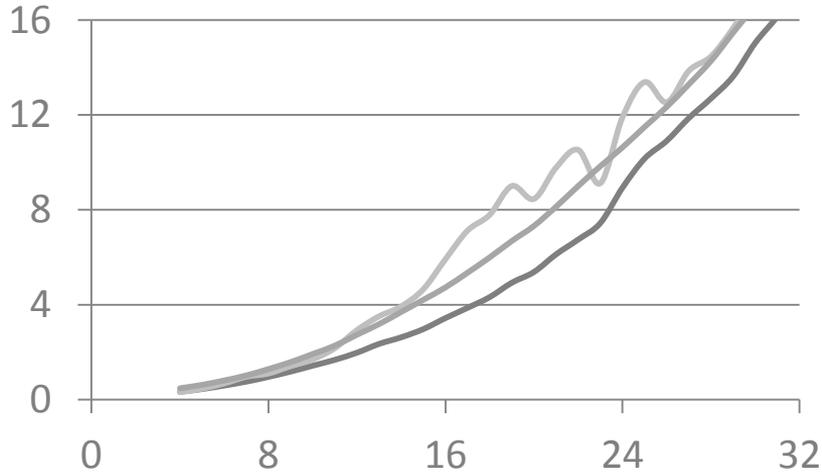


- 1 matrix per block
- 2 matrices per block
- 4 matrices per block

Performance gains

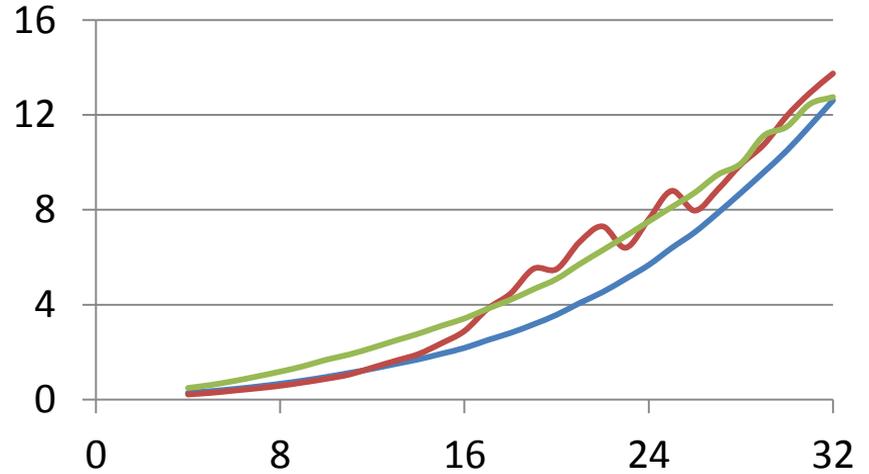
1. Syncthreads()
2. Warp-synchronous
3. **Warp shuffle**

Execution time, warp shuffle



~~Matrix size~~
~~Prefer shared~~

Execution time, warp shuffle



Matrix size
Prefer L1

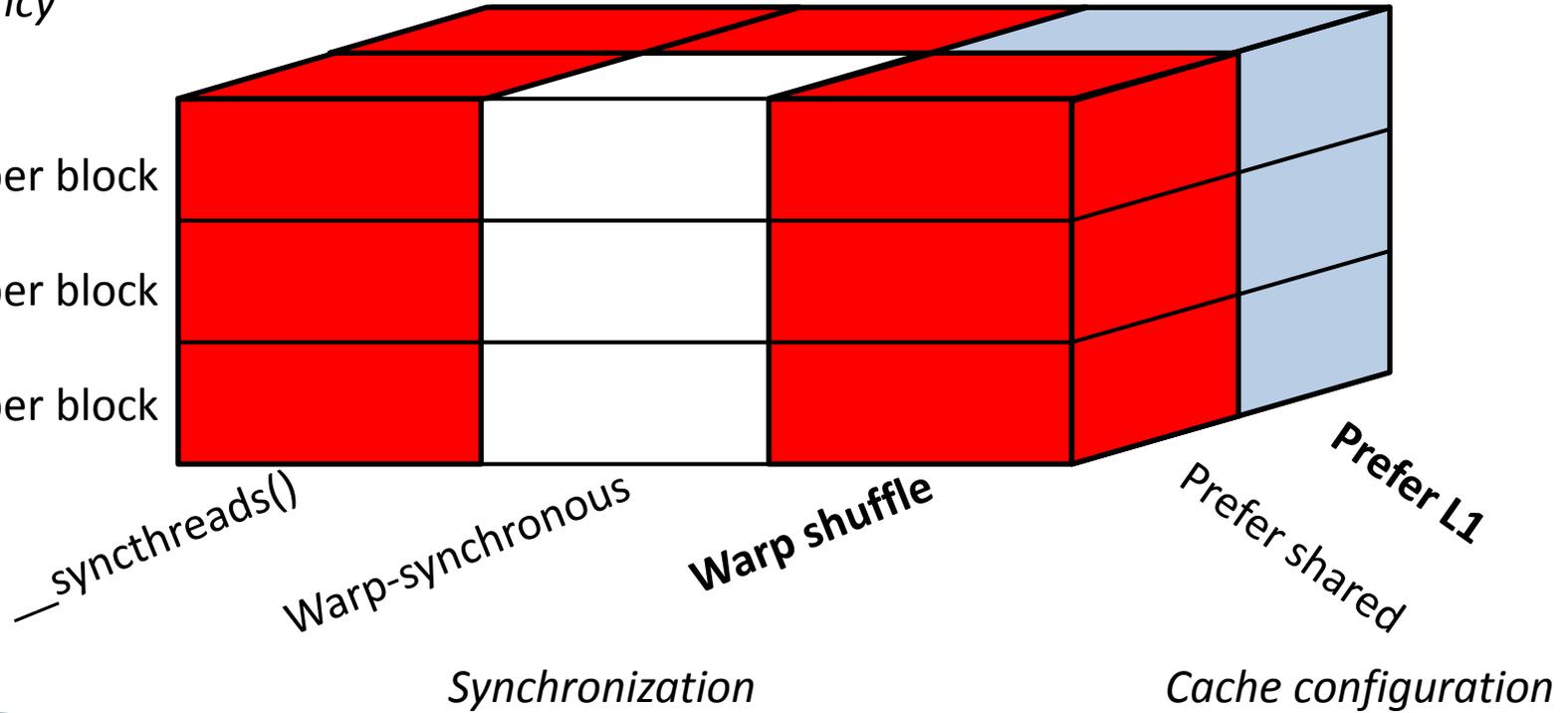
Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block



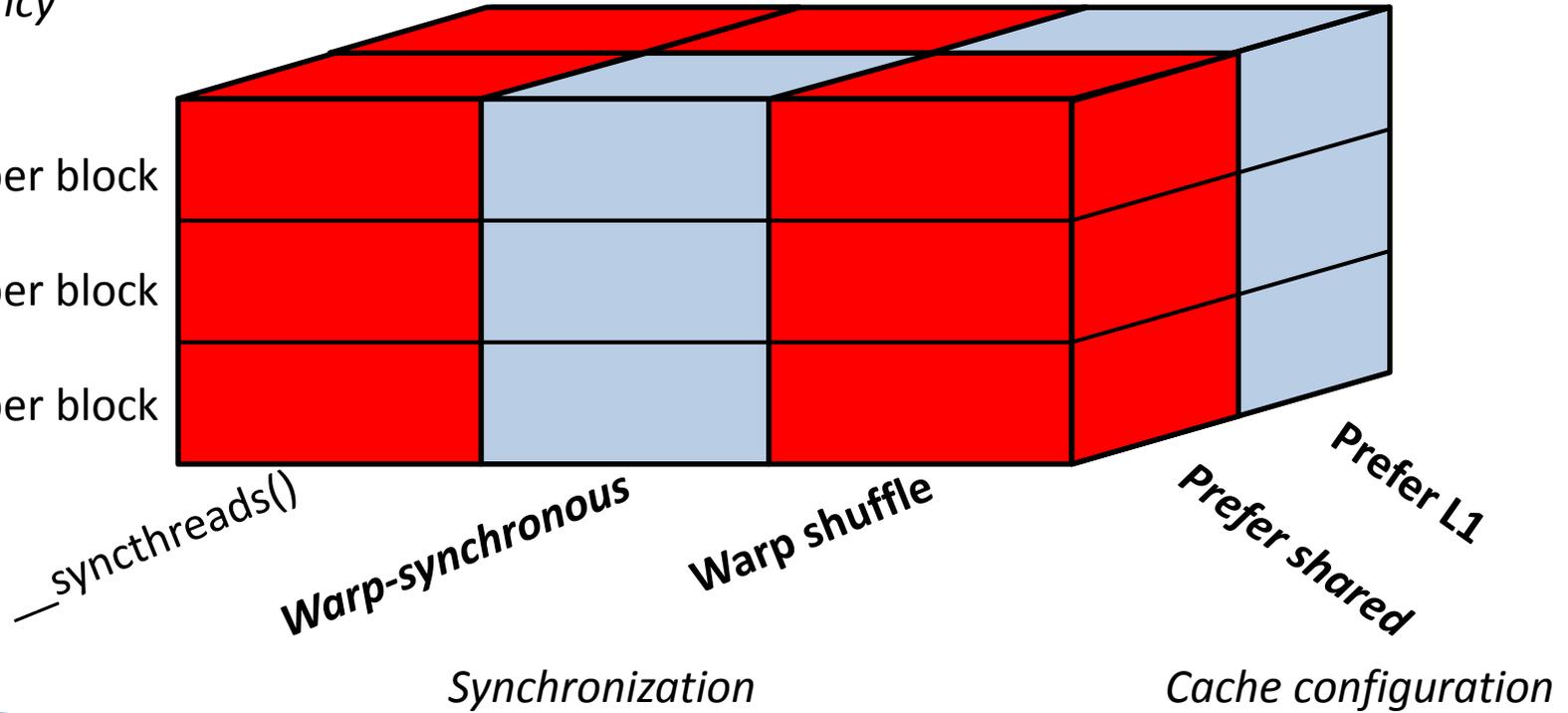
Performance gains

Occupancy

4 matrices per block

2 matrices per block

1 matrix per block

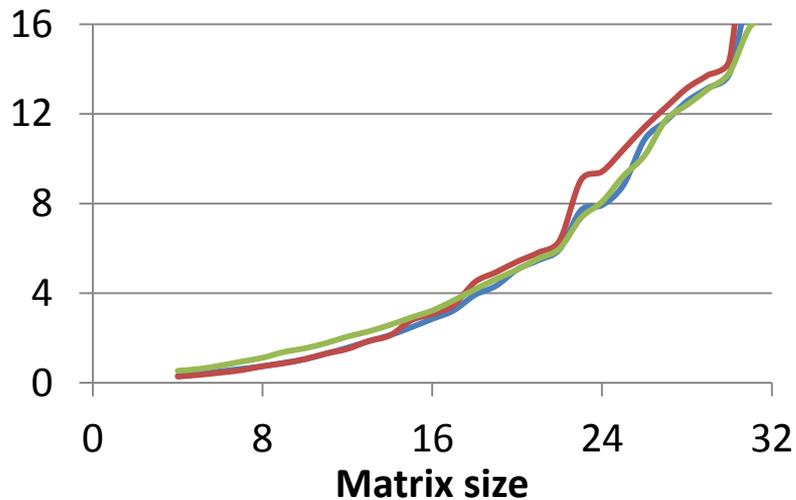


- 1 matrix per block
- 2 matrices per block
- 4 matrices per block

Performance gains

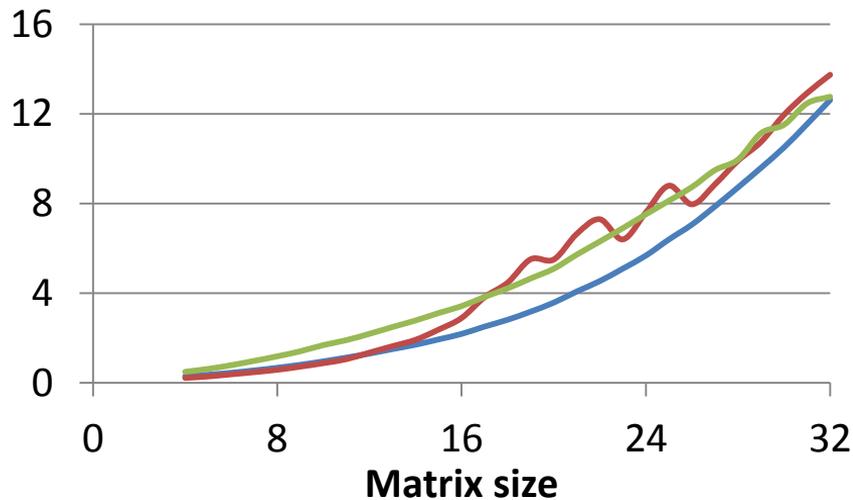
1. Syncthreads()
2. Warp-synchronous
3. Warp shuffle

Execution time, warp-synchronous



Prefer shared

Execution time, warp shuffle



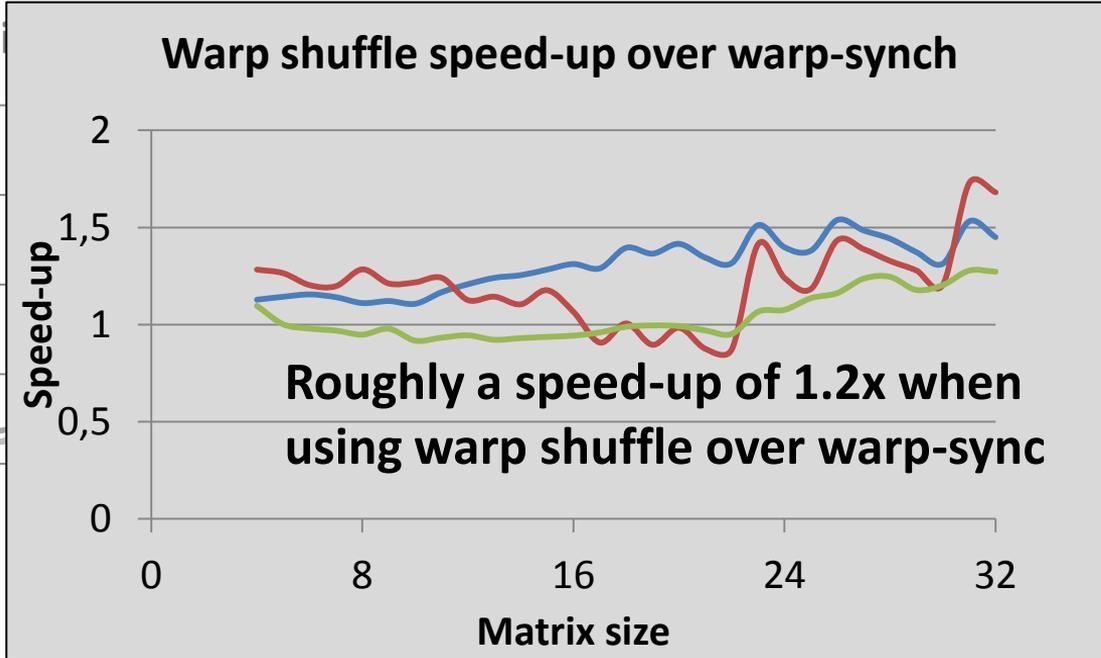
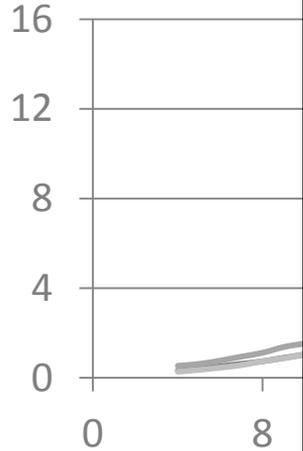
Prefer L1

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

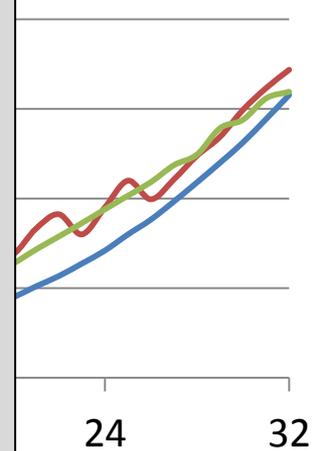
Performance gains

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

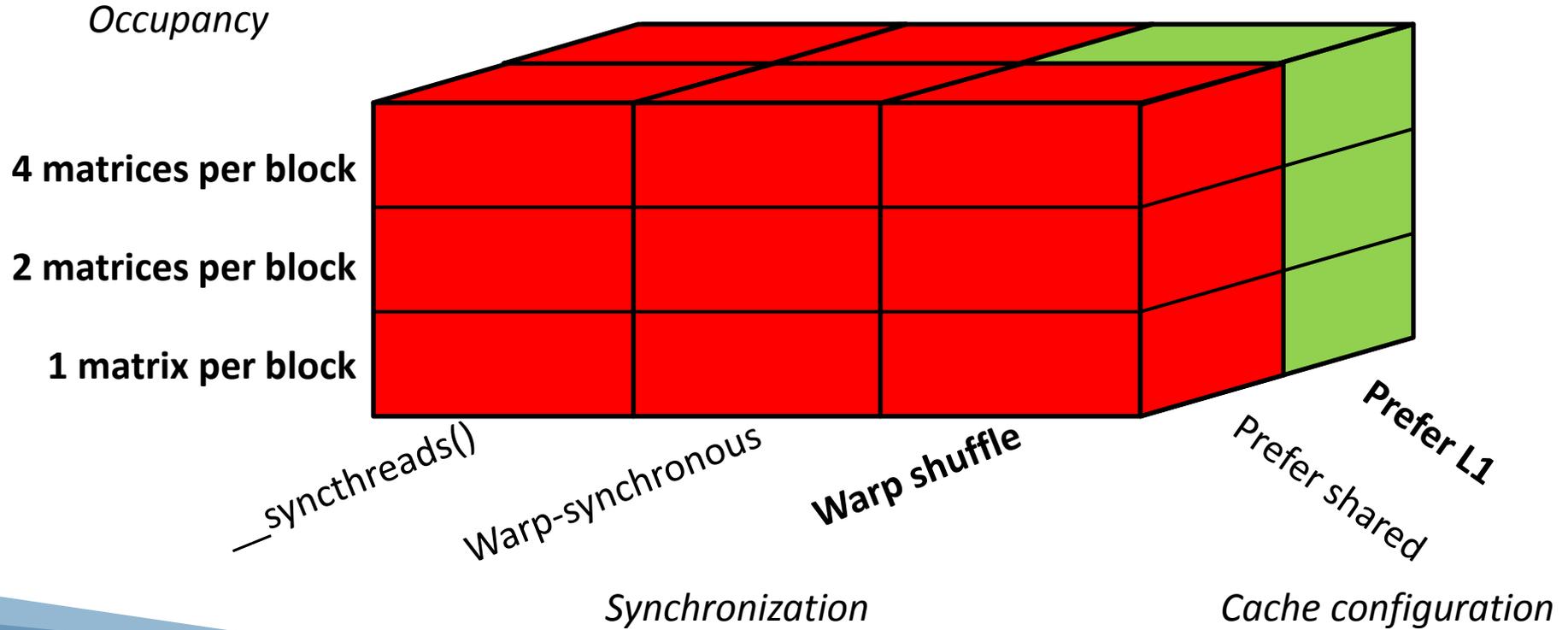
Execution time



Warp shuffle



Performance gains

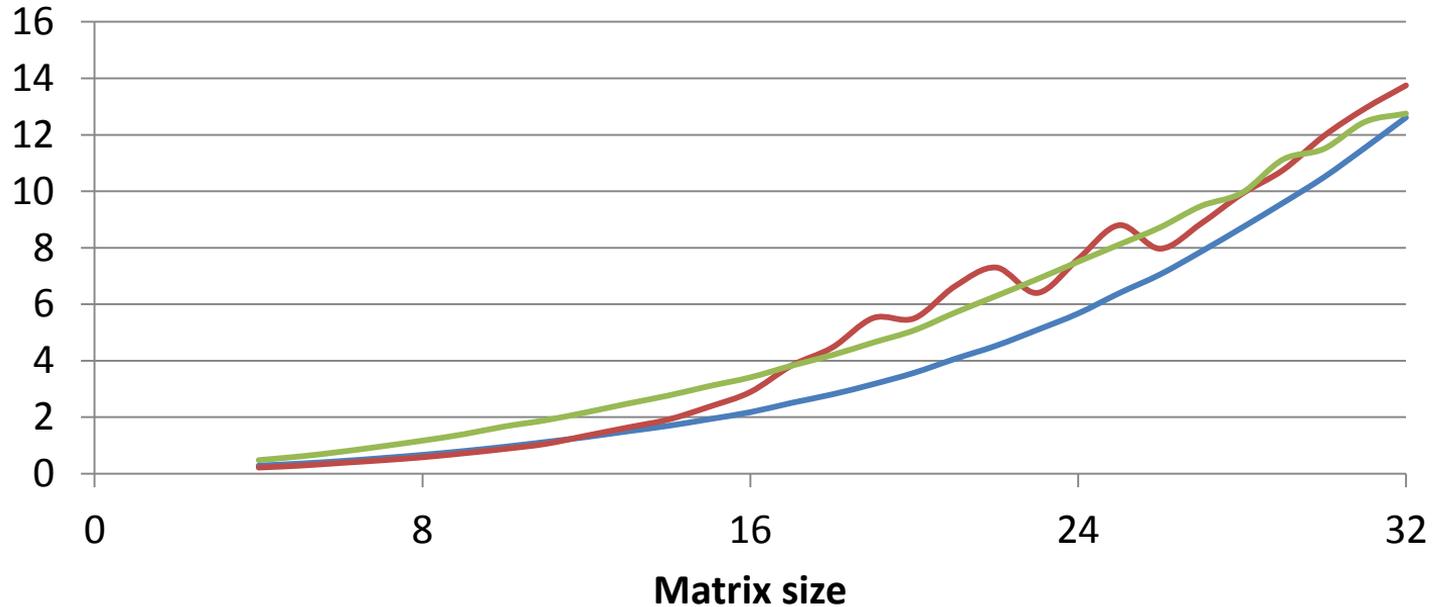


- 1 matrix per block
- 2 matrices per block
- 4 matrices per block

Performance gains

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

Execution time, warp shuffle & Prefer L1



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

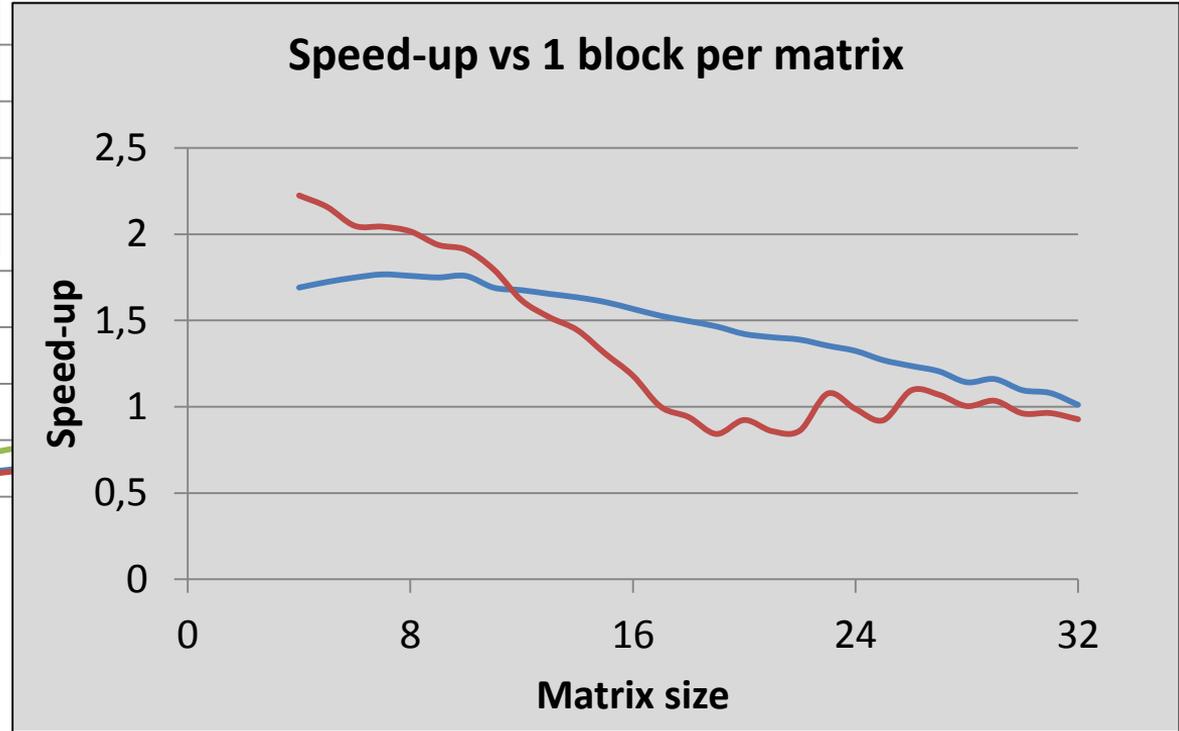
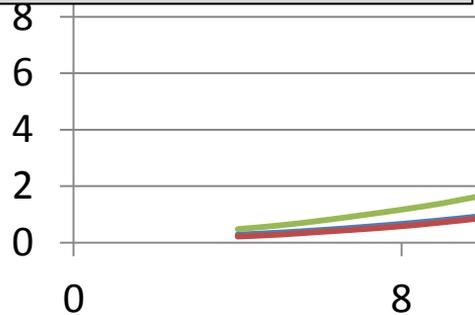
Performance gains

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

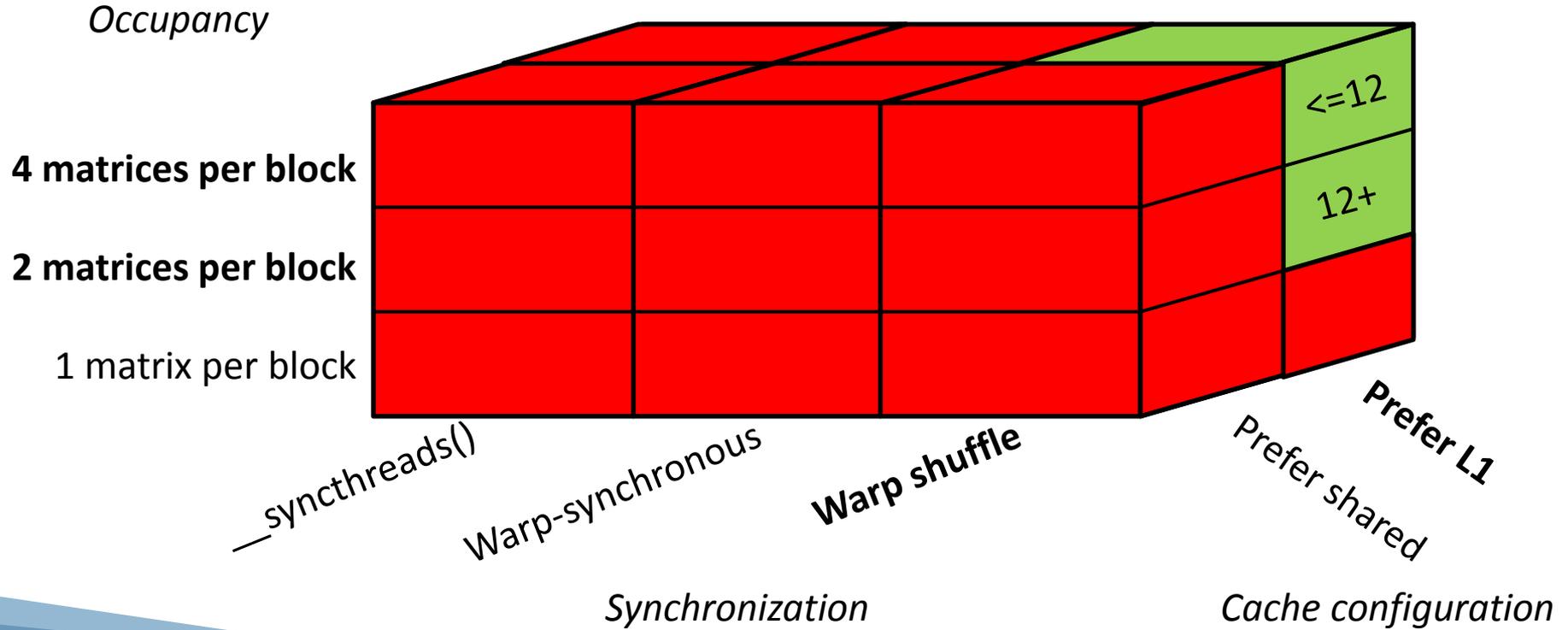
2 matrices per block is most consistent. warp shuffle & Prefer L1

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