Accelerating GPU Kernels for Dense Linear Algebra

Rajib Nath,
Stan Tomov, and
Jack Dongarra

Innovative Computing Lab
University of Tennessee, Knoxville

July 9, 2010

xGEMM performance of CUBLAS-2.3 on GTX280

Oh.... my problem size isn't multiple of 64

- 375 GFlops/s in single precision and 75 GFlops/s in double precision

xGEMM performance of CUBLAS-2.3 on GTX280

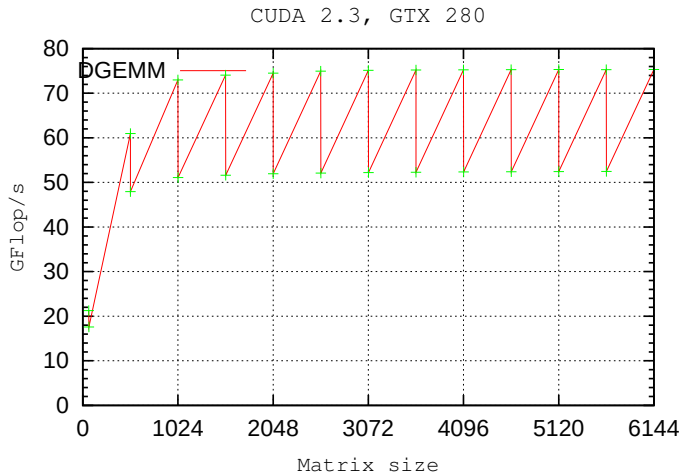
Oh.... my problem size isn't multiple of 64

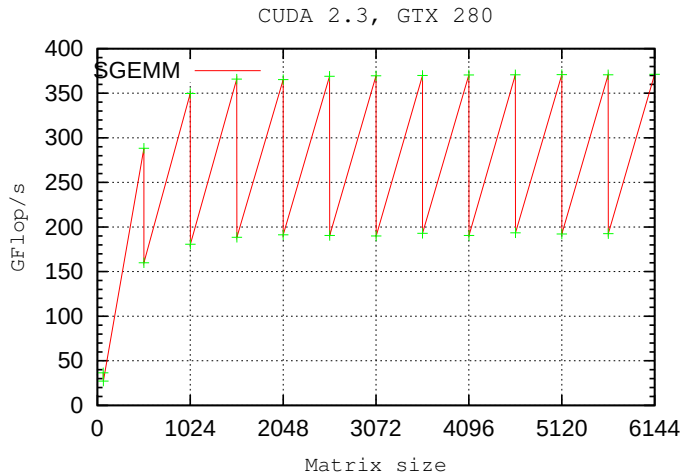
- 375 GFlops/s in single precision and 75 GFlops/s in double precision
- I want to multiply $1025 \times 1025 \times 1025$ matrix. Forum

xGEMM performance of CUBLAS-2.3 on GTX280

Oh.... my problem size isn't multiple of 64

- 375 GFlops/s in single precision and 75 GFlops/s in double precision
- I want to multiply $1025 \times 1025 \times 1025$ matrix. Forum
-

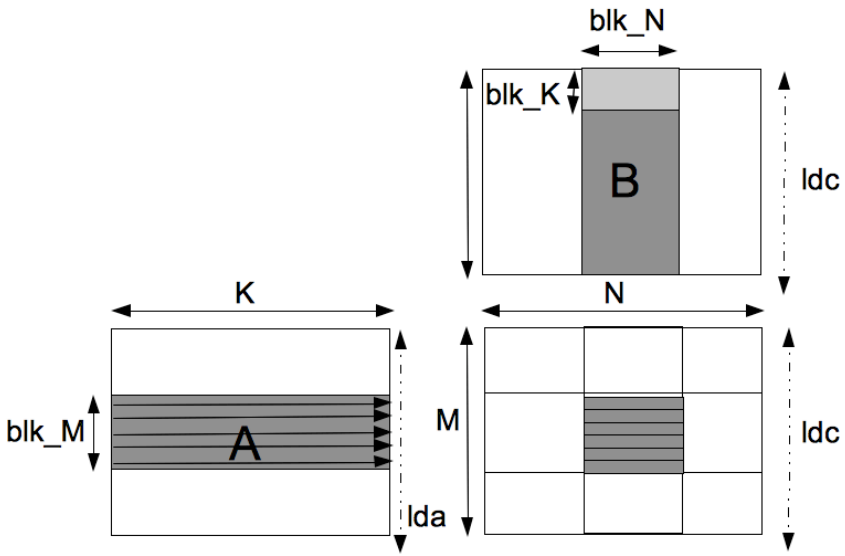




- 75 GFlops/s vs 53 GFlop/s in double precision
- 370 GFlops/s vs 190 GFlops/s in Single Precision

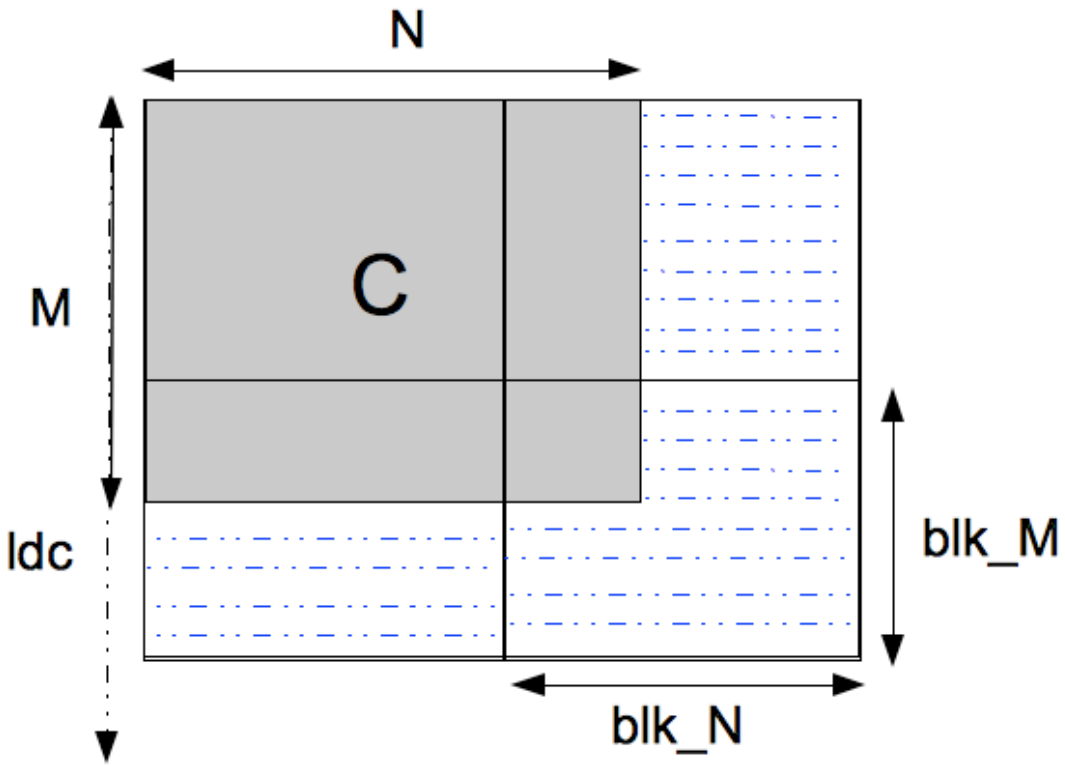
xGEMM Kernel GTX280, Tesla C1060

from Volkov & Demmel



- A thread block computes a block of matrix C
- Each thread computes a row of the block submatrix of C
- Part of matrix B is loaded into shared memory and computations are done in terms of **axpy**

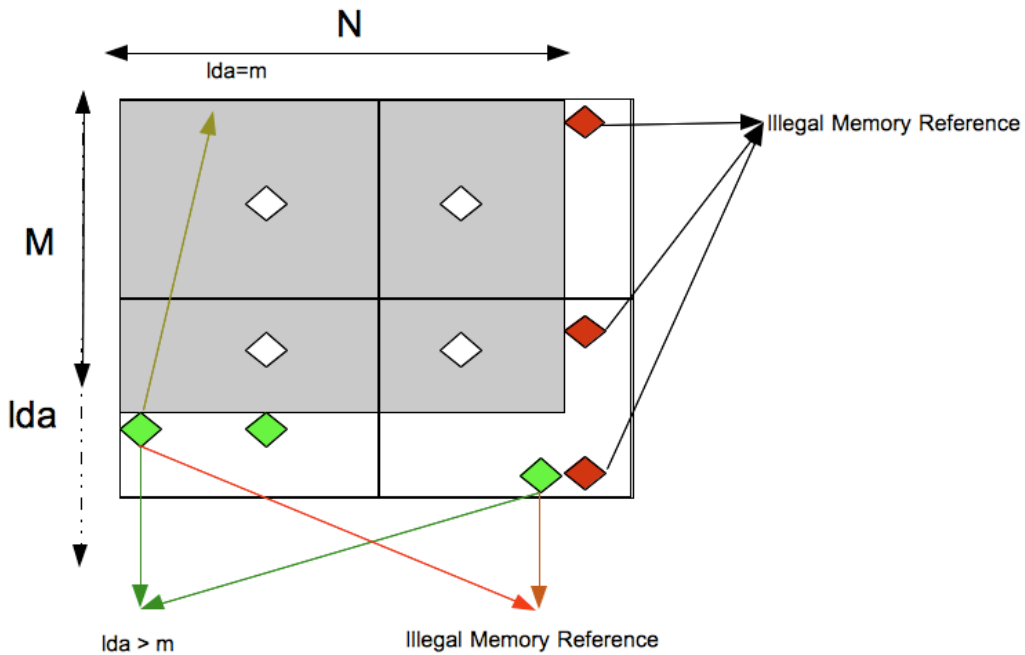
Fringe Elements in xGEMM Kernel GTX280, Tesla C1060



- Compute anyway, discard the results if not needed.

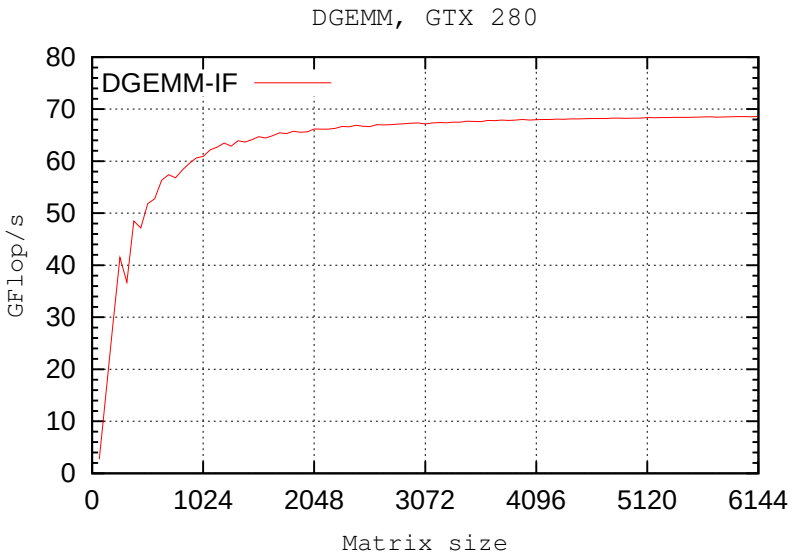
Potential Illegal Memory Reference in xGEMM Kernel

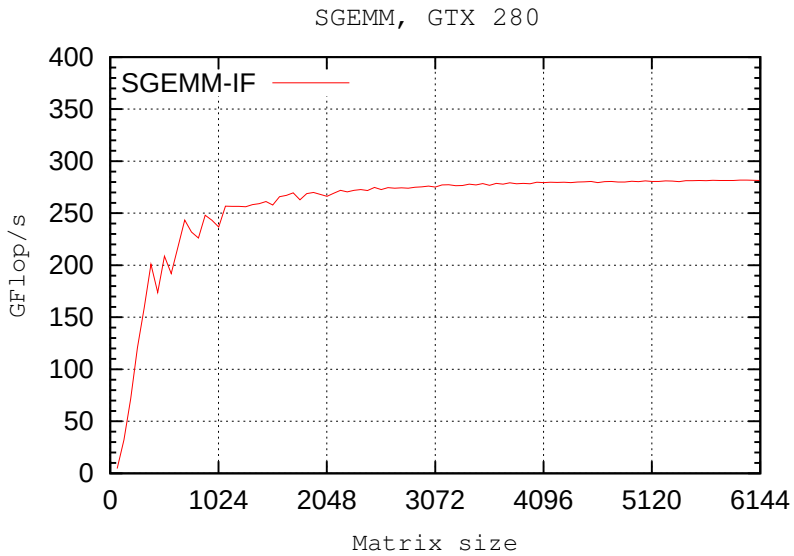
if we don't control memory reference



- Solution: Use **if** statement before any memory access.

xGEMM Kernel with if statement in the inner loop

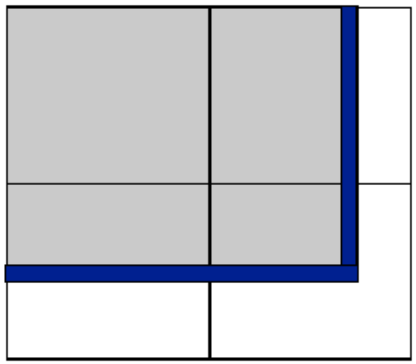
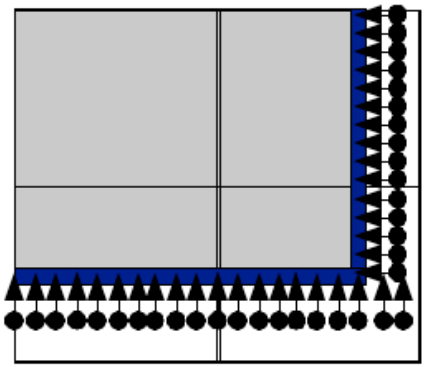
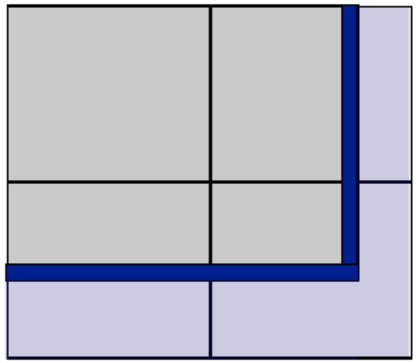




- 75 GFlops/s vs 53GFlop/s (CUBLAS-2.3) vs 68 GFlops/s (if) in double precision
- 370 GFlops/s vs 190GFlops/s (CUBLAS-2.3) vs 280 GFlops/s (if) in Single Precision

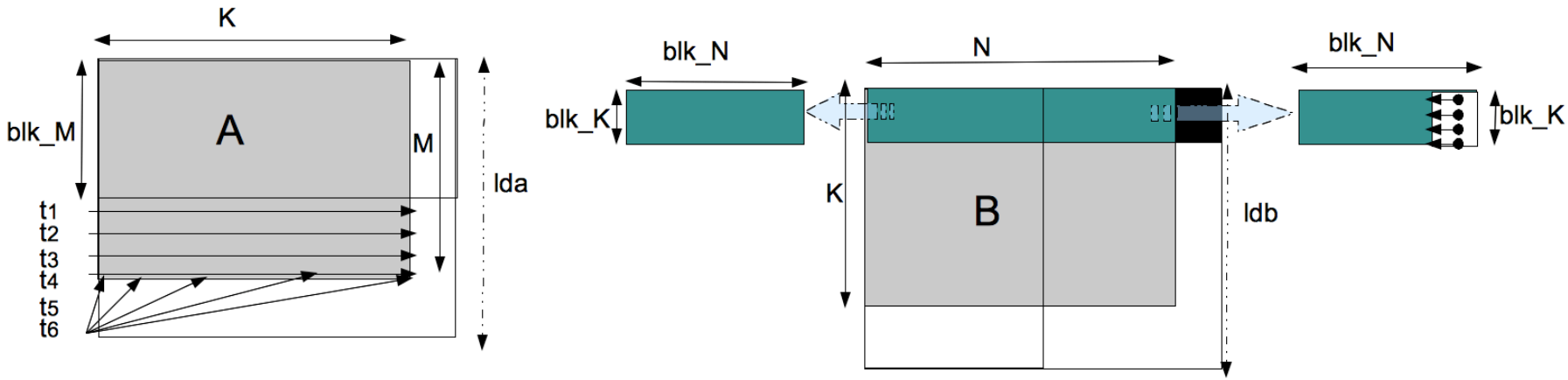
Pointer Redirecting Approach for xGEMM Kernel

in GTX280, TeslaC1060

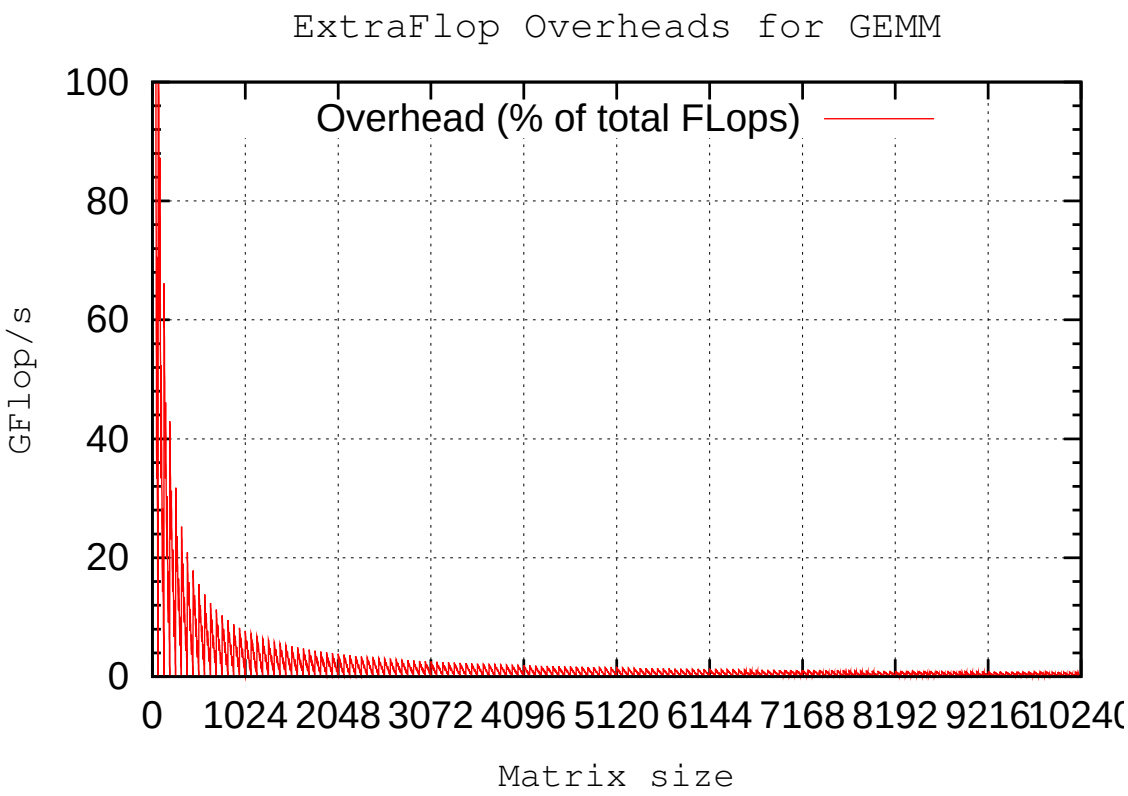
- No assumption about the leading dimension of the matrix.
- Access closest valid element in the matrix in terms of euclidean distance

Pointer Redirecting Approach for xGEMM Kernel



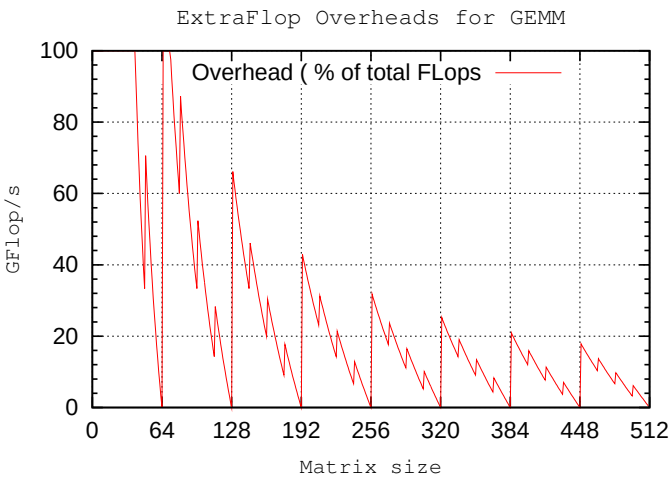
- Accessing matrix A is straightforward.
- Accessing matrix B is done such a way that all the accesses to global memory are coalesced

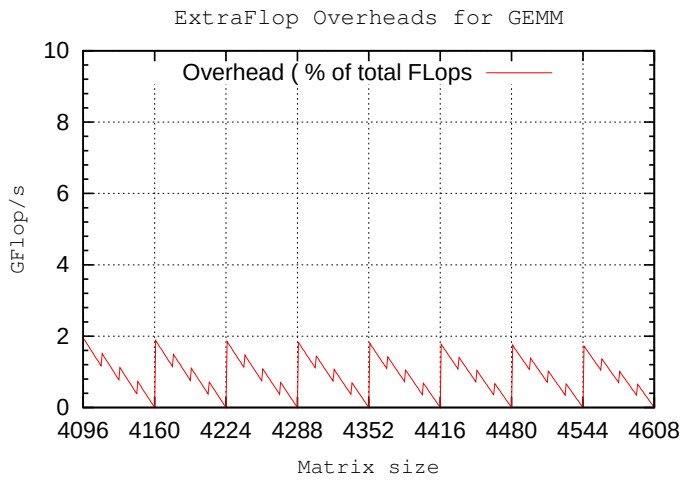
Overhead



- Overhead scaled to 100 for visibility

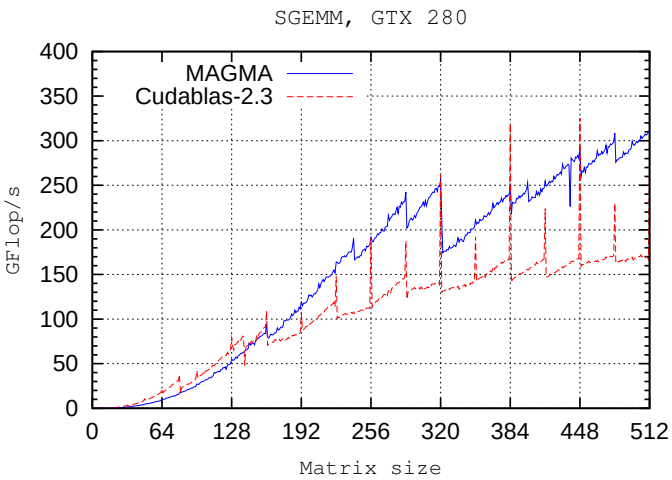
Overhead : Zoomed

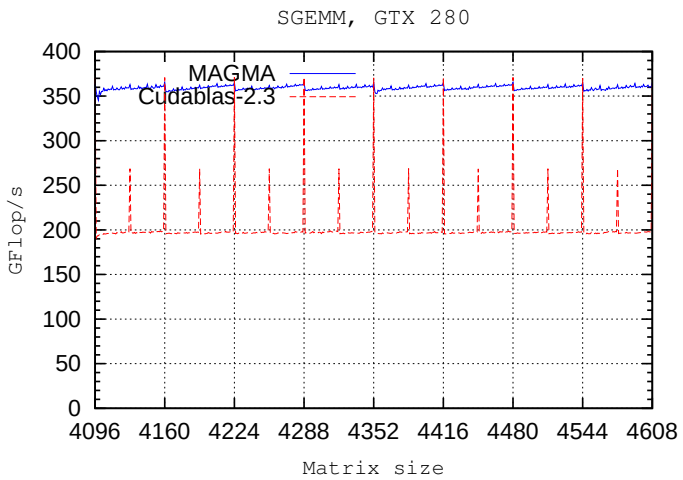




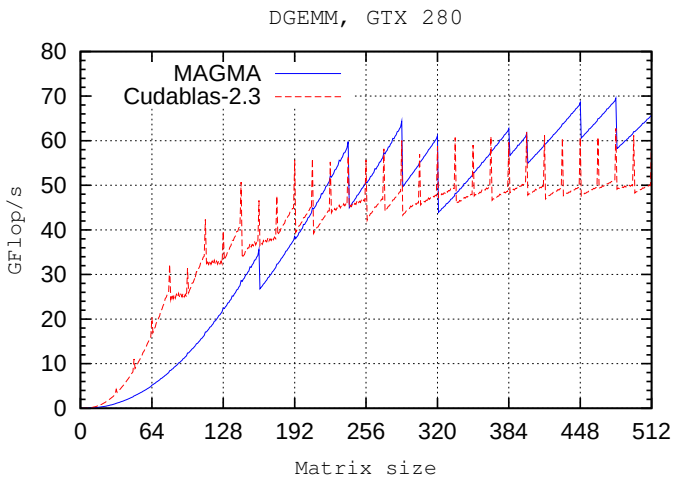
- Decent overhead in sufficient large dimension.

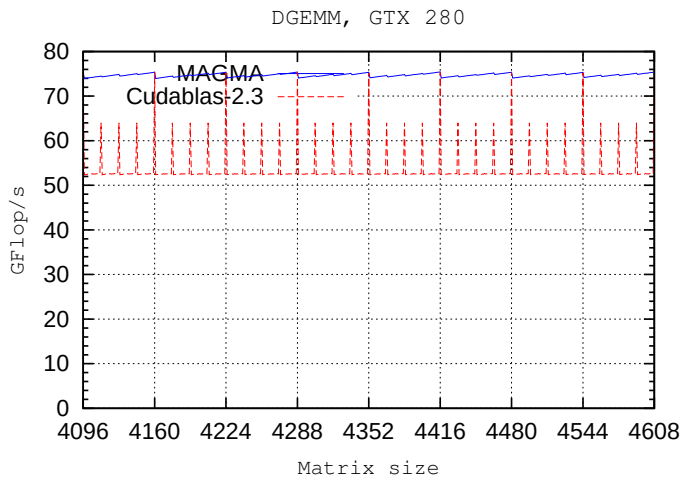
sGEMM


SGEMM, GTX 280
MAGMA
Cudablas-2.3
GFlop/s
Matrix size
0
50
100
150
200
250
300
350
400
0
64
128
192
256
320
384
448
512


SGEMM, GTX 280
MAGMA
Cudablas-2.3
GFlop/s
Matrix size
0
50
100
150
200
250
300
350
400
4096
4160
4224
4288
4352
4416
4480
4544
4608

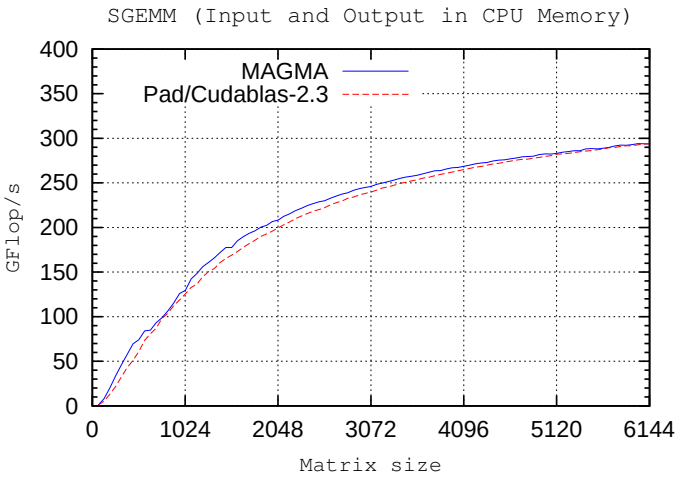
dGEMM

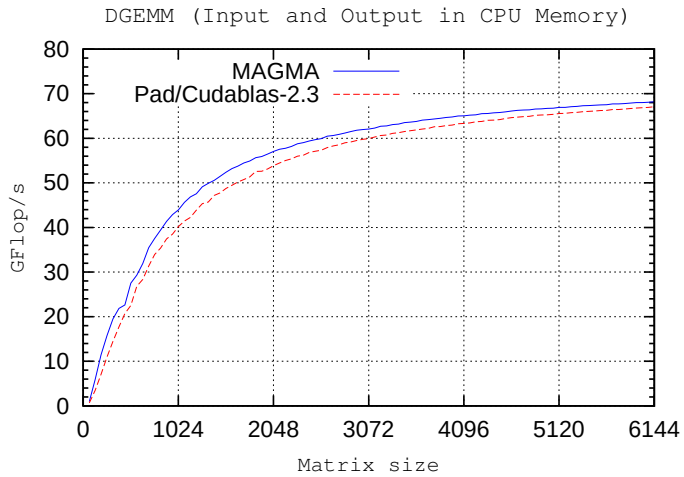

DGEMM, GTX 280
MAGMA
Cudablas-2.3
GFlop/s
Matrix size
0
10
20
30
40
50
60
70
80
0
64
128
192
256
320
384
448
512


DGEMM, GTX 280
MAGMA
Cudablas-2.3
GFlop/s
Matrix size
0
10
20
30
40
50
60
70
80
4096
4160
4224
4288
4352
4416
4480
4544
4608

Pointer Redirecting vs Padding in xGEMM

when input and output data is in CPU Memory





- performance is better
- requires less amount of memory
- think about the matrix is already in GPU

QR factorization

Matrix Size	CUBLAS-2.3	MAGMABLAS
1001	47.65	46.01
2001	109.69	110.11
3001	142.15	172.66
4001	154.88	206.34
5001	166.79	226.43
6001	169.03	224.23
7001	175.45	246.75
8001	177.13	251.73
9001	179.11	269.99

Table: Performance comparison between MAGMA BLAS with pointer-redirecting and CUBLAS in one sided QR factorization,

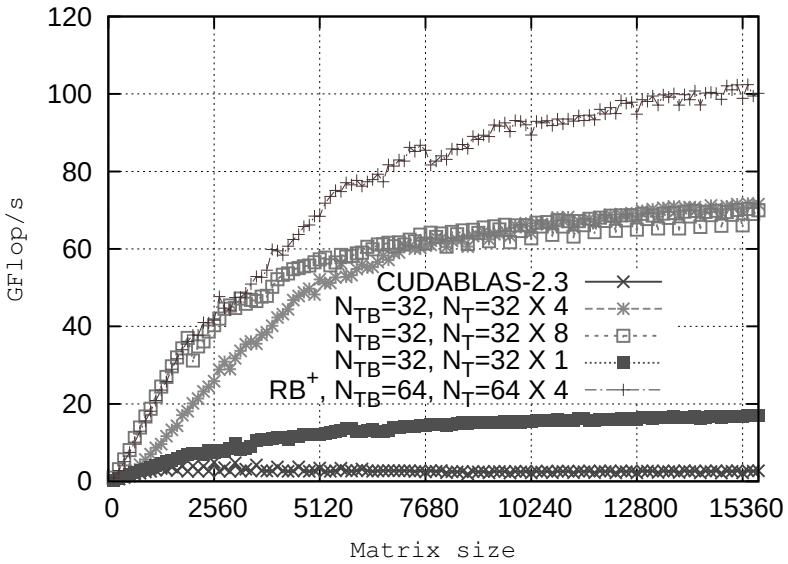
sSYMV

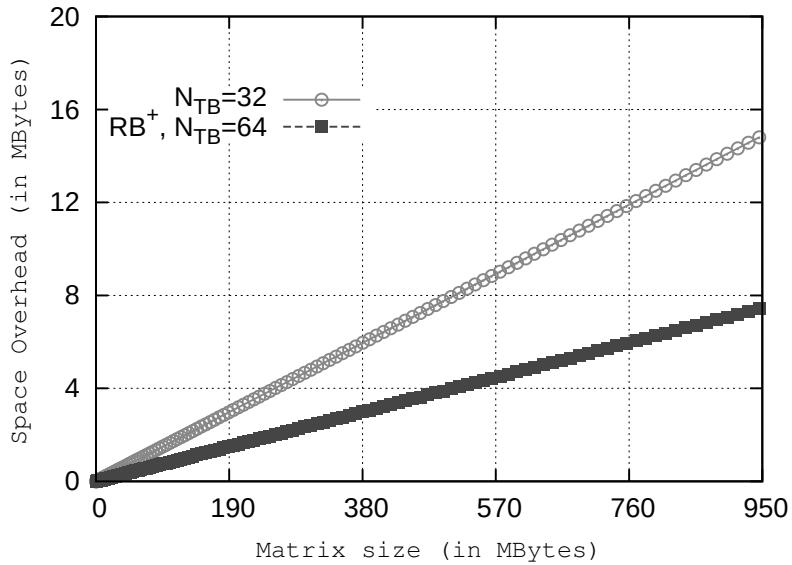
$$y := \alpha Ax + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y,$$

where A is an N by N symmetric matrix, x and y are vectors, and α and β are scalars.

- Needed for tri-diagonalization
- CUBLAS 2.3 gets upto 5 GFlops/s in GTX280
- MAGMABLAS 0.2 gets upto 45 GFlops/s in GTX280
- New algorithm in upcoming MAGMA1.0 gets upto 102 GFlops/s in GTX280
  1. Requires workspace for better performance
  2. Access half of the matrix A only once
  3. Auo-tuned kernel
  4. Uses recursive blocking to work with bigger block size
  5. Reshape thread blocks inside a single kernel

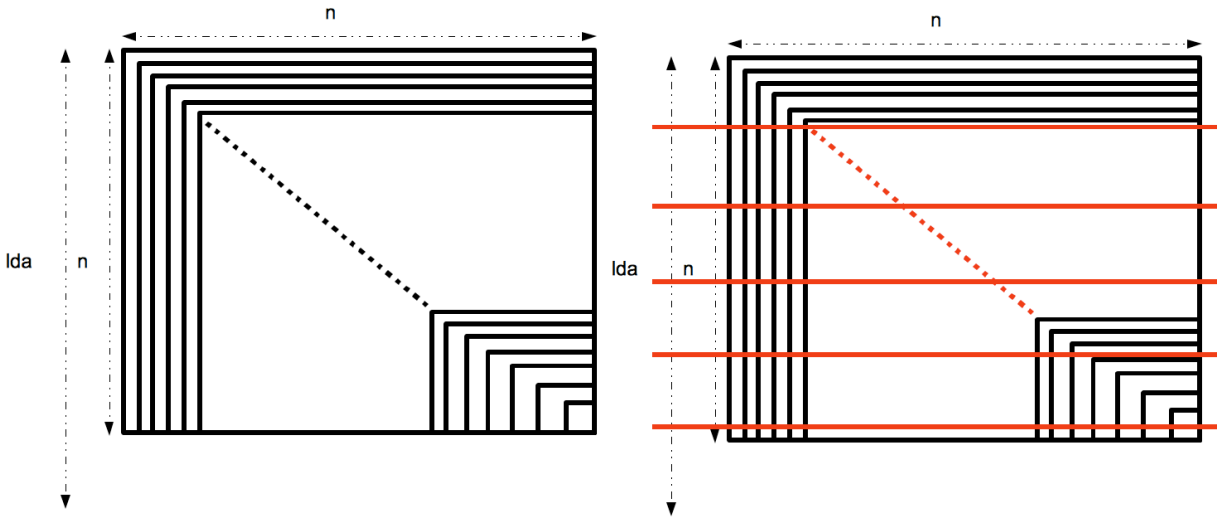
sSYMV





- Space overhead 0.39% of the matrix size (irrespective of leading dimension)
- Generic gets to 95 GFlops/s VS 102 GFlops/s in good dimension.

sSYMV



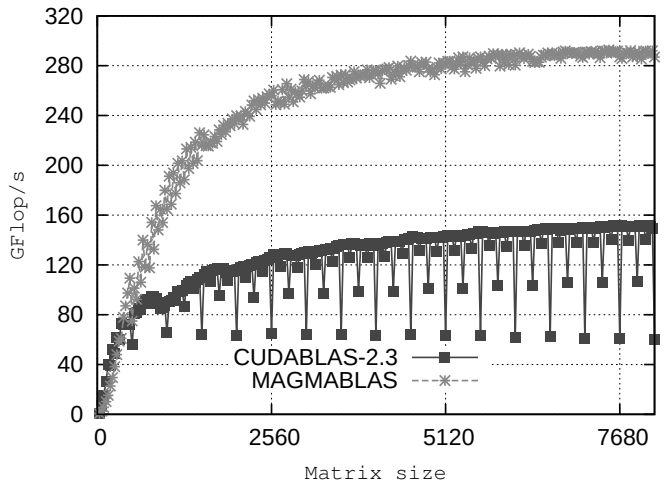
- Leading Dimension of the matrix should be multiple of 16
- Tri-diagonalization requires N SYMV where the starting address of each matrix is A[i,i] for i =1 to N .
- Pointer redirecting used at two places:
  1. At the end to support generic case
  2. At the beginning to enforce coalesced memory access

sSYR2K: symmetric rank 2k update

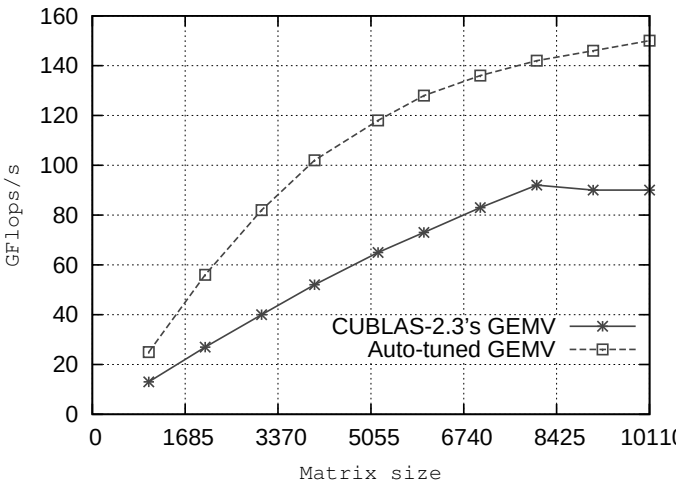
$$C := \alpha AB^T + \alpha BA^T + \beta C \quad \text{or} \quad C := \alpha A^T B + \alpha B^T A + \beta C,$$

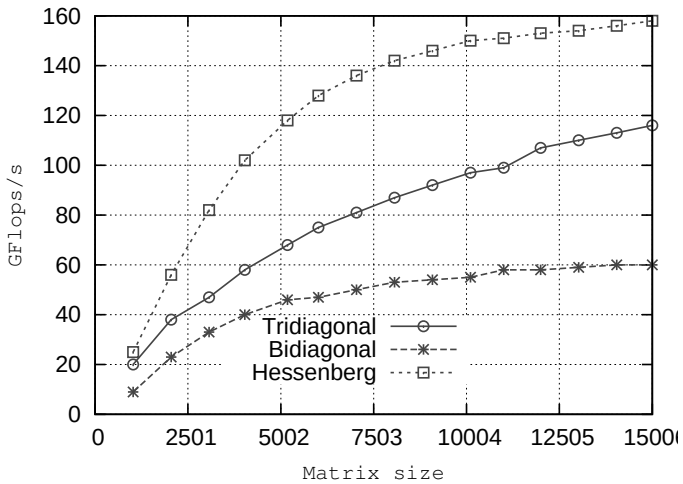
where α and β are scalars, C is an $N \times N$ symmetric matrix and A is an $N \times K$ matrix in the first case and a $K \times N$ matrix in the second case.

- Needed for tri-diagonalization
- Pointer redirecting used for generic dimension
- Circular loop skewing used to avoid partition camping
- Auto-tuned



Two-sided factorization with new kernels





Conclusion

- BLAS can be optimized for generic size
  1. GEMM speedup 49% in single precision and 30% in double precision for generic dimension
  2. All other BLAS routines can be written using same way for arbitrary matrix size, e.g. xSYMV, and xSYR2K
- The higher level routines get faster
  1. QR factorization is speeded up 20% to 50% in single precision using MAGMABLAS as compared to that of CUBLAS.
  2. Two-sided factorization (e.g. hessenberg, bidiagonal. tri-diagonal) gets faster
- Load balancing and autotuning gets easier
- BlahCuda in NVIDIA forum is happy