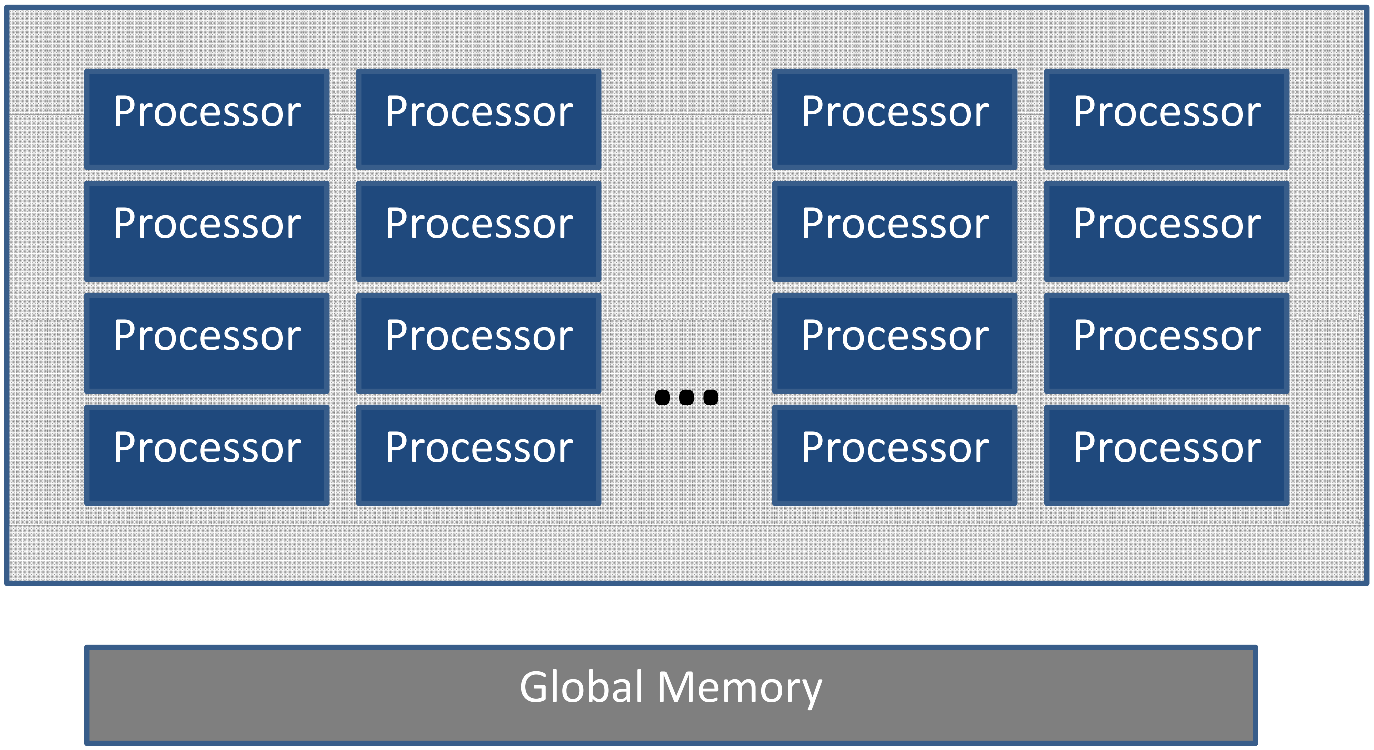
A GPGPU Compiler for Memory Optimization and Parallelism Management

Yi Yang, Ping Xiang, Jingfei Kong, Huiyang Zhou

Department of Electrical and Computer Engineering
North Carolina State University

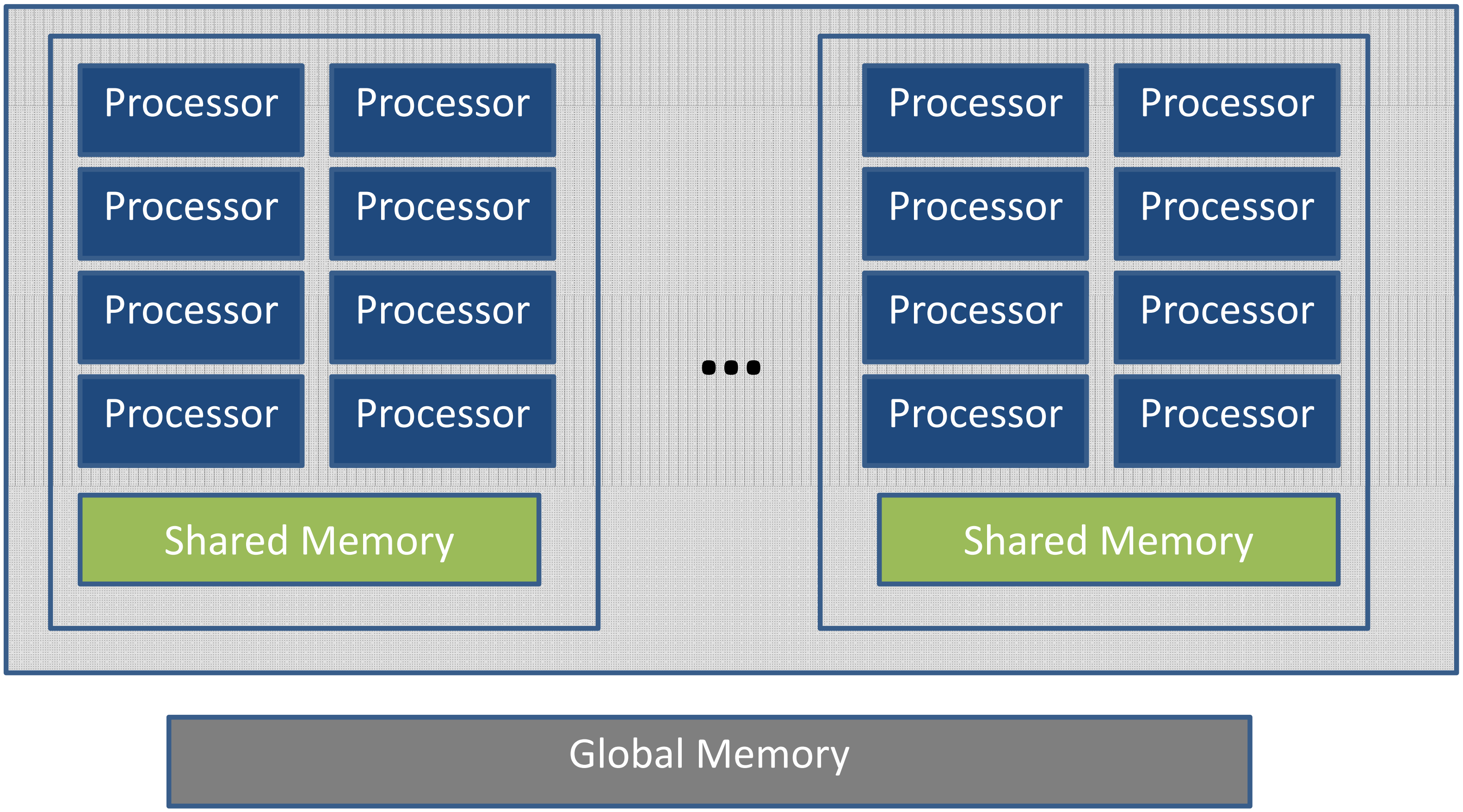
School of EECS
University of Central Florida

A Simplified View of GPU Architecture



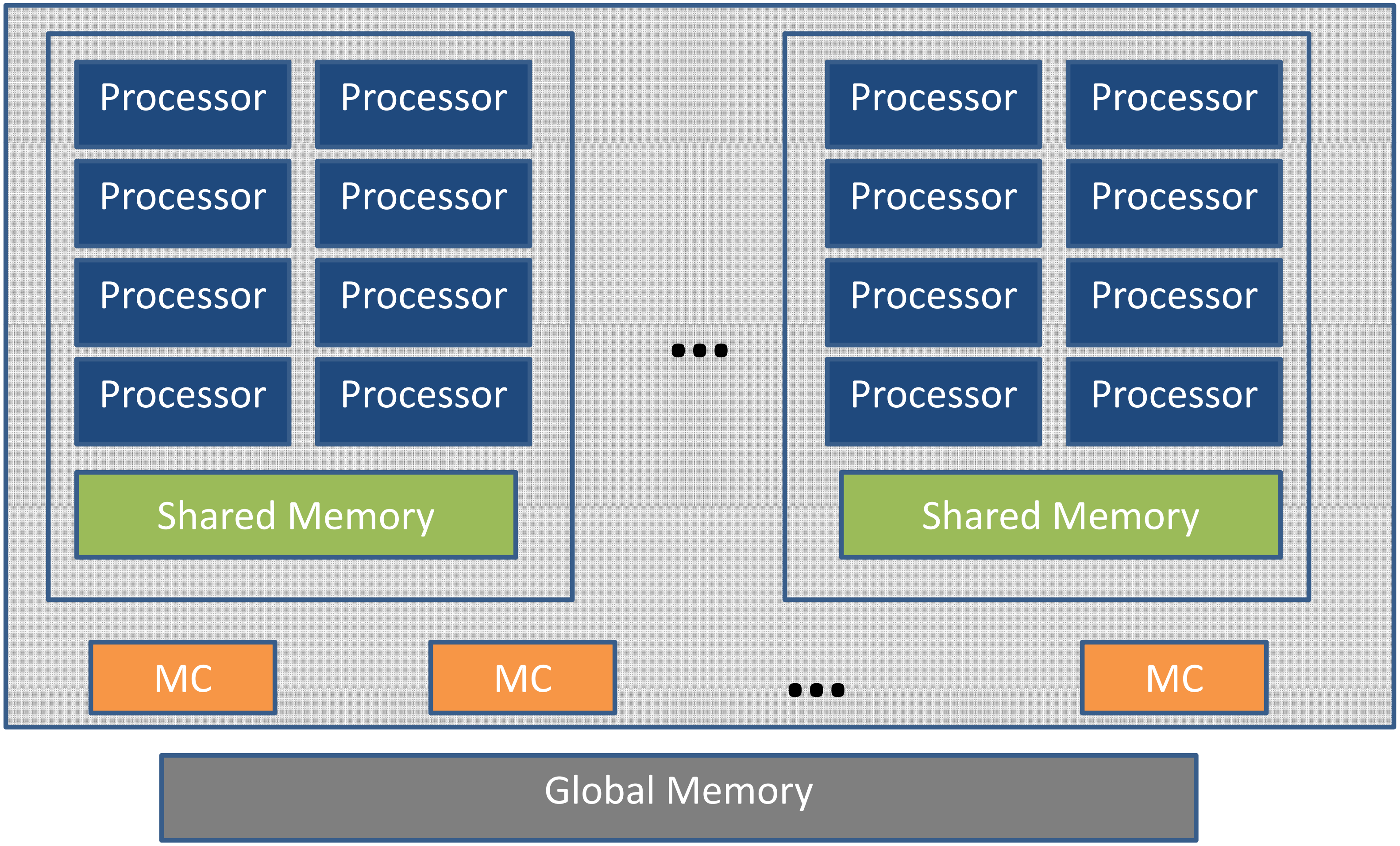
- All processors run the same code, single program multiple data (SPMD)
- Communication among processors is costly.

Understanding GPU Architecture



- Processors are organized in groups, called Streaming Multiprocessors(SM)
- On-chip shared memory, a fast software-managed cache in each SM
- Fast (local) communication among processors in a SM .

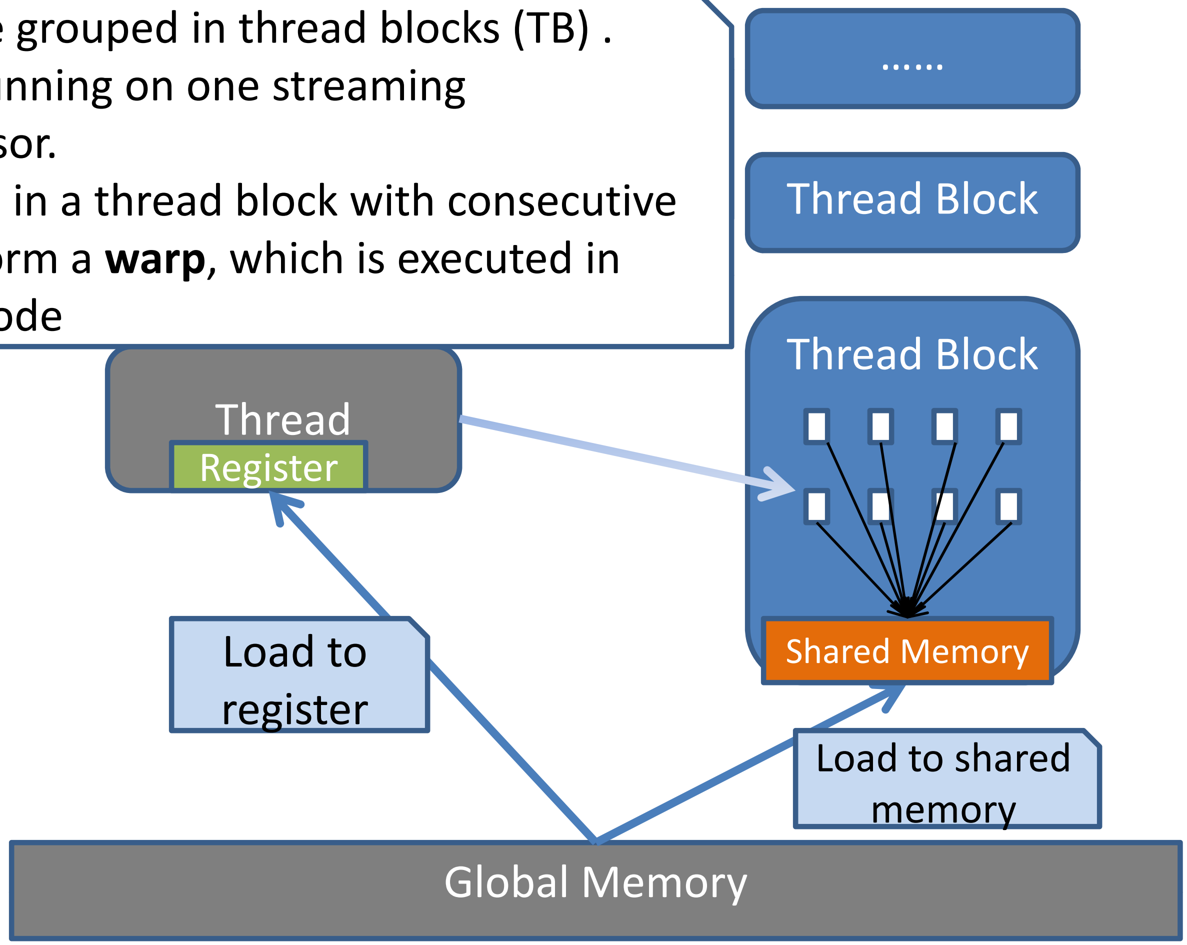
Understanding GPU Architecture



- Several memory controllers (MCs)shared among all the processors
- Memory requests need to be evenly distributed among MCs. Otherwise, conflicts/partition clamping

Thread Execution Model

•Threads are grouped in thread blocks (TB) . Each TB is running on one streaming multiprocessor.
• 32 Threads in a thread block with consecutive thread ids form a **warp**, which is executed in the SIMD mode



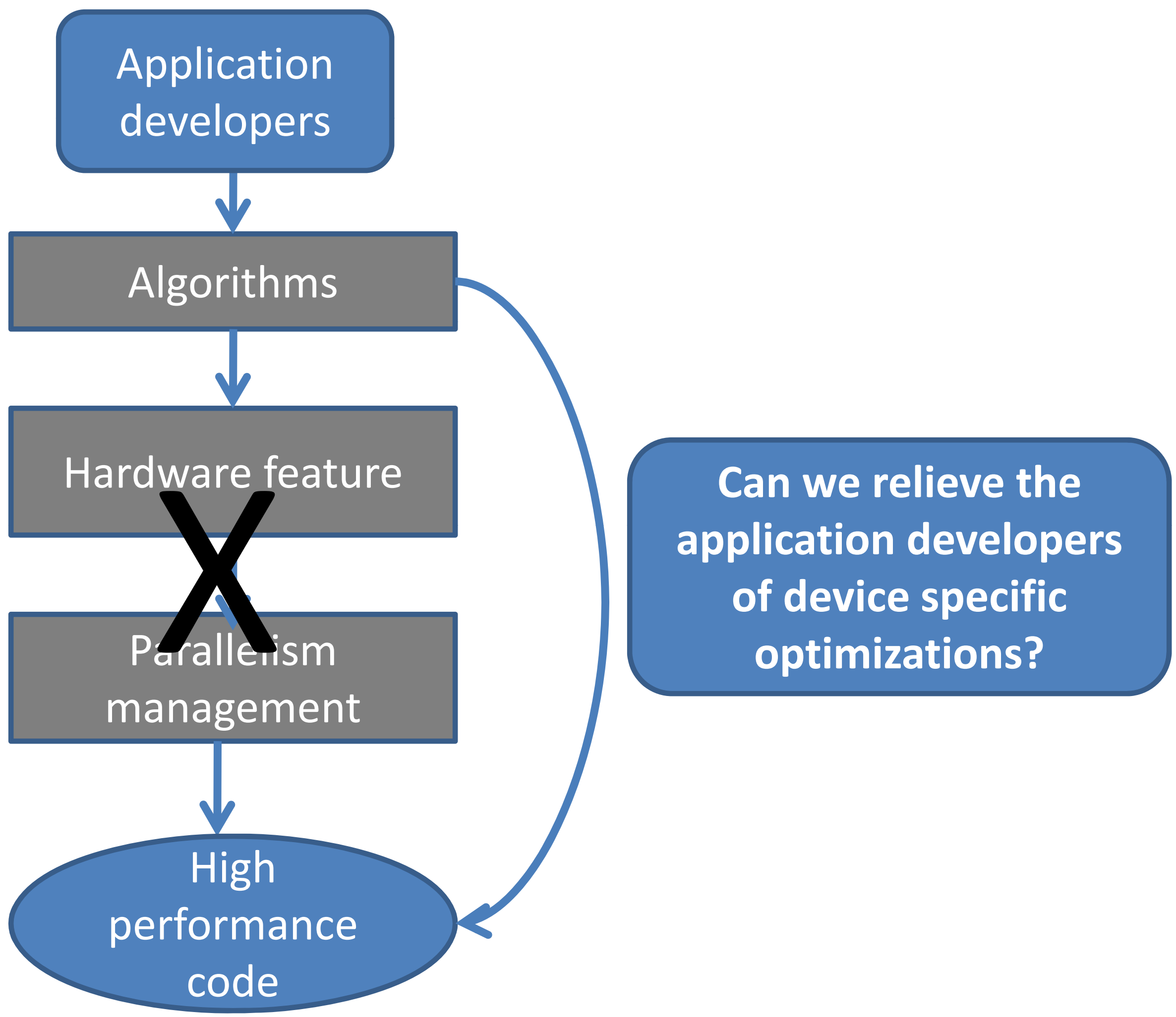
Key to Performance

- Bandwidth of global memory accesses
  - Coalesced global memory accesses
  - Distributed memory accesses among memory controllers/ partitions
- Shared memory
  - Software-managed cache
- Balanced parallelism management: TLP vs. ILP
  - Thread level: register usage, ILP
  - Thread block level: shared memory usage, TLP

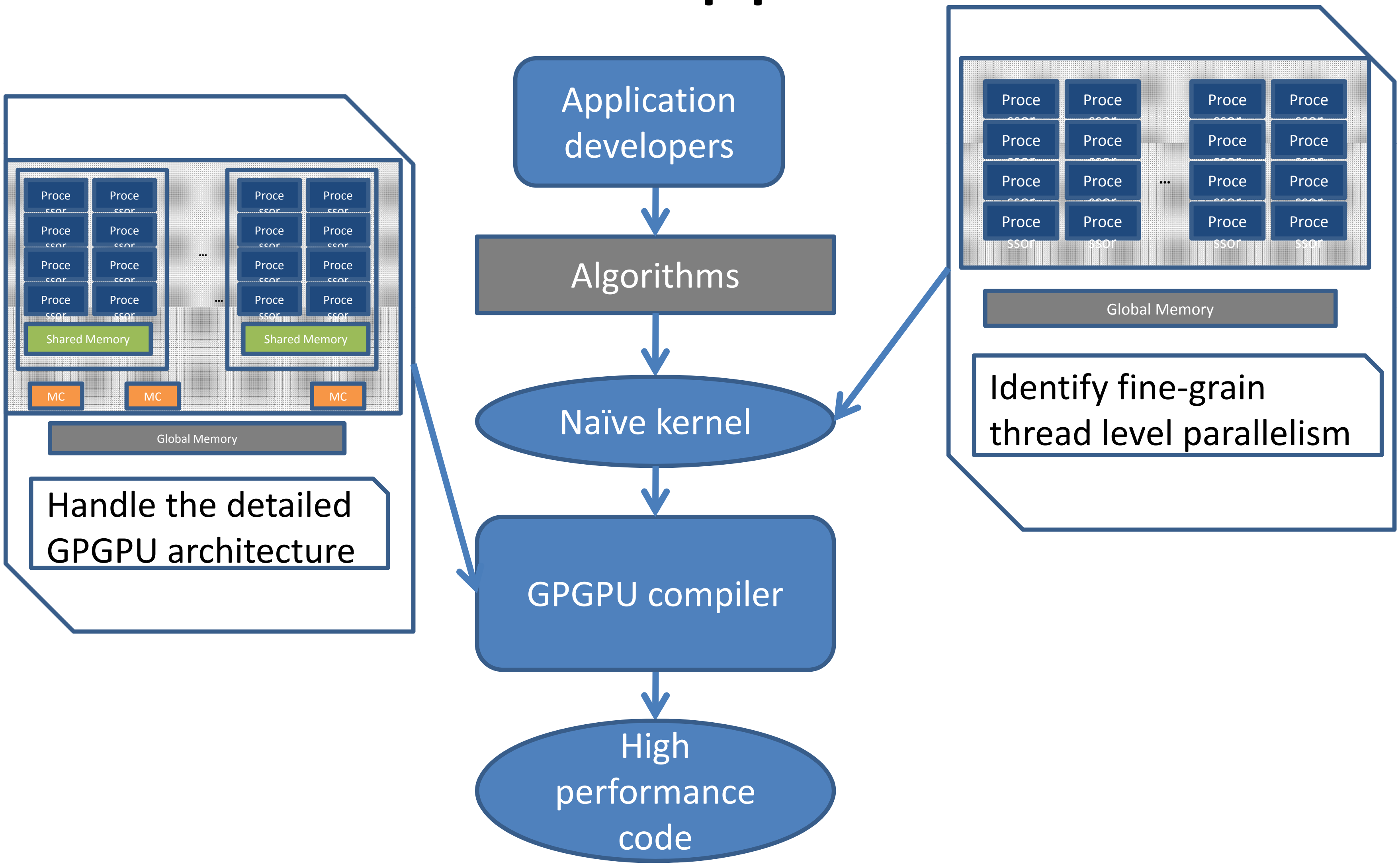
Outline

- Background
- Motivation
- Compiler Framework
  - Memory coalescing
  - Thread (block) merge
- Experimental Results
- Conclusion

Motivation


Application developers
Algorithms
Hardware feature
X
Parallelism management
High performance code
Can we relieve the application developers of device specific optimizations?

Our Approach



Naïve Kernel

- Fine-grain data-level parallelism
- Compute one element/pixel in the output domain
- Example: Matrix multiplication

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

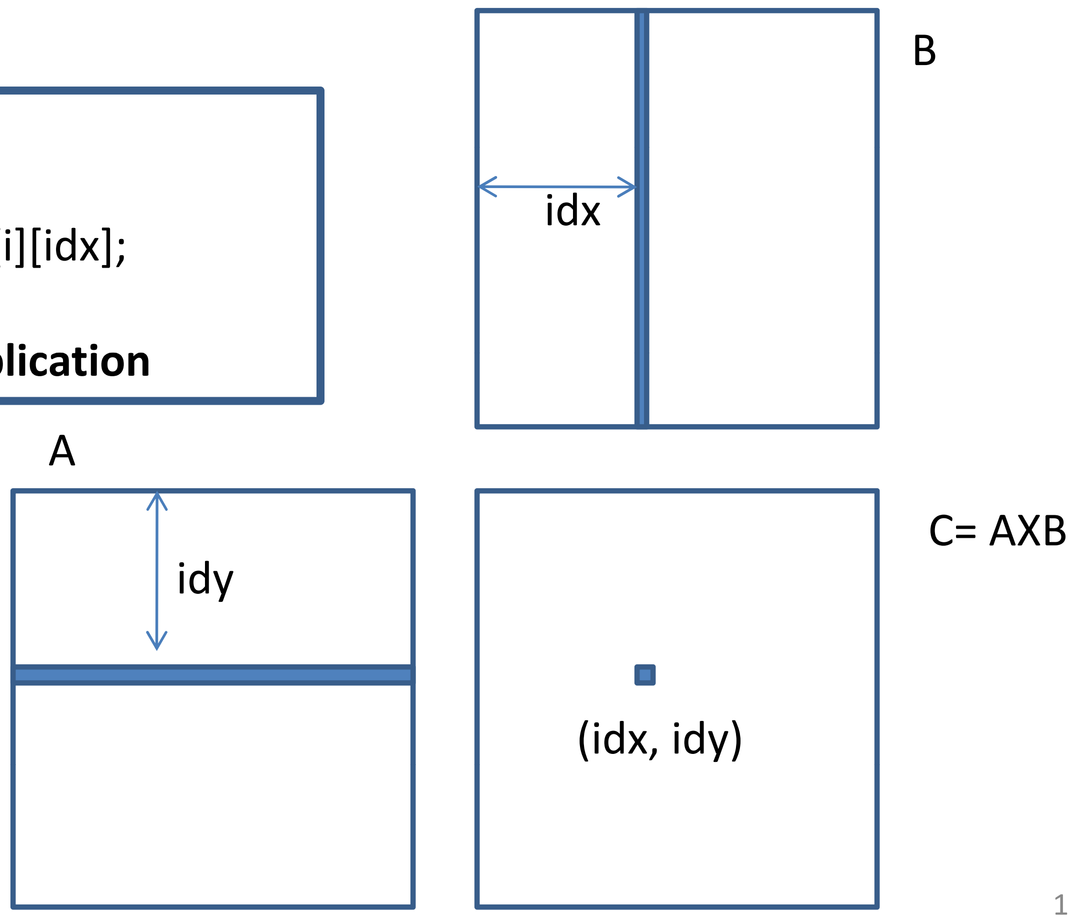
Naïve matrix multiplication

Physical Meaning of the Naïve Kernel

- One thread computes one element at (idx, idy) in the product matrix

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

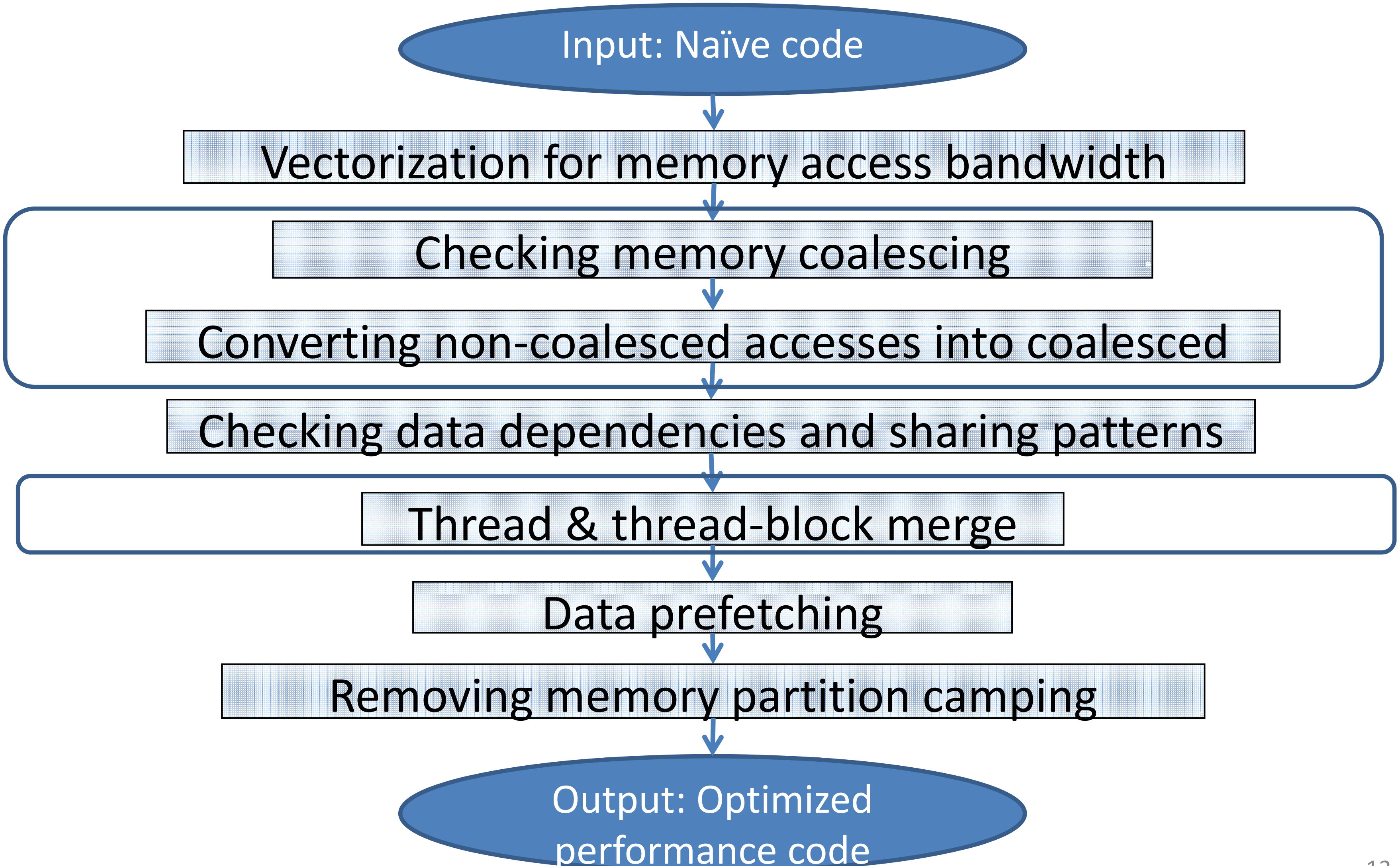
Naïve matrix multiplication



Outline

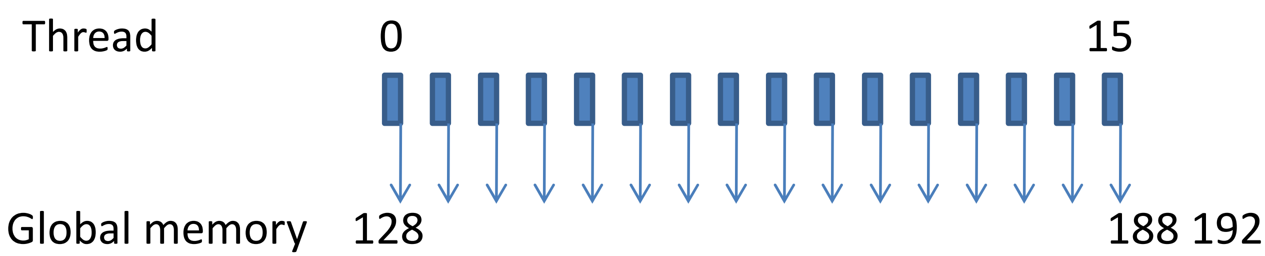
- Background
- Motivation
- Compiler Framework
  - Memory coalescing
  - Thread (block) merge
- Experimental Results
- Conclusion

Compiler Framework

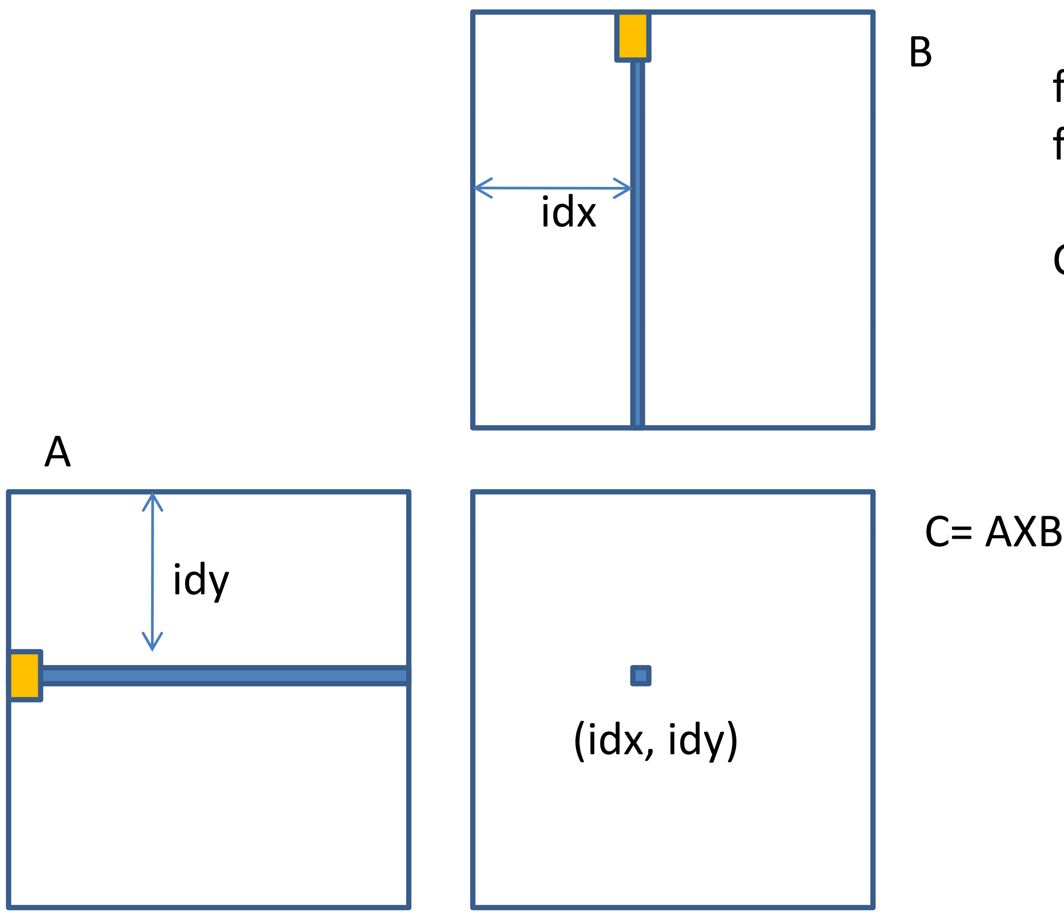


Coalesced Global Memory Access

- Needed by GPU to achieve high memory bandwidth
- Examined at the half-warp granularity
- Requirements for coalesced global memory accesses
  - Aligned:
    - Half of warp threads must access the data with starting address to be a multiple of 64 bytes
  - Sequential (less strict for GTX 280/480):
    - Half of warp threads must access the data sequentially



Checking Memory Coalescing

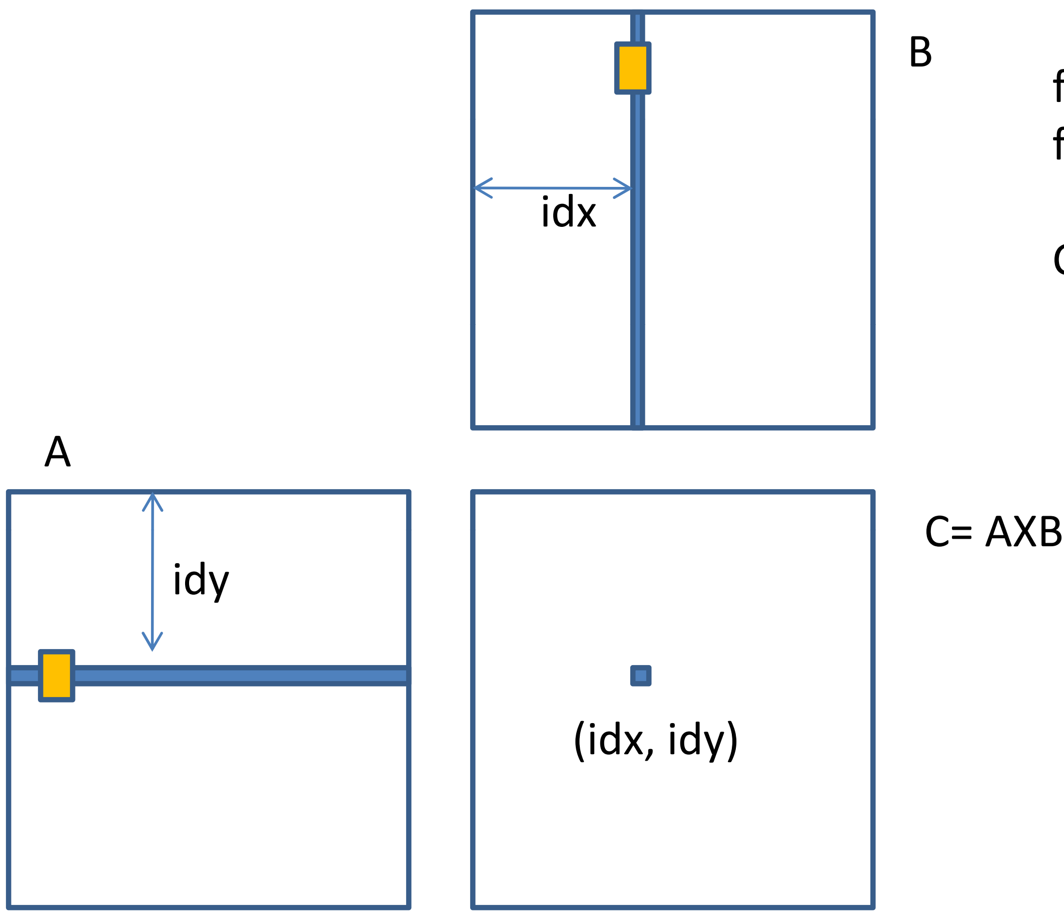



```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication

i = 0

Checking Memory Coalescing

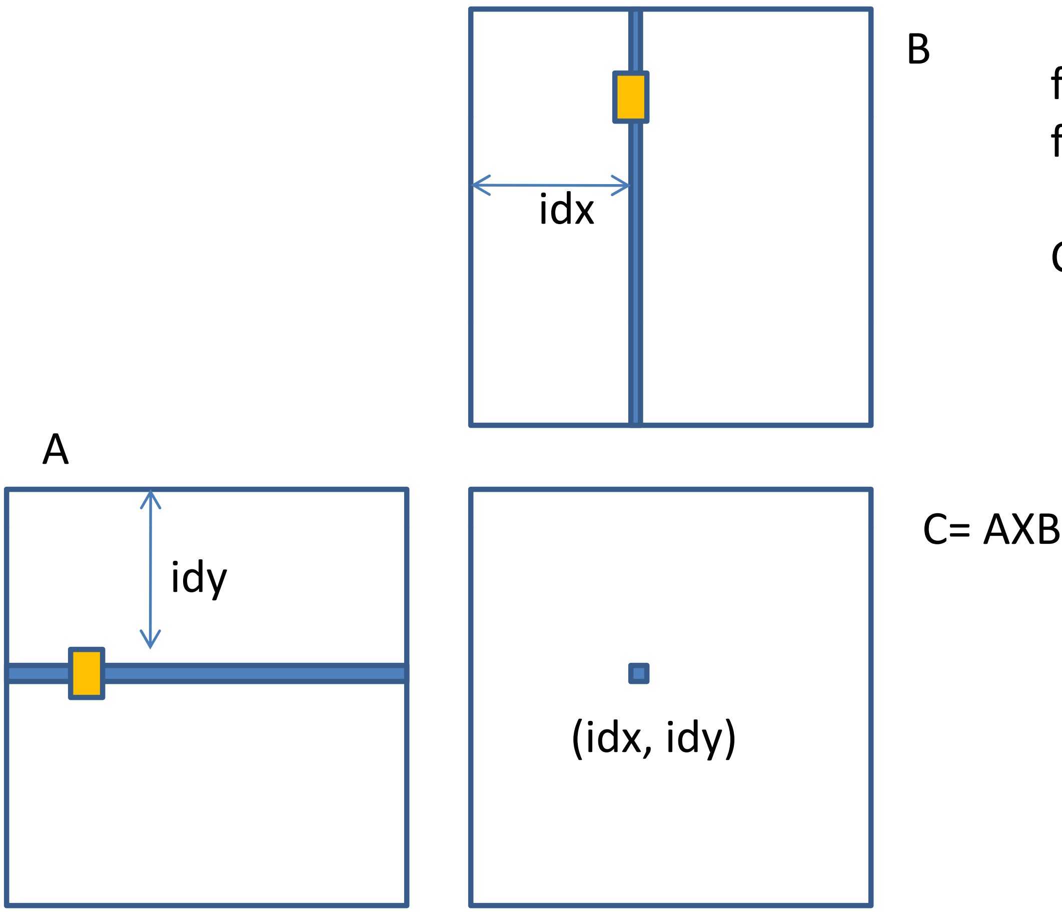



```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication

i = 1

Checking Memory Coalescing

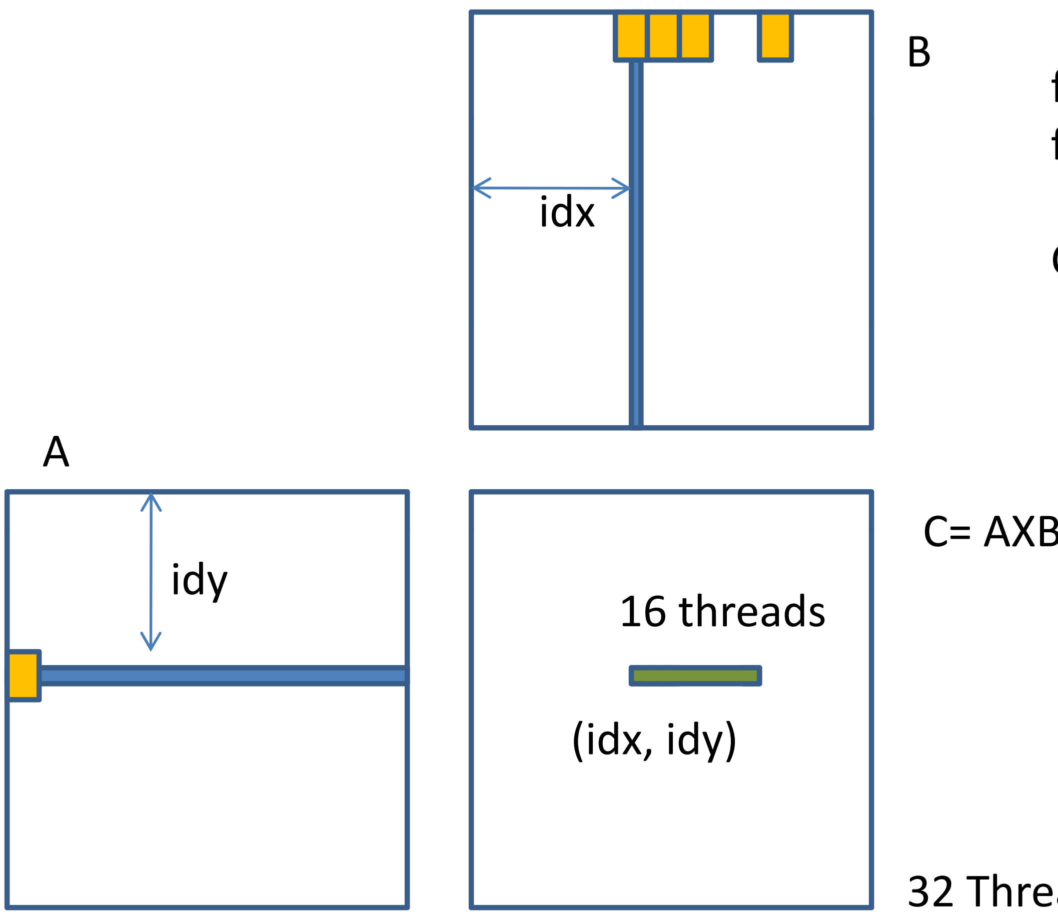



```
float sum = 0;
for (int i=0; i<w; i++)
            sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication

i = 2

Checking Memory Coalescing



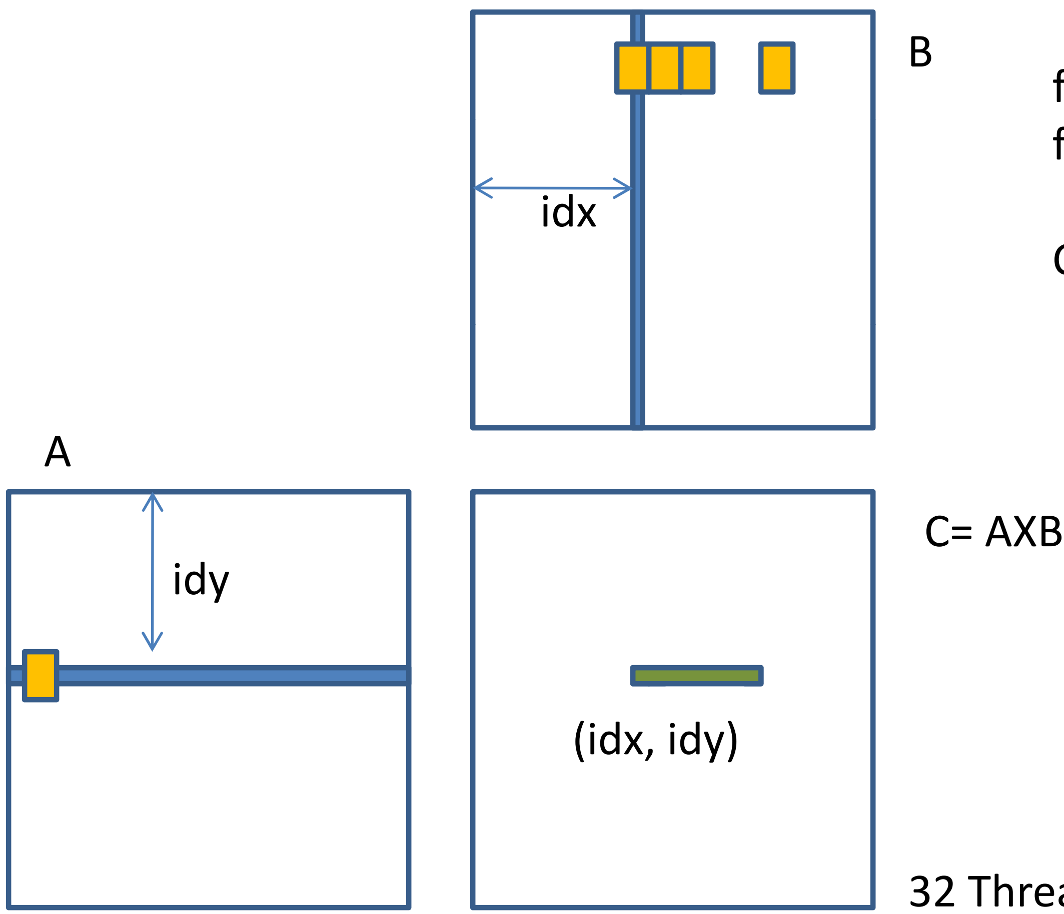

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication

i = 0
All 16 threads access one element A[idy][0].

32 Threads in a thread block with consecutive thread ids form a **warp,** e.g., threads with id (idx, idy), (idx+1, idy),(idx+2, idy),…, (idx+31, idy) assuming idx is a multiple of 32.

Checking Memory Coalescing



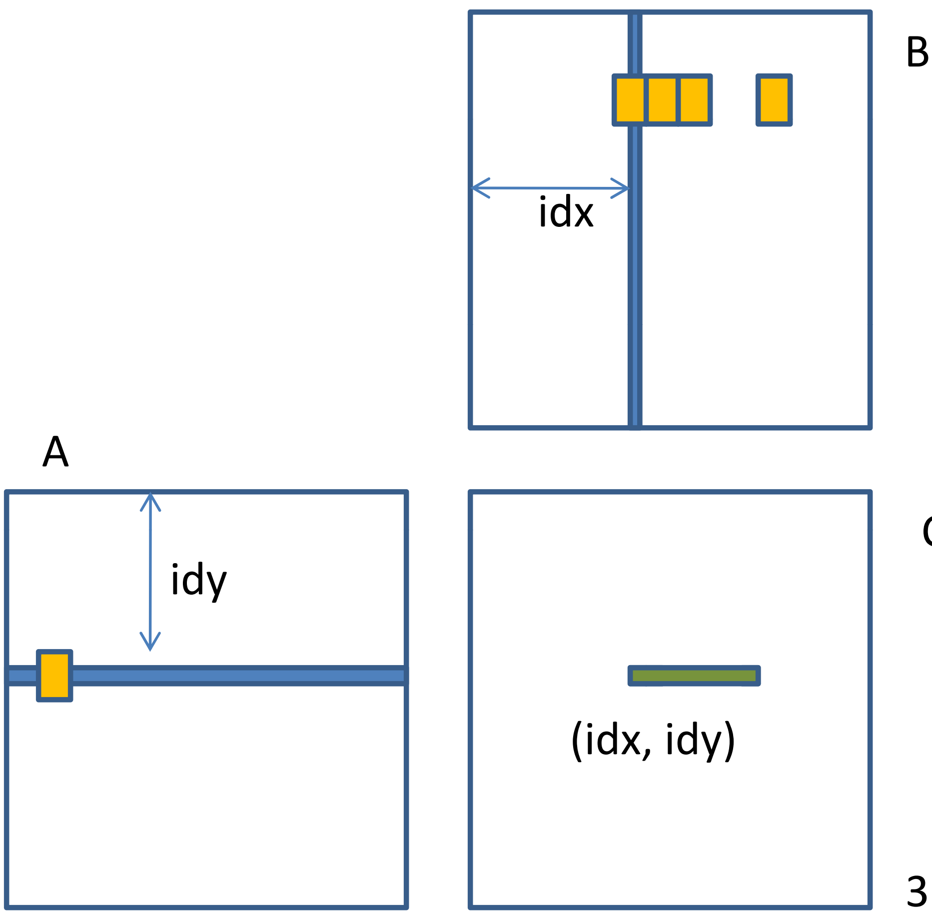

```
float sum = 0;
for (int i=0; i<w; i++)
            sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication

i = 1
All 16 threads access one element A[idy][1].

32 Threads in a thread block with consecutive thread ids form a **warp,** e.g., threads with id (idx, idy), (idx+1, idy),(idx+2, idy),…, (idx+31, idy) assuming idx is a multiple of 32.

Checking Memory Coalescing




```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

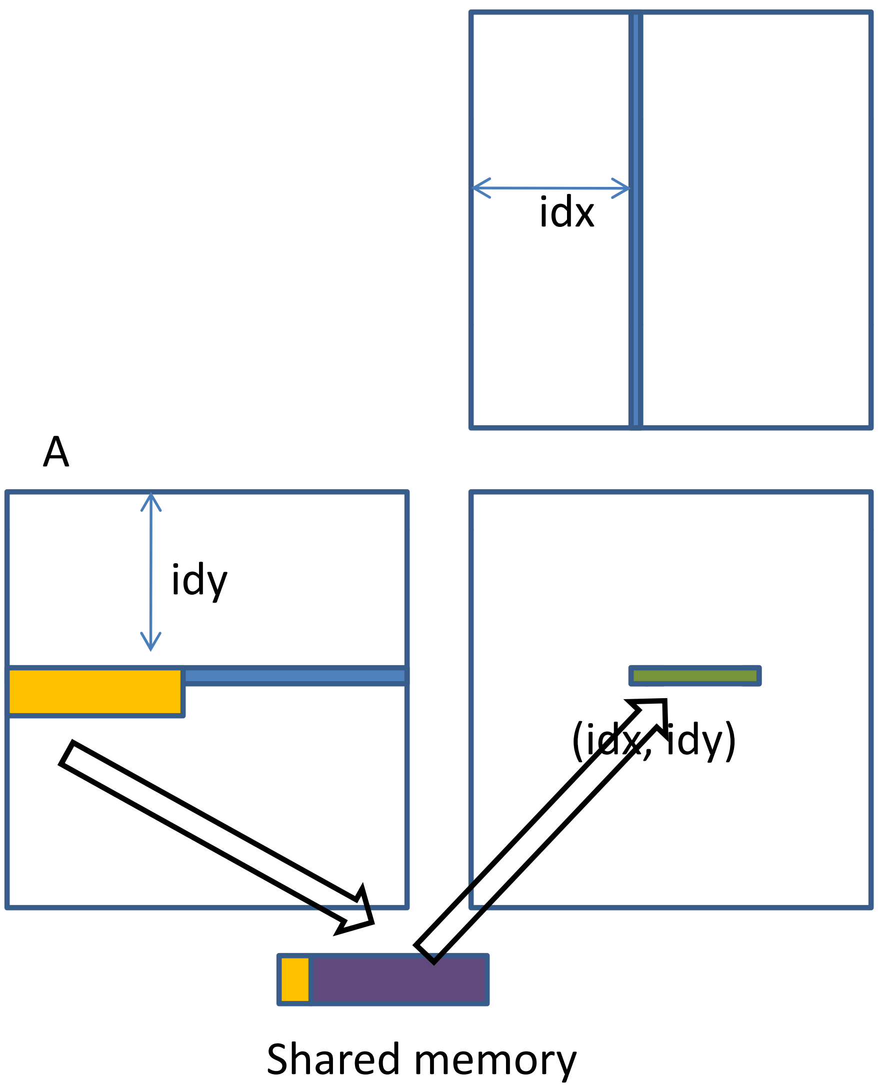
Naïve matrix multiplication

C= AXB

i = 2
All 16 threads access one element A[idy][2].

32 Threads in a thread block with consecutive thread ids form a **warp,** e.g., threads with id (idx, idy), (idx+1, idy),(idx+2, idy),..., (idx+31, idy) assuming idx is a multiple of 32.

Code Conversion for Coalesced Memory Accesses



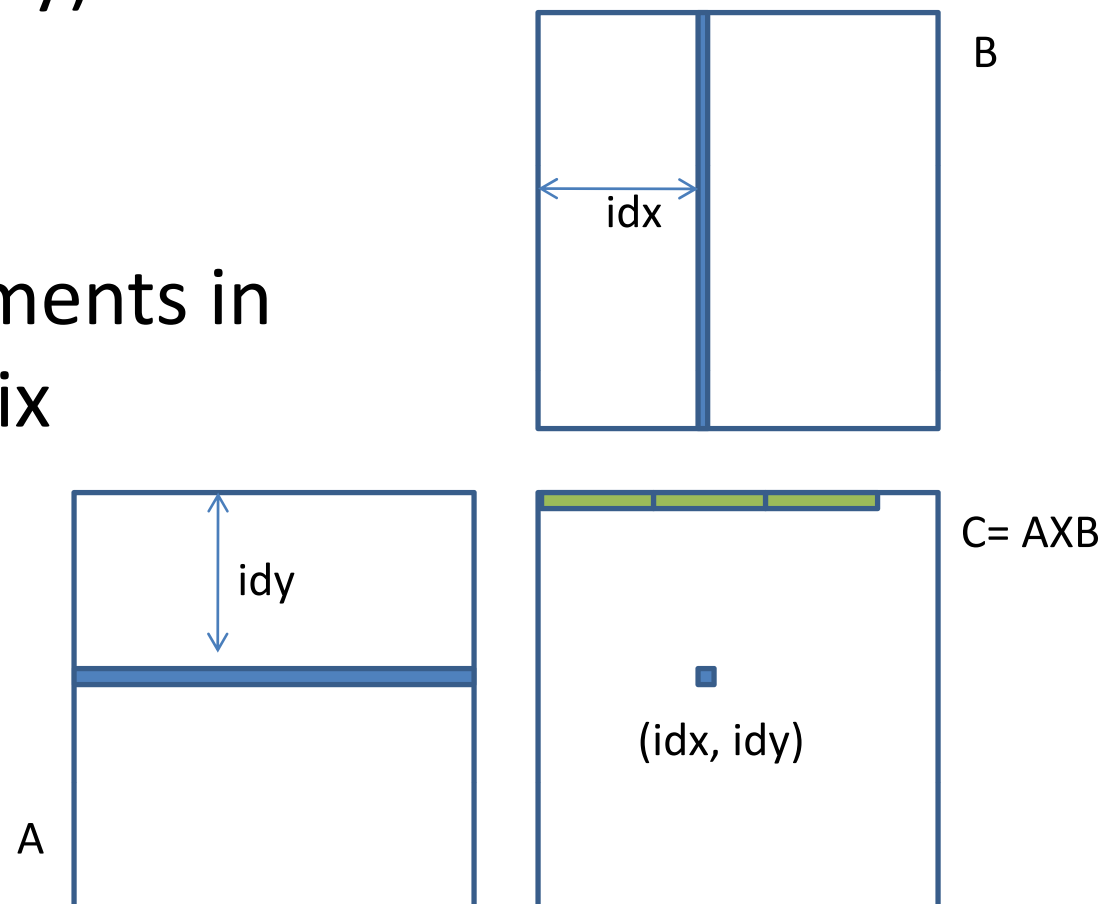

```
for (i=0; i<w; i=(i+16)) {
        __shared__ float shared0[16];
        shared0[(0+tidx)]=A[idy][(i+tidx)];
        __syncthreads();
        for (int k=0; k<16; k=(k+1)) {
            sum+=shared0[(0+k)]*B[(i+k)][idx]);
        }
        __syncthreads();
}
c[idy][idx] = sum;
```

Access data from the shared memory

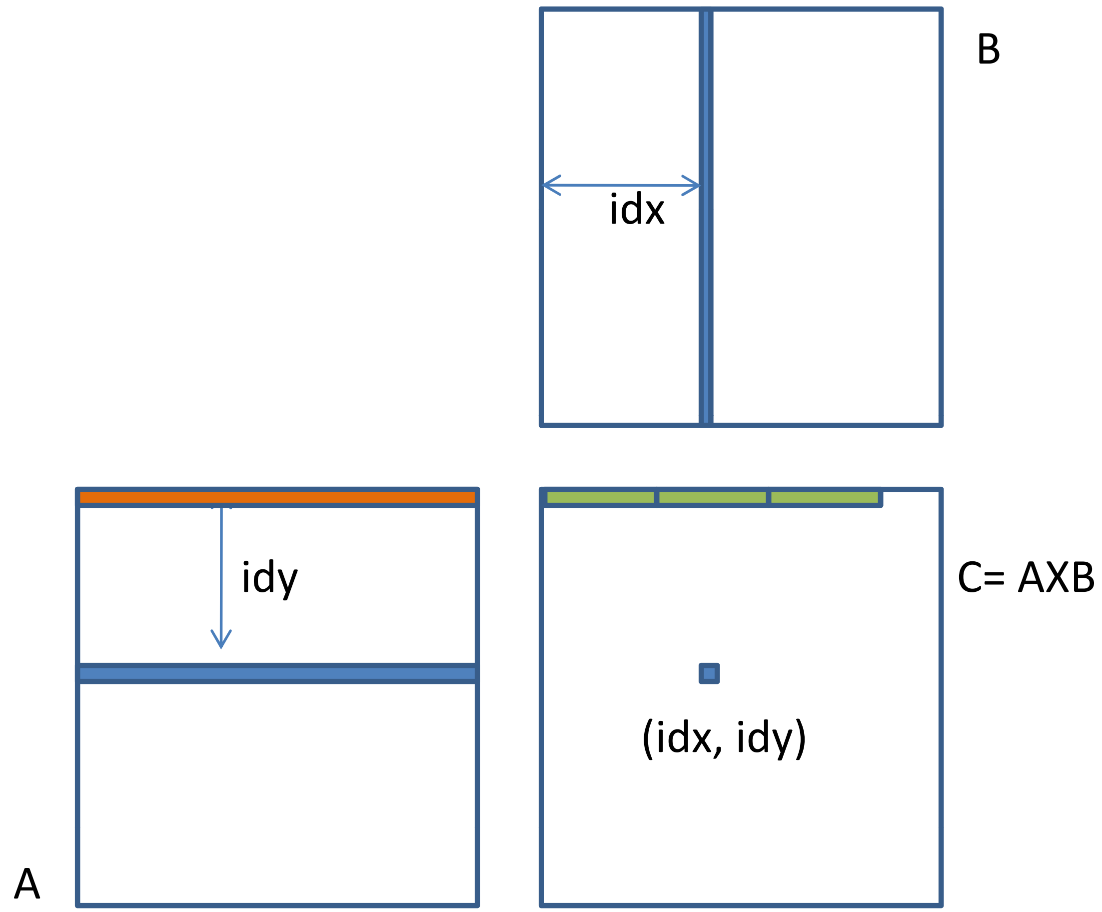
Load global memory into shared memory (coalesced)

Physical Meaning of the Coalesced Kernel

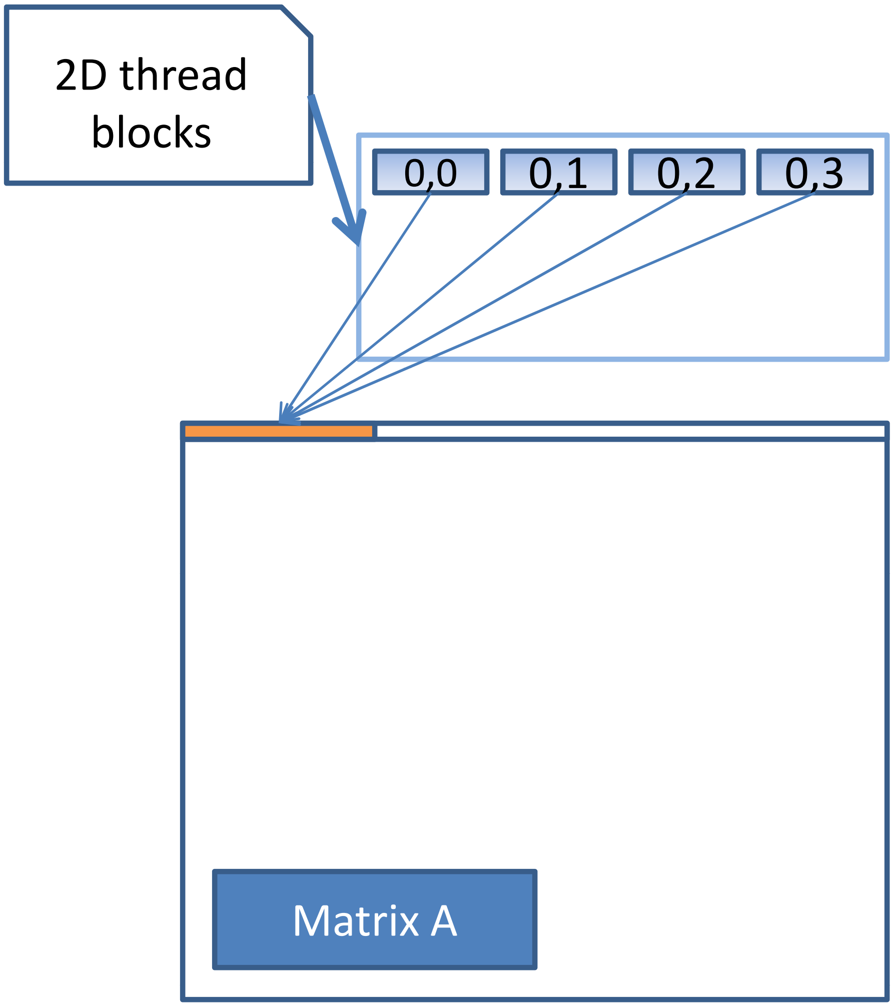
- One thread computes one element at (idx, idy) in the product matrix
- One thread block computes 16 elements in the product matrix
- Tile size: 16x1



Checking Data Dependence and Data Sharing



Detect Data Sharing Among Thread Blocks



for (i=0; i<w; i=(i+**16**))

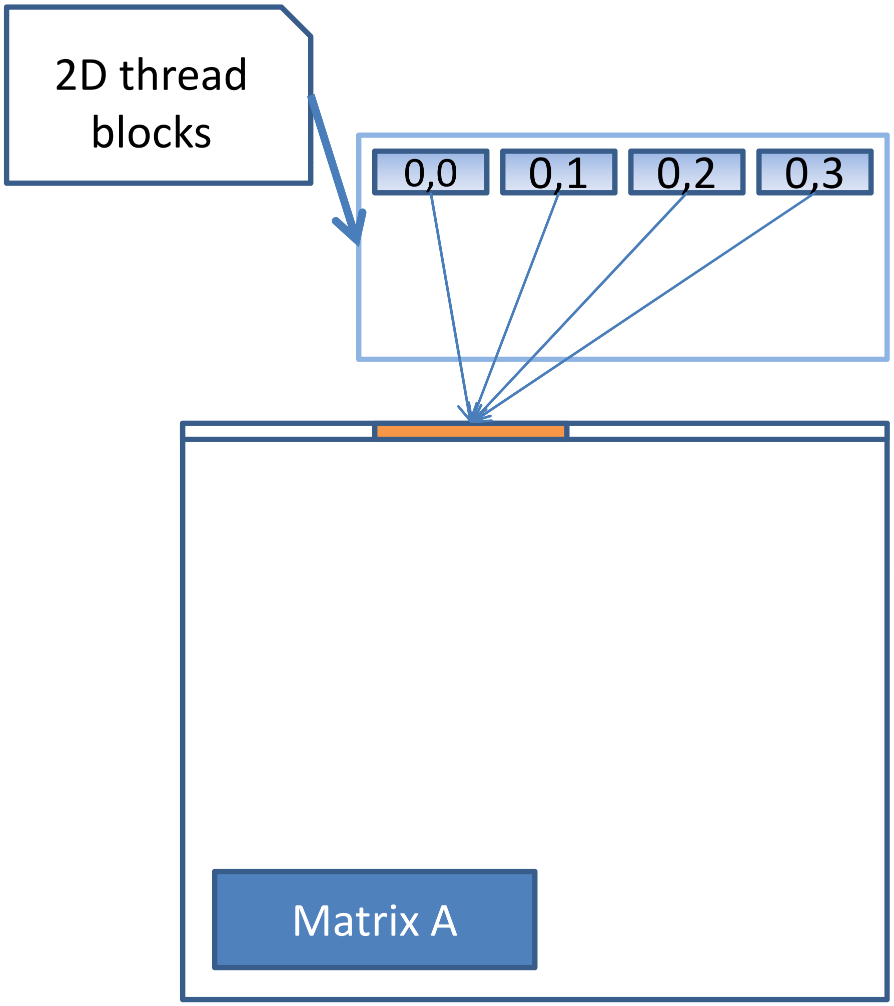
...A[idy][(i+tidx)]

tidx = 0 :15 as block size as 16 threads

i = 0

Thread blocks (0,0) – (0,3) access A[0][0:15]

Detect Data Sharing Among Thread Blocks

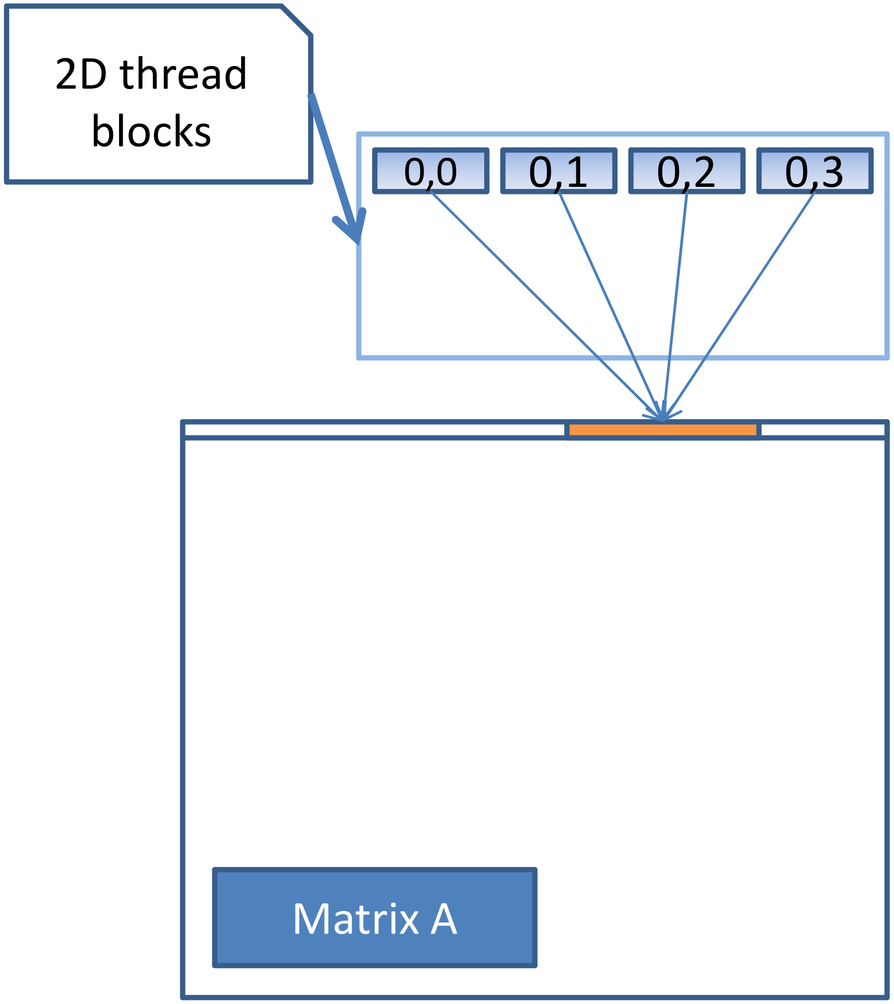


A[idy][(i+tidx)]

i = 16

Thread blocks (0,0) – (0,3) access A[0][16:31]

Detect Data Sharing Among Thread Blocks



A[idy][(i+tidx)]

i = 32

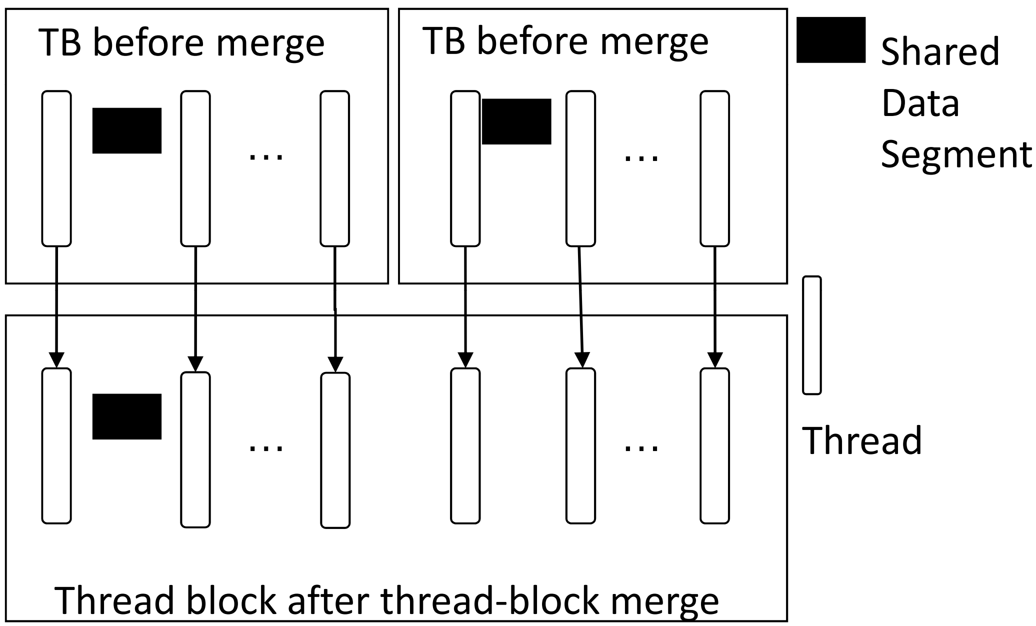
Thread blocks (0,0) – (0,3) access A[0][32:47]

A[idy][(i+tidx)]
Thread blocks in X direction access the same global memory

Thread Block Merge

- Preferred when shared data are in shared memory

Parallelism impact
- Increase the workload of each thread block
- Keep the workload of each thread



Improve memory reuse by merging neighboring thread blocks

Thread Block Merge

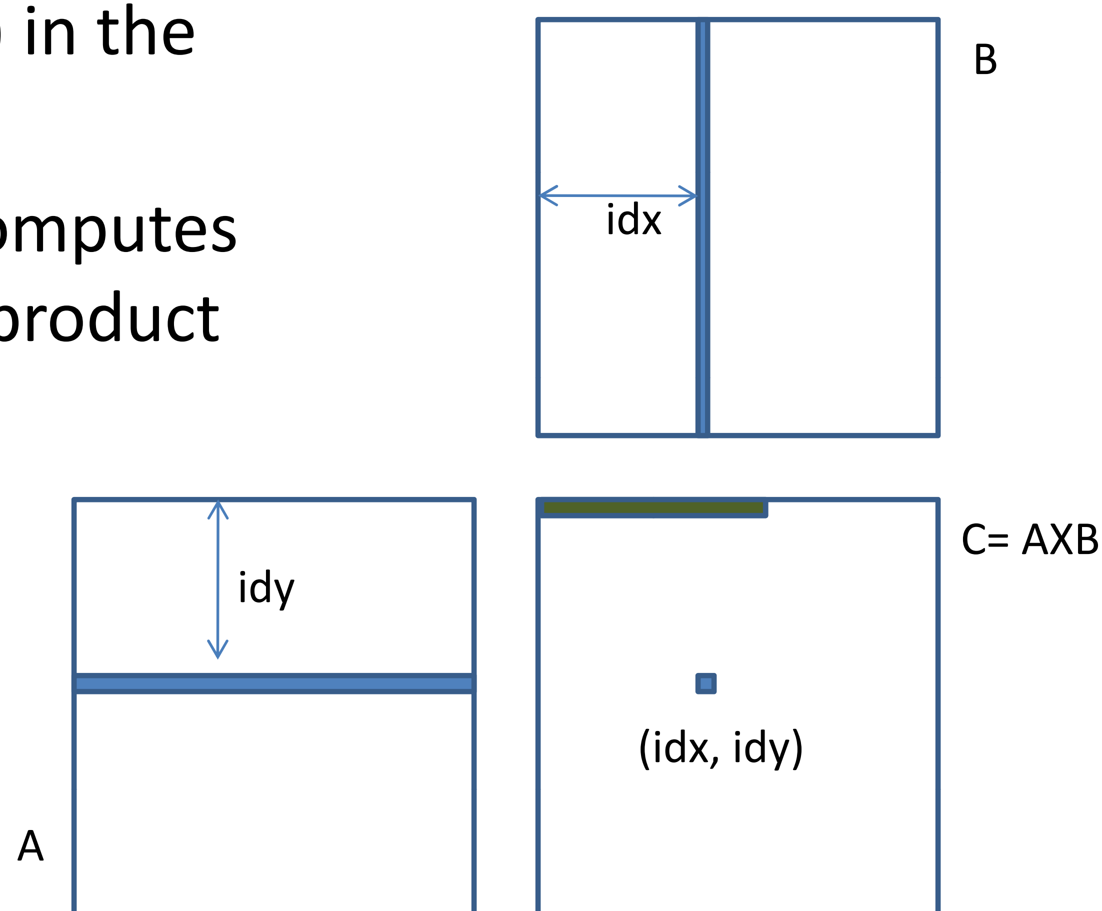
Only first thread block (before merge) needs to load the shared data segment

```
float sum = 0;
for (i=0; i<w; i=(i+16)) {
 __shared__ float shared0[16];
 if (tidx<16)  {
    shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
 }
 __syncthreads();
 int k;
 for (k=0; k<16; k=(k+1)) {
    sum+=shared0[(0+k)]*b[(i+k)][idx]);
 }
 __syncthreads();
}
c[idy][idx] = sum;
```

Thread block merge of MM

Physical Meaning of the Kernel (merged 2 blocks along the X direction)

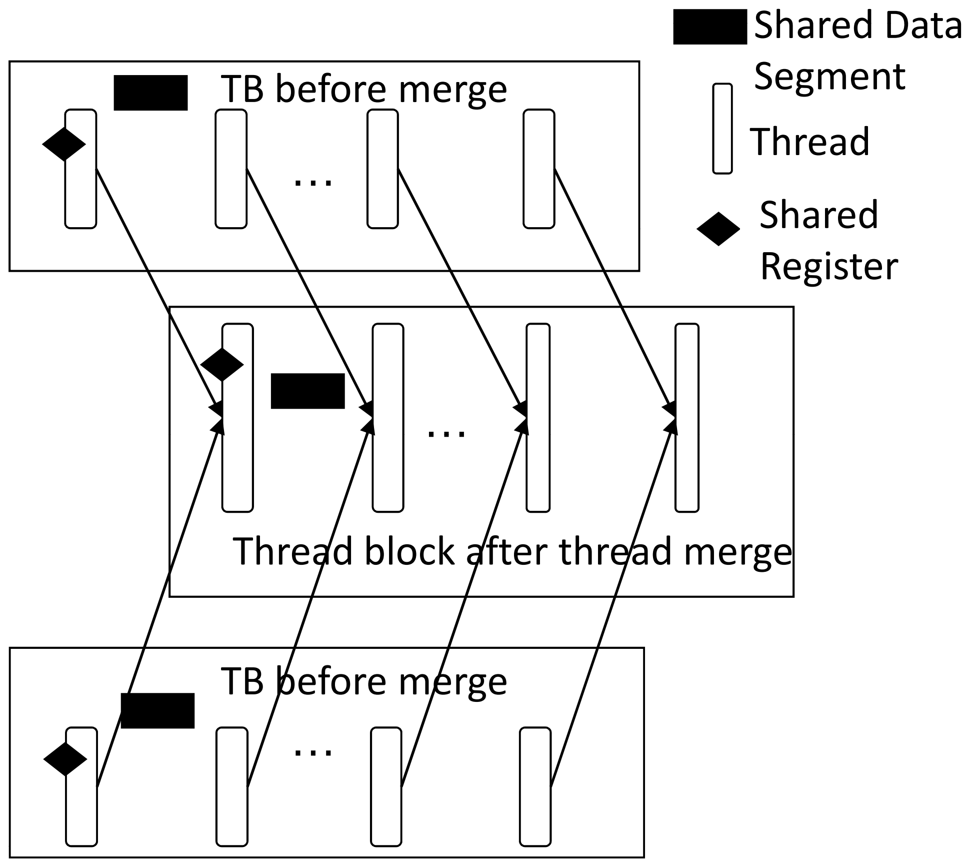
- One thread computes one element at (idx, idy) in the product matrix
- One thread block computes 32 elements in the product matrix
- Tile size: 32x1



Thread Merge

- Preferred when shared data are in register file

Parallelism impact
- Increase thread workload (ILP)
- Keep the number of threads in a thread block



Improve memory reuse by merging threads from neighboring thread blocks.

Code Before Thread Merge

```
float sum = 0;
for (i=0; i<w; i=(i+16)) {
  __shared__ float shared0[16];
  if (tidx<16)  {
      shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
  }
  __syncthreads();
  int k;
  for (k=0; k<16; k=(k+1)) {
      sum+=shared0[(0+k)]*b[(i+k)][idx]);
  }
  __syncthreads();
}
c[idy][idx] = sum;
```

Data shared among thread blocks along Y direction

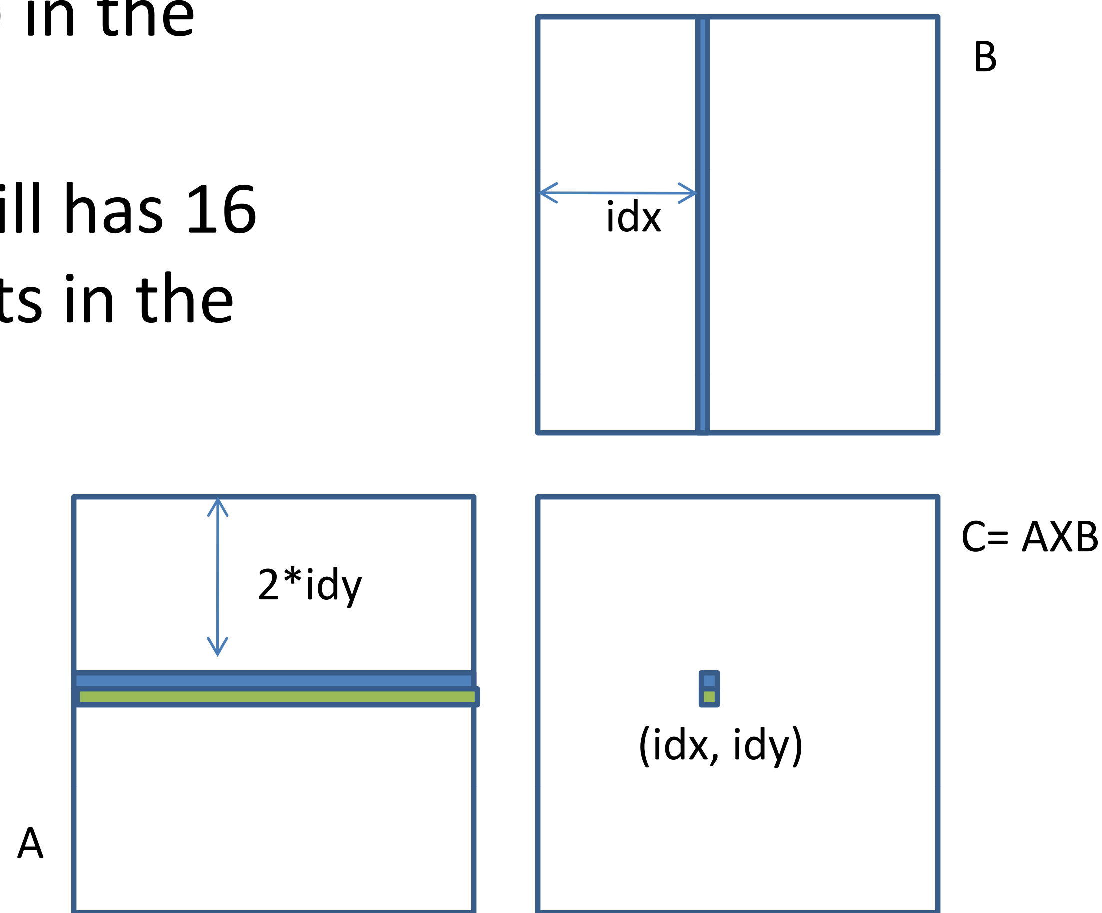
Code After Thread Merge

Duplicate statements except the shared data

```
float sum_0 = 0;
......
float sum_31 = 0;
for (i=0; i<w; i=(i+16)) {
  __shared__ float shared0_0[16];
  ......
  __shared__ float shared0_31[16];
  if (tidx<16)  {
      shared0_0[(0+tidx)]=
              a[idy*32+0][((i+tidx)+0)];
      ......
      shared0_31[(0+tidx)]=
       a[idy*32+31][((i+tidx)+0)];
   }
   syncthreads();
int k;
    for (k=0; k<16; k=(k+1)) {
       float r0 = b[(i+k)][idx]);
       sum_0+=shared0[(0+k)]*r0;
       ......
       sum_31+=shared0_31[0+k]*r0;
    }
    __syncthreads();
}
c[idy*32+0][idx] = sum_0;
......
c[idy*32+31][idx] = sum_31;
```

Physical Meaning of the Kernel (merged 2 threads along Y direction)

- One thread computes two element at (idx, idy) in the product matrix
- One thread block still has 16 threads (32 elements in the product matrix)
- Tile size: 16x2



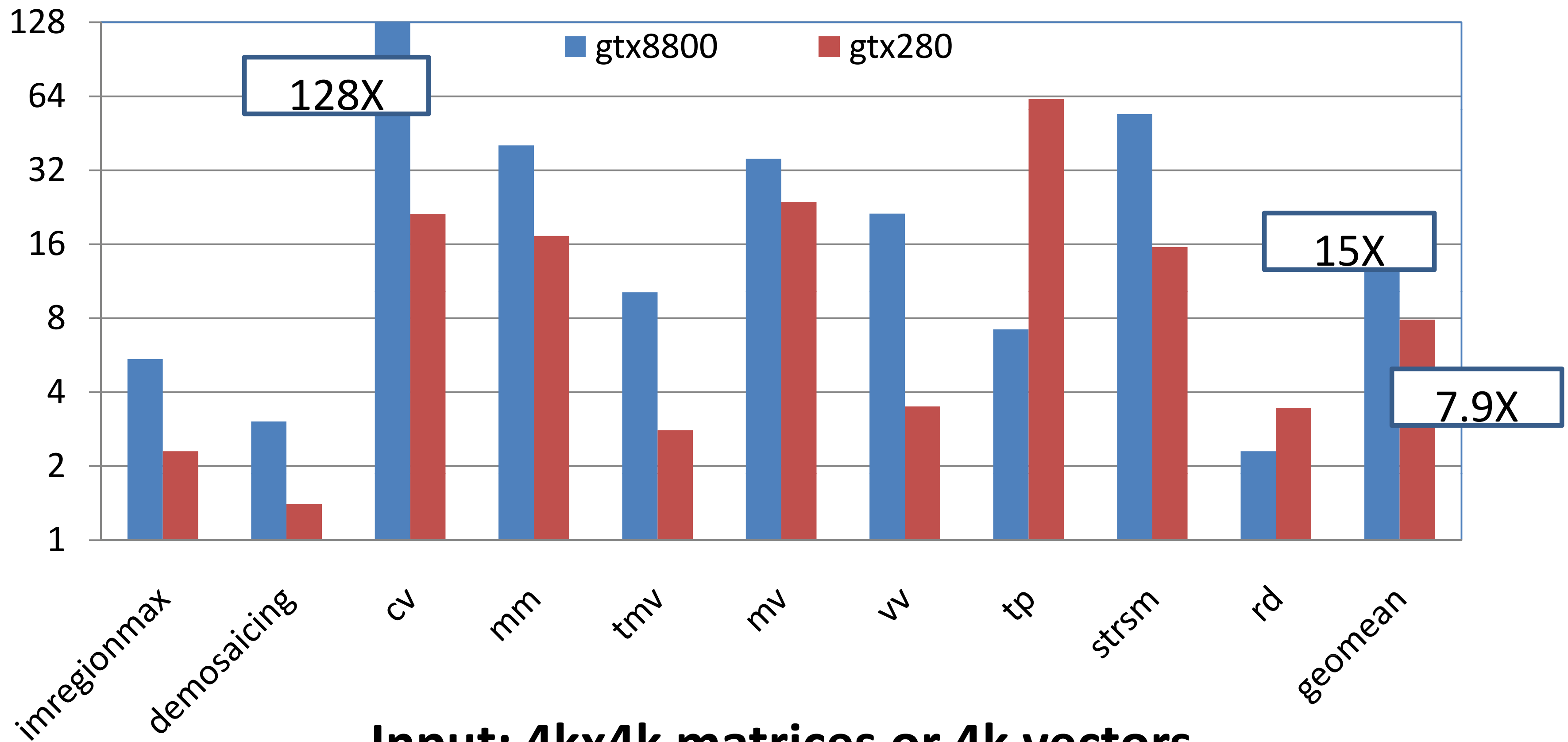
Outline

- Background
- Motivation
- Compiler Framework
  - Memory coalescing
  - Thread (block) merge
- Experimental Results
- Conclusion

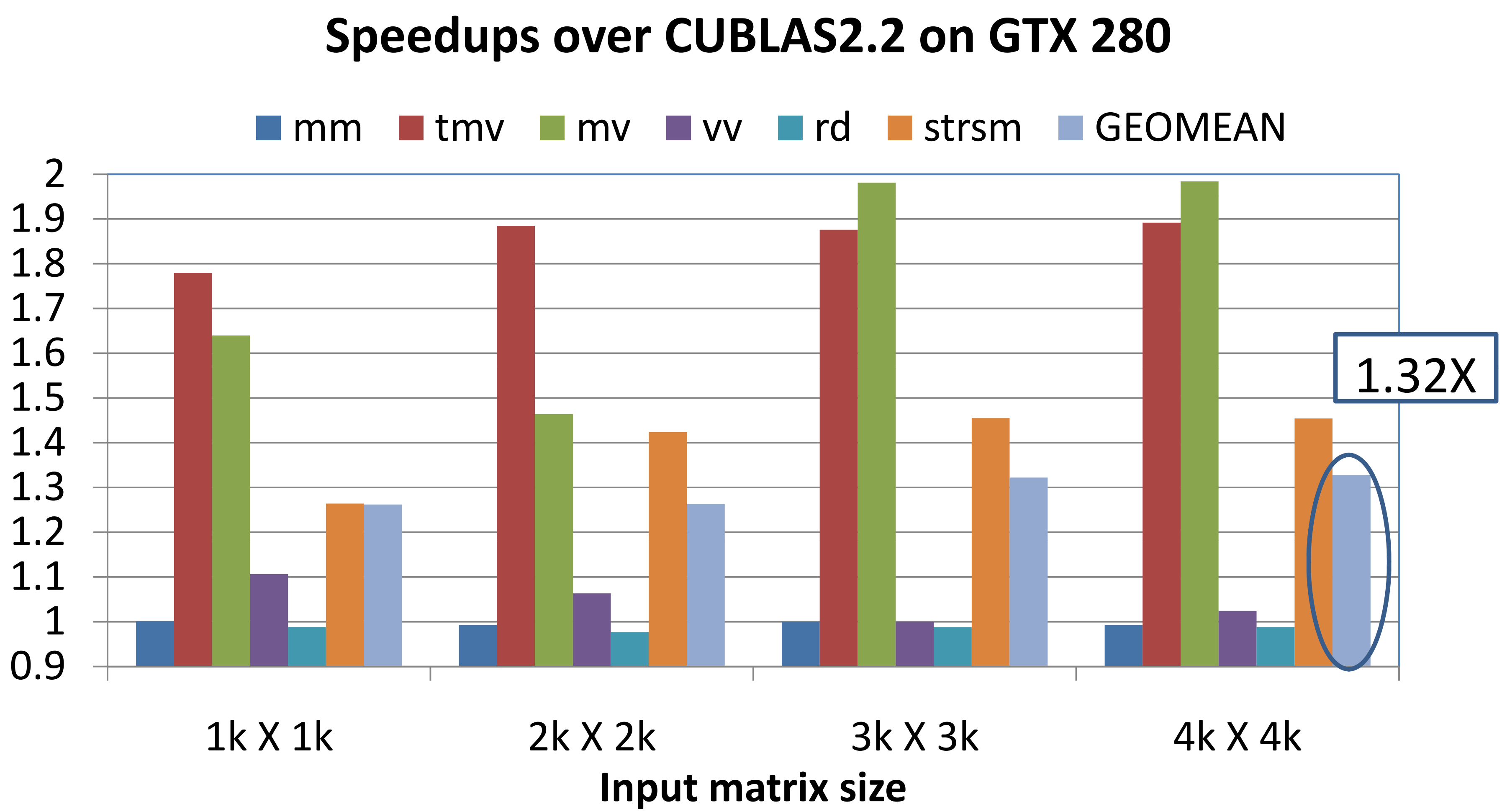
Experimental Methodology

- The proposed compiler is implemented as a source-to-source translator using Cetus [Lee et.al., LCPC 2003]
- Experimental environment
  - Operating system: 32-bit CentOS 5.2
  - CUDA SDK 2.2 for GTX8800 and GTX280
  - CUDA SDK 3.1 beta for GTX 480 (Fermi)
- 10 scientific/media processing algorithms
  - Naïve kernel code is available at http://code.google.com/p/gpgpucompiler/

Speedups over Naïve Kernels



Input: 4kx4k matrices or 4k vectors



•Similar performance for matrix multiplication, vector-vector multiplication, and reduction

•14x-1.9x sppedup for transpose matrix vector multiplication, matrix vector multiplication, and strsm

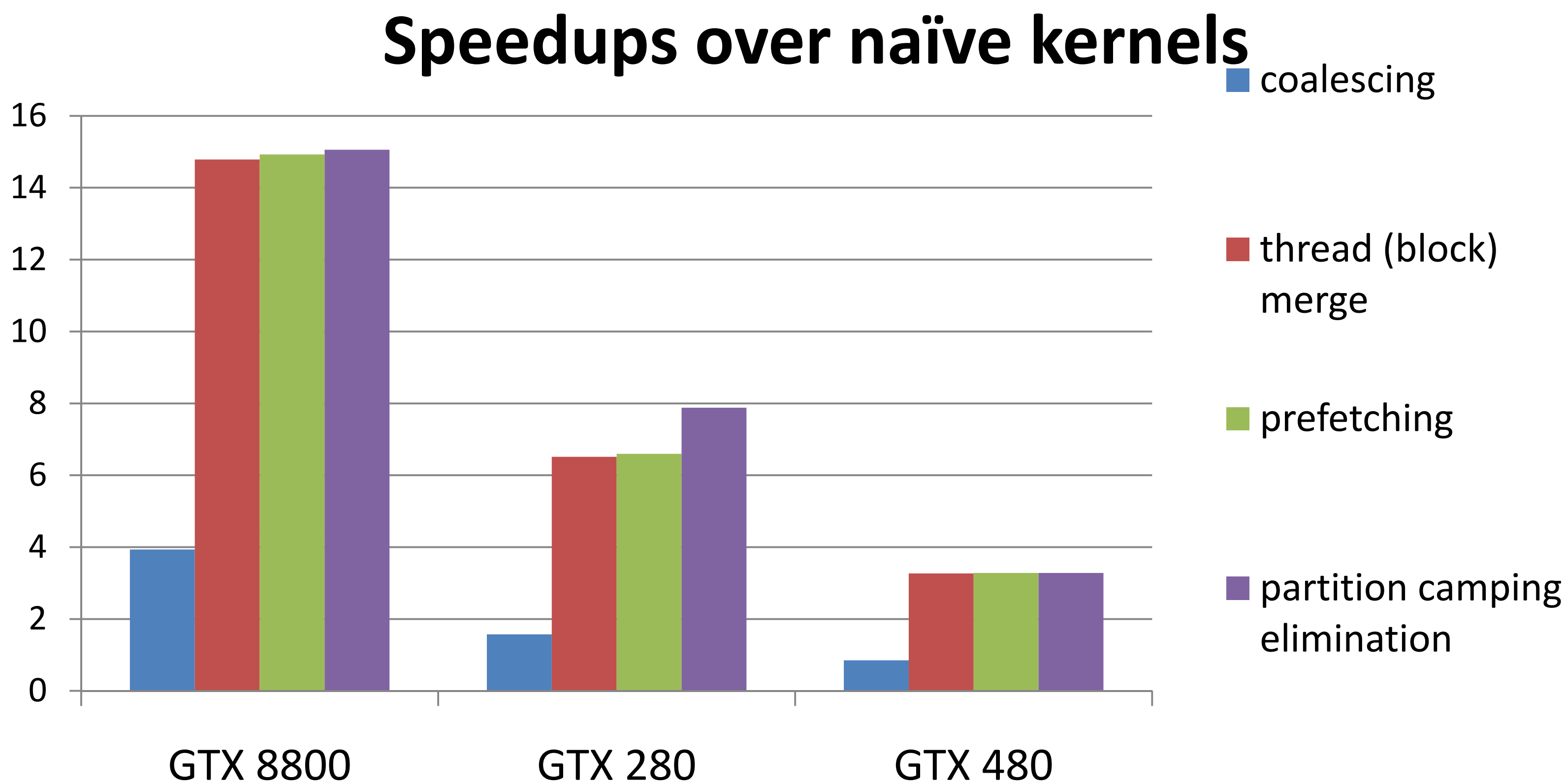
Summary

- We present an optimizing compiler for GPGPU programs.
- Our experimental results demonstrate the effectiveness of the proposed compiler optimizations.

- The open-source compiler website:
  - http://code.google.com/p/gpgpucompiler/
  - Contains the compiler code, the naïve kernels and the optimized kernels (optimized for GTX 280)

Thanks & Questions?

Impact of each optimization

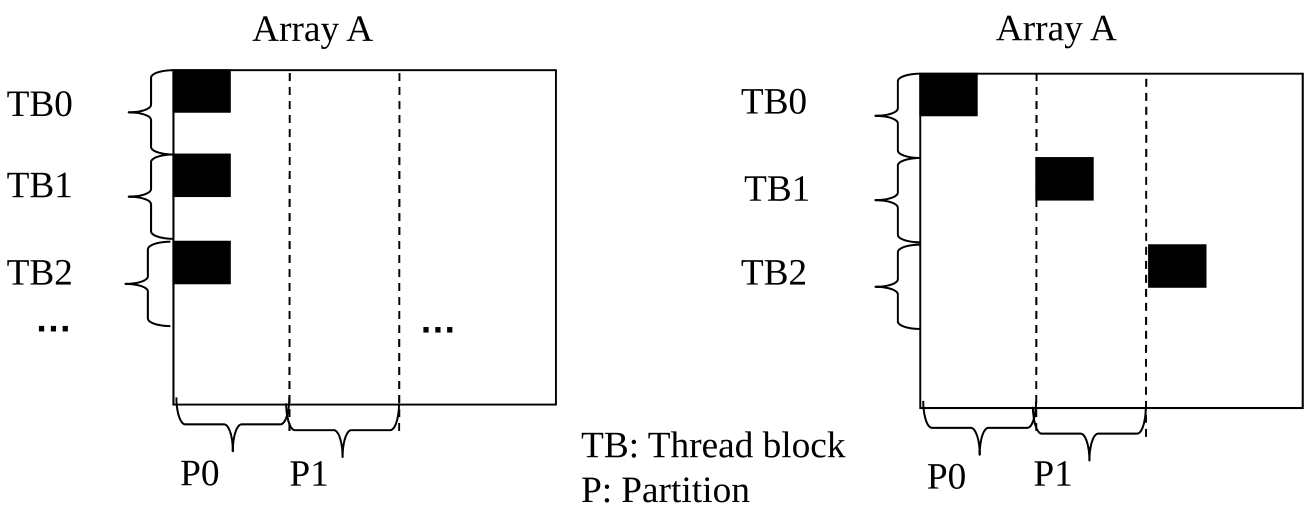


- Coalesced memory access has less impact on the GTX 280 and GTX480.

- Thread (Block) merge achieve similar speedup over the code after the coalesced step (3.7x, 4.1x, 3.8x)

Partition Camping on Global Memory

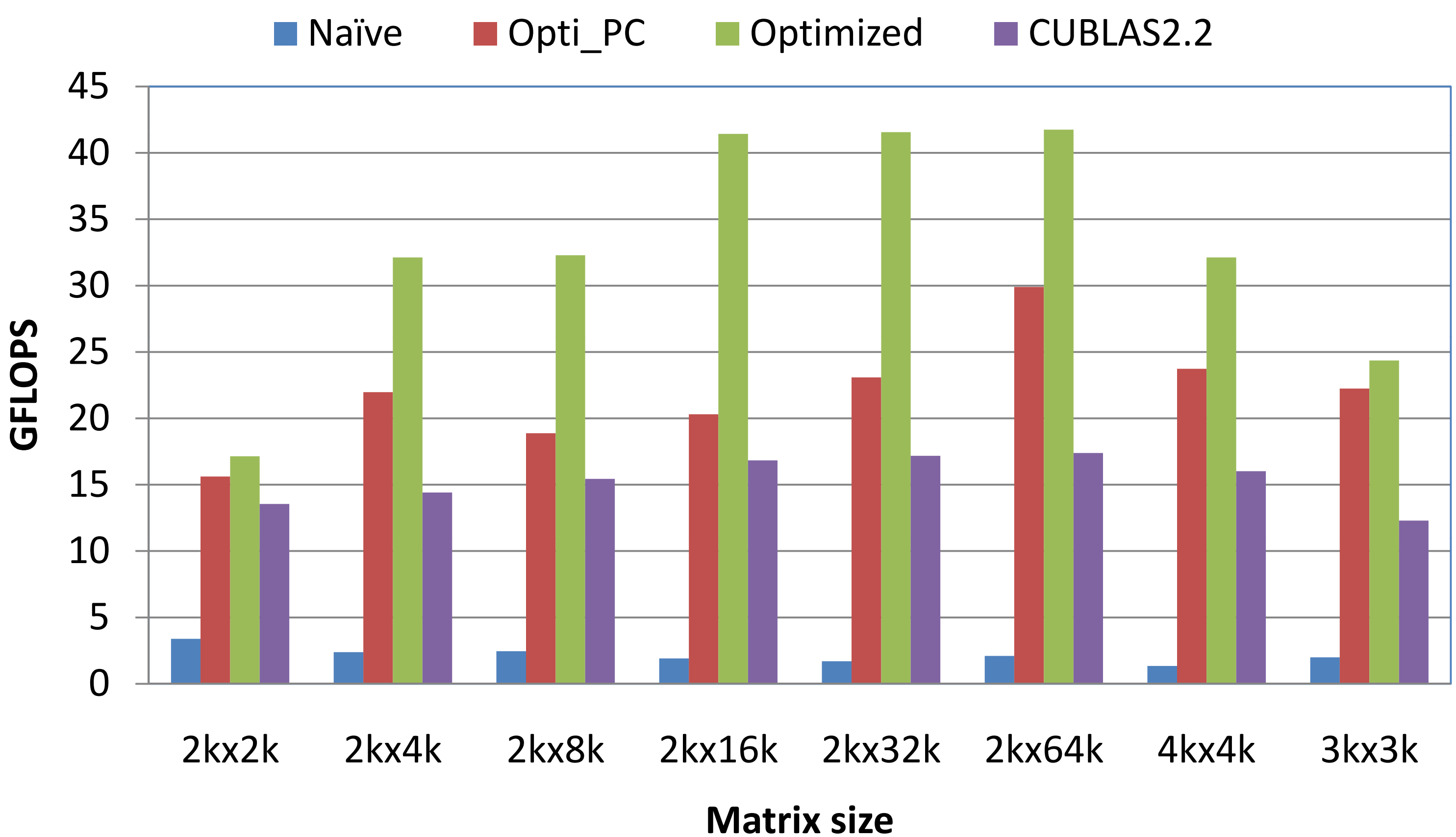
- Global memory traffic should be evenly distributed among all the partitions



(a) Accesses to array A resulting in conflicts at partition 0

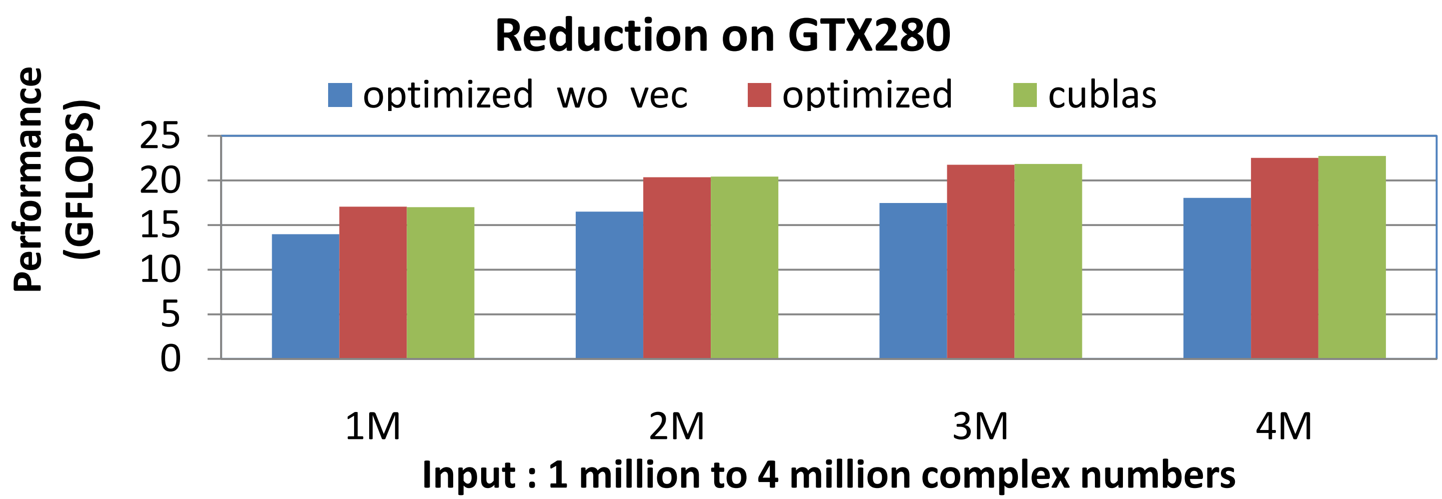
(b) offset to eliminates the conflicts

Matrix vector multiplication on GTX 280



Opti_PC : the optimized kernel without partition camping elimination

Vectorization



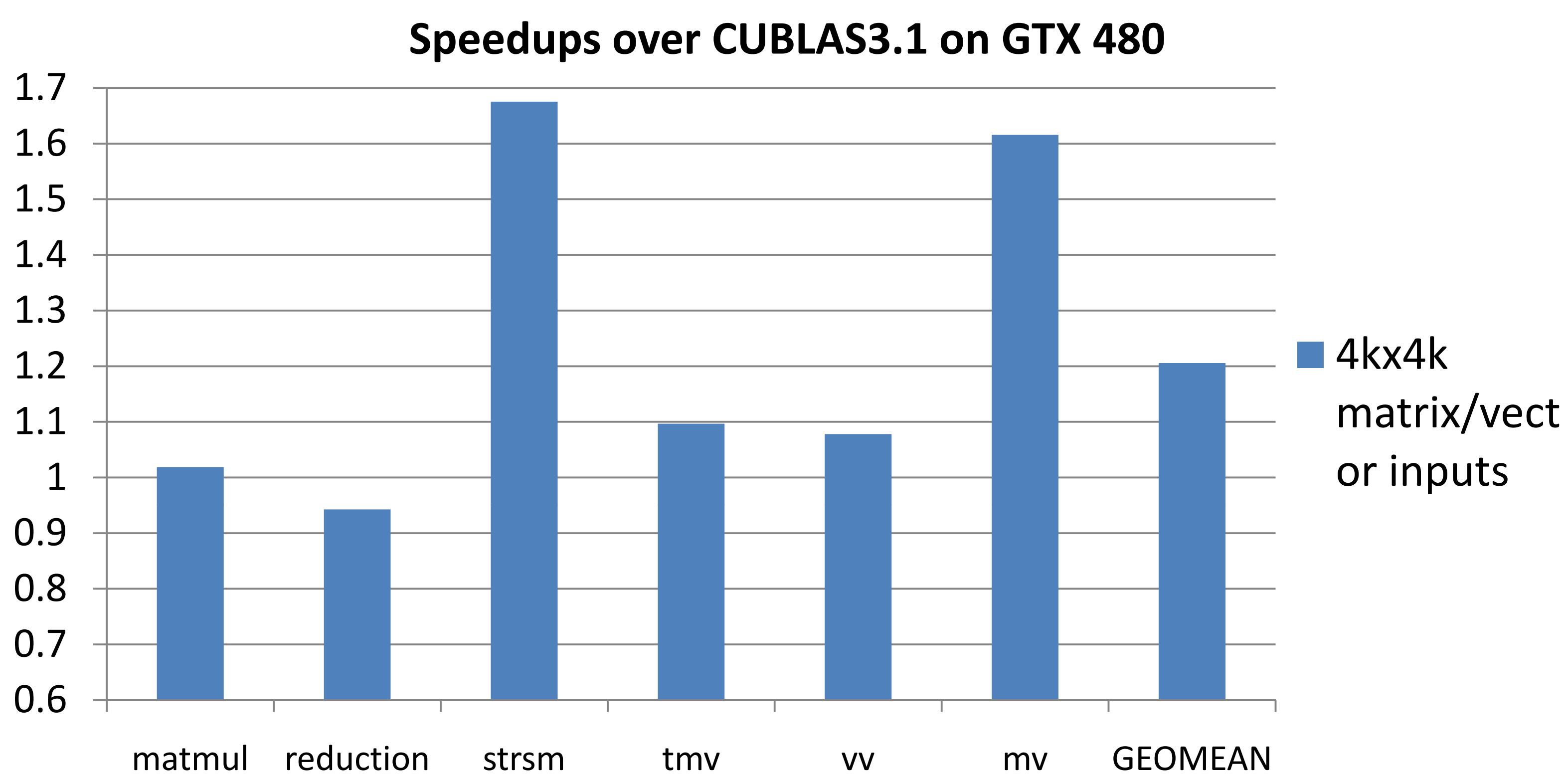

```
float a1 = A[2*idx];
float a2 = A[2*idx+1];
```

Before vectorization

```
float2* A_f2 = (float2*)A;
float2 a_f2 = A[2*idx];
float a1 = a_f2.x;
float a2 = a_f2.y;
```

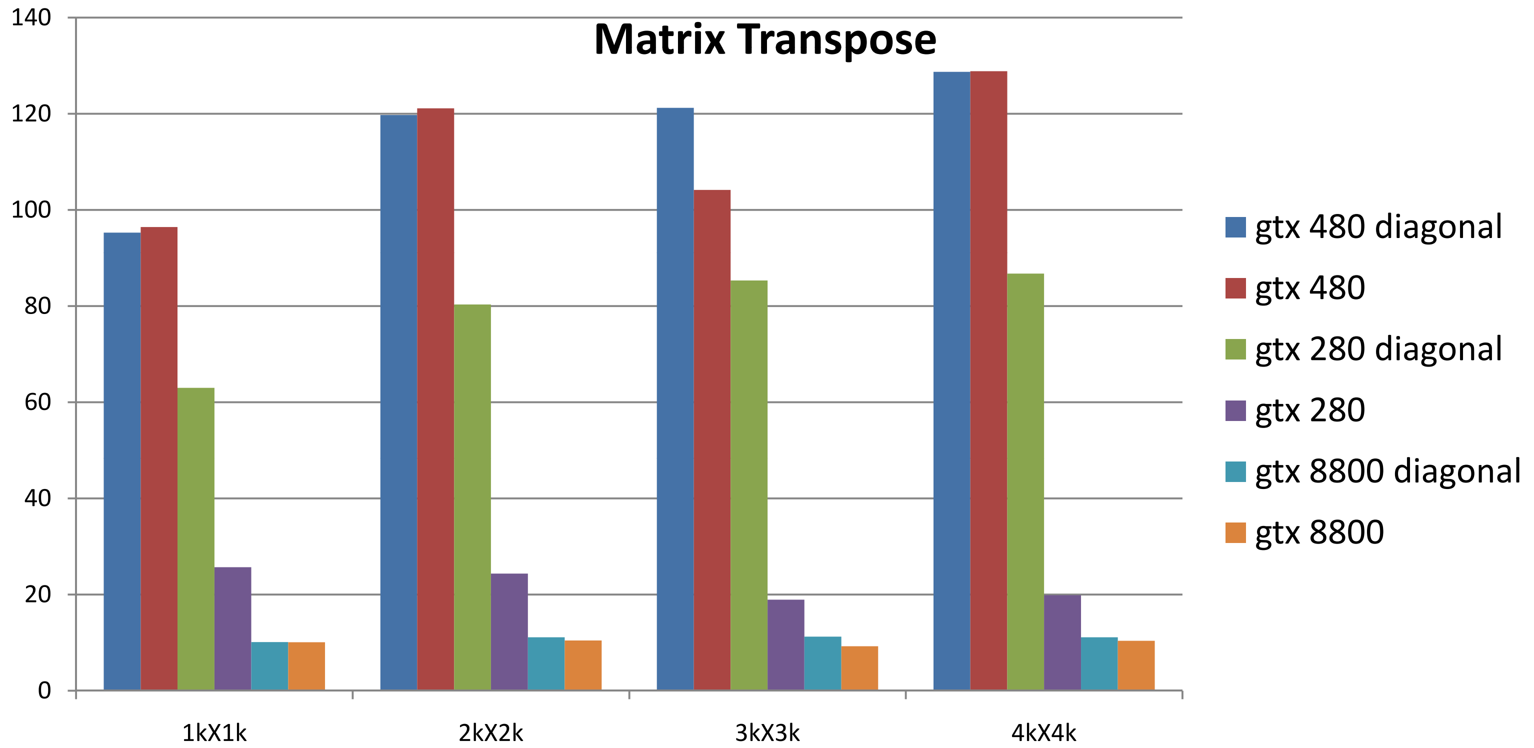
After vectorization

CUBLAS on Fermi

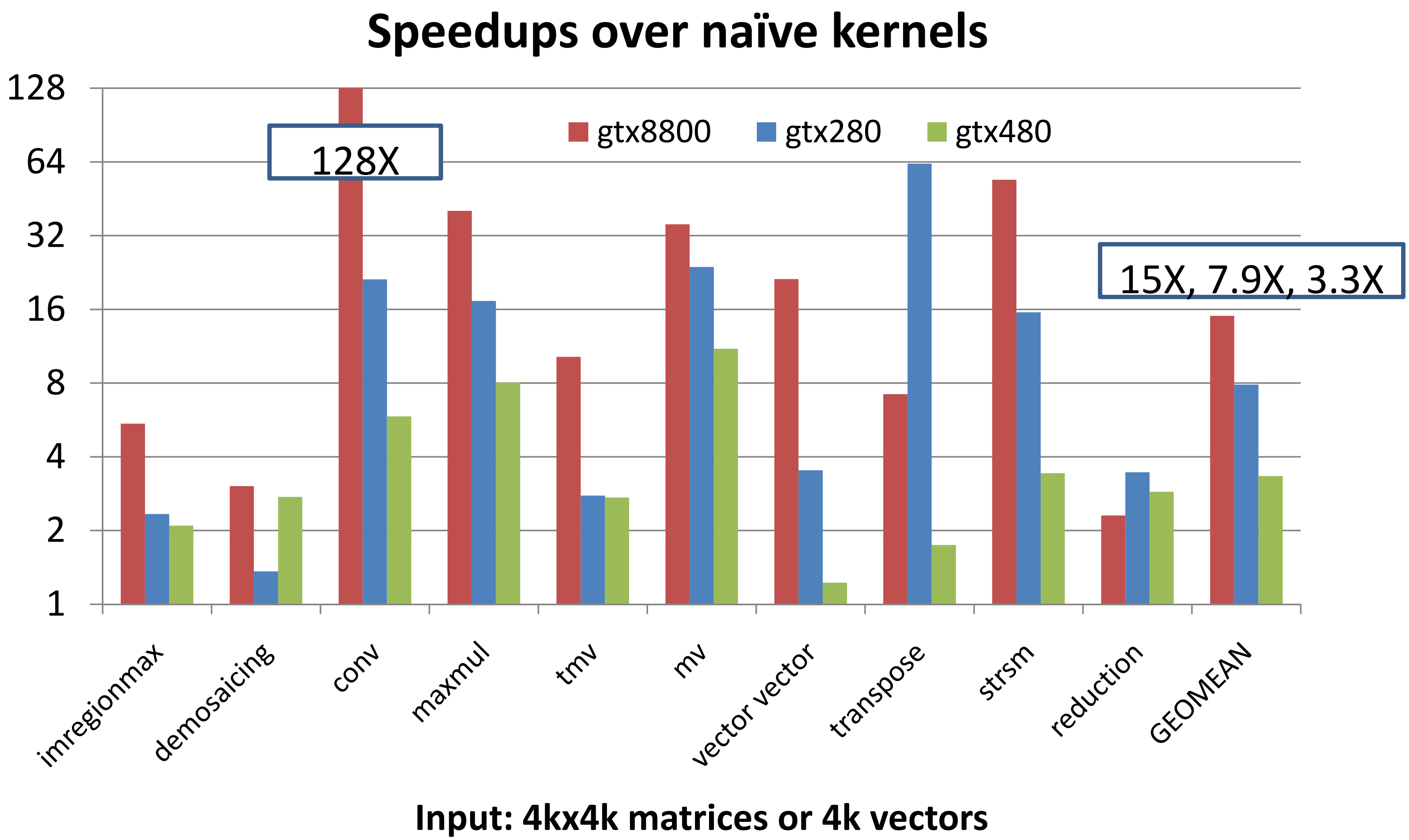


The optimized version is based on GTX 280 configuration.

Transpose



Matrix Transpose
140
120
100
80
60
40
20
0
1kX1k
2kX2k
3kX3k
4kX4k
gtx 480 diagonal
gtx 480
gtx 280 diagonal
gtx 280
gtx 8800 diagonal
gtx 8800


Speedups over naïve kernels
128
64
32
16
8
4
2
1
128X
gtx8800
gtx280
gtx480
15X, 7.9X, 3.3X
imregionmax
demosaicing
conv
maxmul
tmv
mv
vector vector
transpose
strsm
reduction
GEOMEAN
Input: 4kx4k matrices or 4k vectors