Code Optimizations for High Performance GPU Computing

Yi Yang and Huiyang Zhou

Department of Electrical and Computer Engineering
North Carolina State University

Question to answer

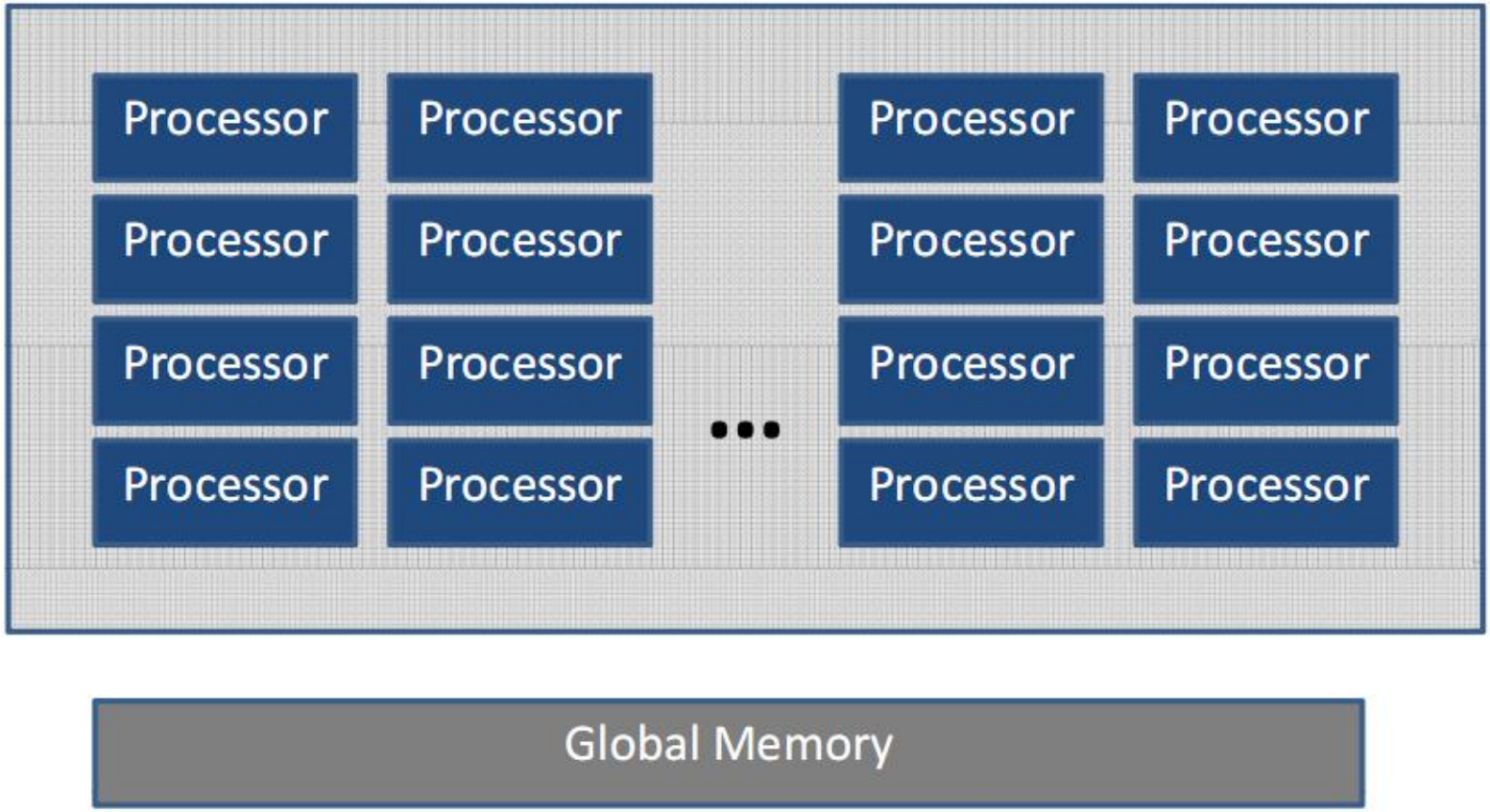
- Given a task to accelerate some algorithm, e.g., solving PDE, image fltering, etc., using GPU computing

- How can we start? How can we systematically develop high performance GPGPU programs?

Outline

- Hardware abstraction
- A systematic approach to developing high performance GPGPU programs
- Optimization techniques with case studies
  - Coalesced memory accesses
  - Data reuse through thread (block) merge
  - Eliminating partition conflicts
  - Leveraging constant cache
- Conclusions

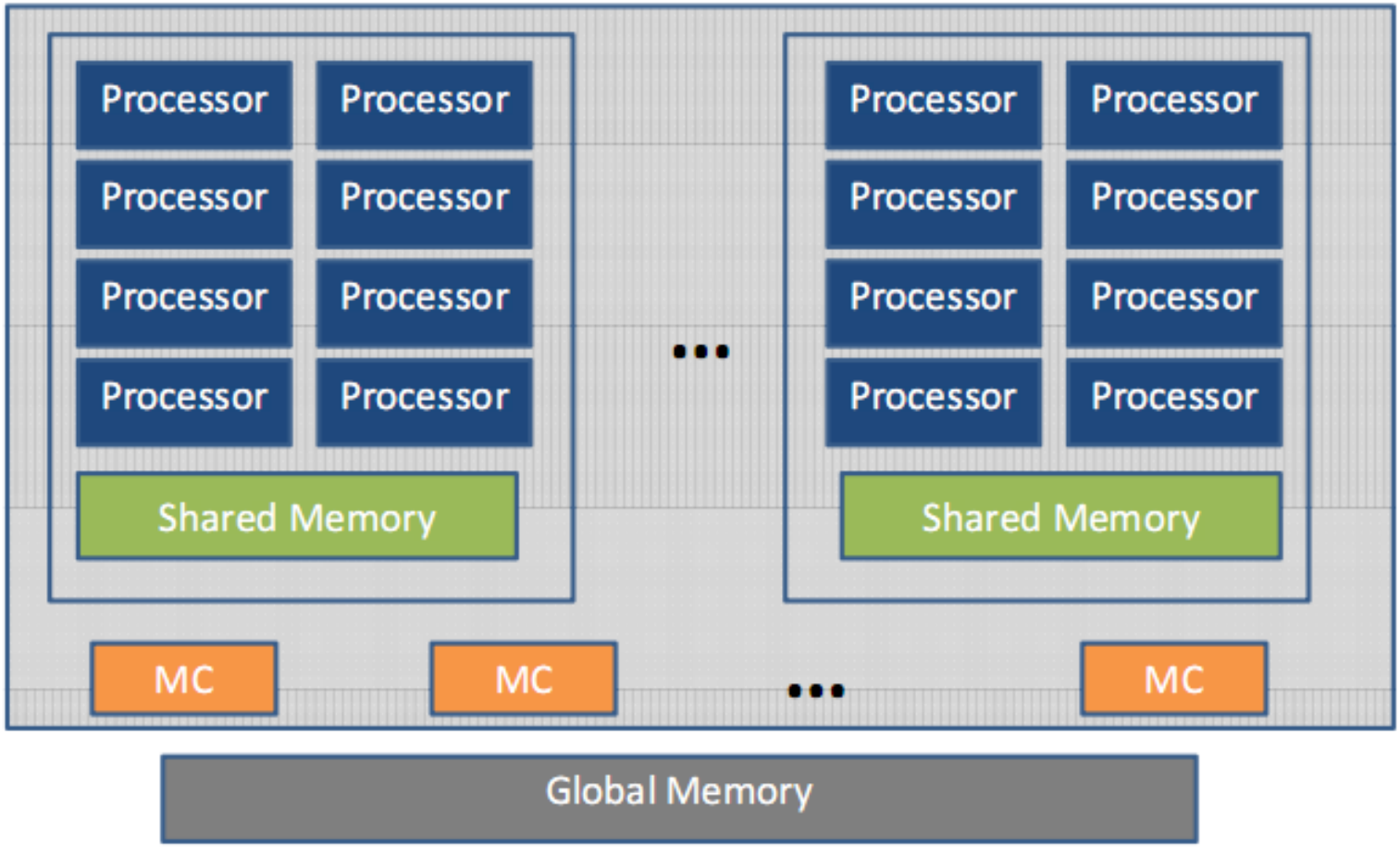
Hardware Abstraction of GPU Architecture



Based on this simple abstraction, develop a naïve implementation without considering optimizations.

Focuses: Data level parallelism Functional correctness

GPGPU Architecture

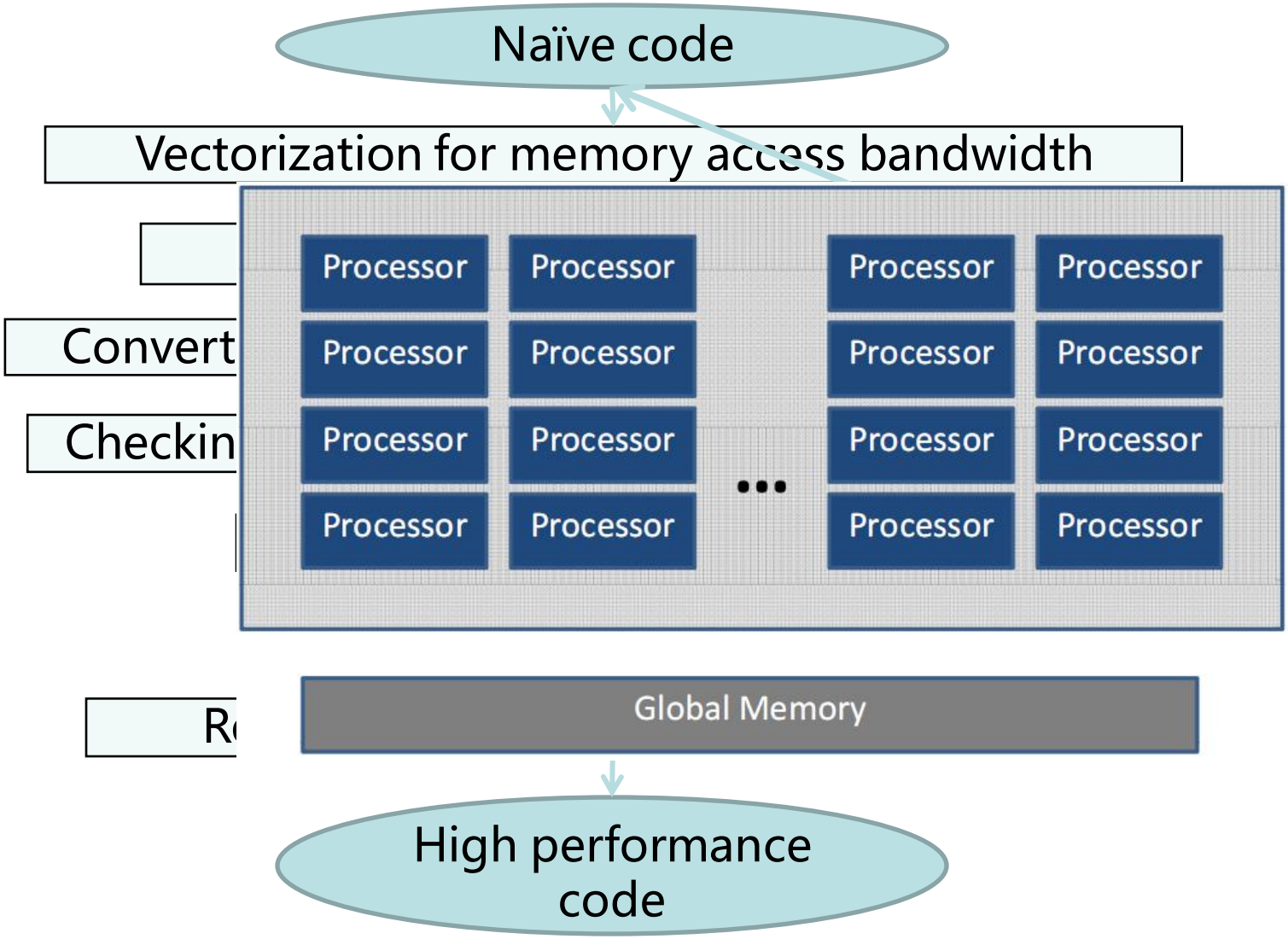


- Fast (local) communication among processors in a SM through shared memory
- Memory requests need to be evenly distributed among MCs to avoid conflicts/partition camping

Key to Performance

- Global memory access bandwidth
  - Coalesced global memory accesses
  - Memory partitions
- Fast data accesses
  - Shared memory
  - Constant Cache
  - Texture Cache
  - Register
- Balanced resource usage, balanced ILP and TLP
  - Thread level: register usage
  - Thread block level: shared memory usage,

Developing High Performance GPGPU Code



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;
```

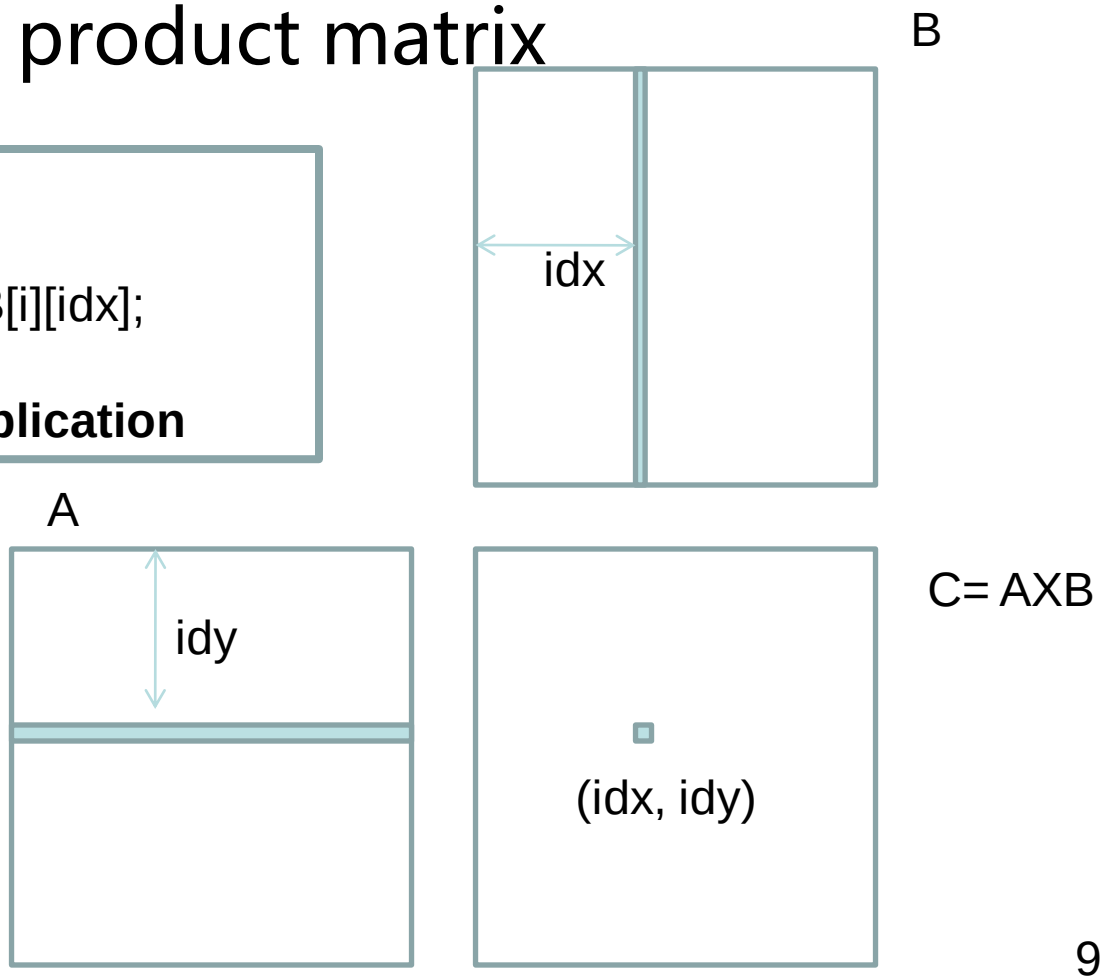
Naïve matrix multiplication

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;
```

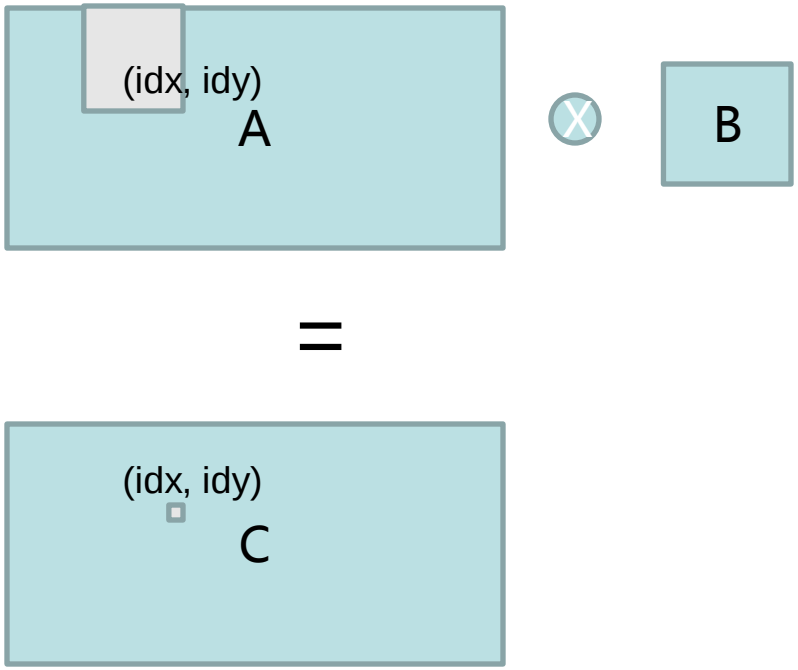
Naïve matrix multiplication



Outline

- Hardware abstraction
- A systematic approach to developing high performance GPGPU Programs
- Optimization techniques with case studies
  - Coalesced memory accesses
  - Data reuse through thread (block) merge
  - Eliminating partition conflicts
  - Leveraging constant cache
- Conclusions

Case Study: Convolution



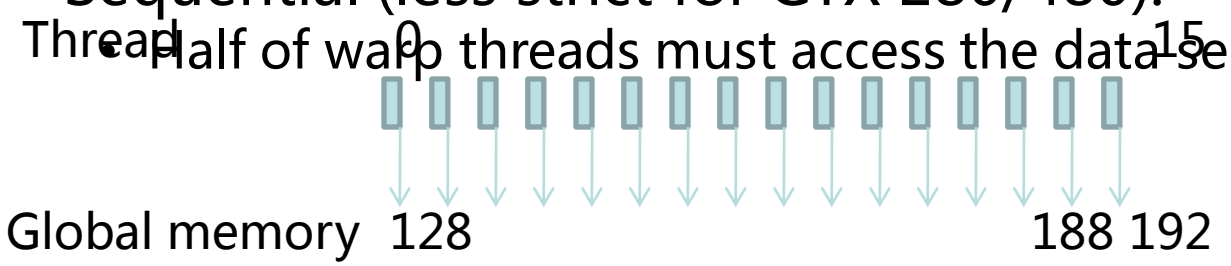

```
float sum = 0;
for (j=0; j<8; j=j+1) {
    for (i=0; i<8; i=i+1) {
        float a;
        float b;
        a = A[idy-j][idx-i];
        b = B[j] [i];
        sum += a*b;
    }
}
C[idy][idx] = sum;
```

Naïve version of Convolution one thread computes one output pixel at (idx, idy)

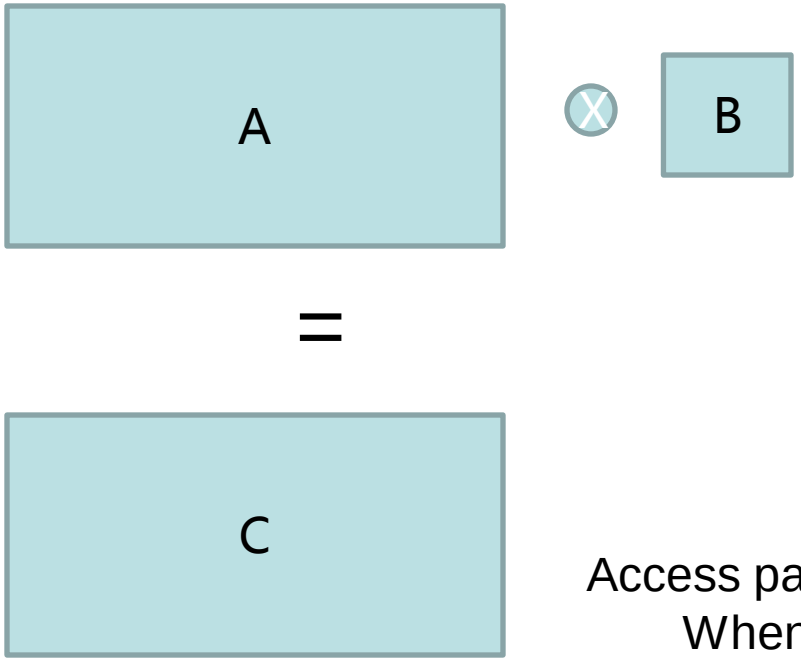
- A is input matrix
- B is 8x8 filter matrix
- C is output matrix

Coalesced Global Memory Access

- Needed by GPU to achieve high memory bandwidth
- Examined at the half-warp granularity
- Threads in a warp have consecutive thread ids
- 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 coalesced memory accesses




```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j][i];
      sum += a*b;
 }
```

Inner loop of convolution

Access pattern of B[j][i]:

When i = 0; B[j][0] for all the threads in a warp

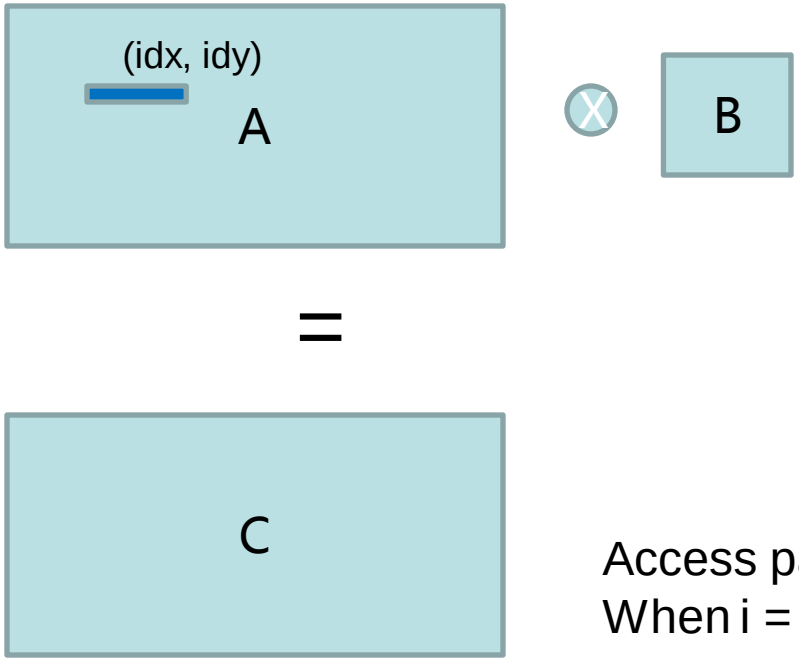
When i = 1; B[j][1] for all the threads in a warp

…

Therefore, it is not coalesced.

As B is a small kernel, we can store it in the shared memory or constant memory (cache).

Checking coalesced memory accesses

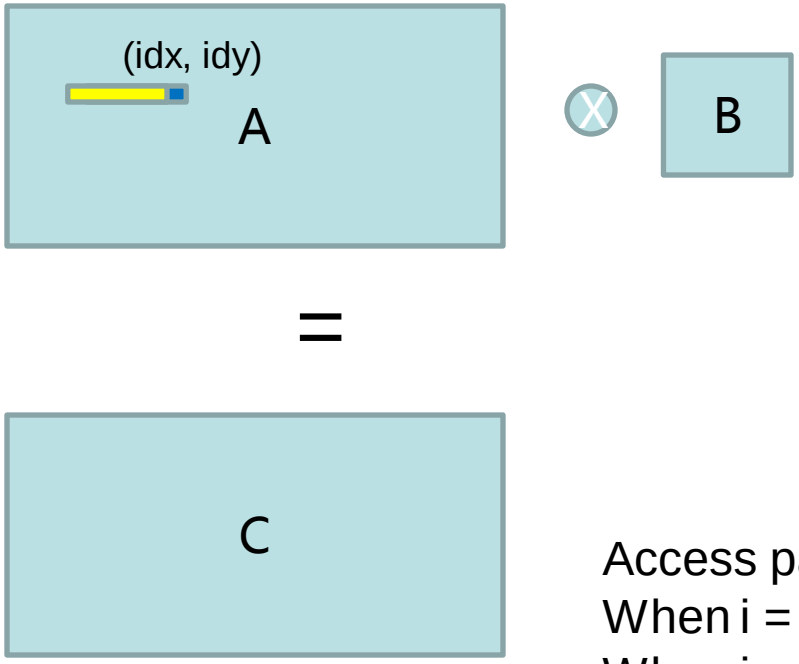



```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j] [i];
      sum += a*b;
 }
```

Inner loop of convolution

Access pattern of A[idy-j][idx-i]
When i = 0, A[idy-j][idx]

Checking coalesced memory accesses

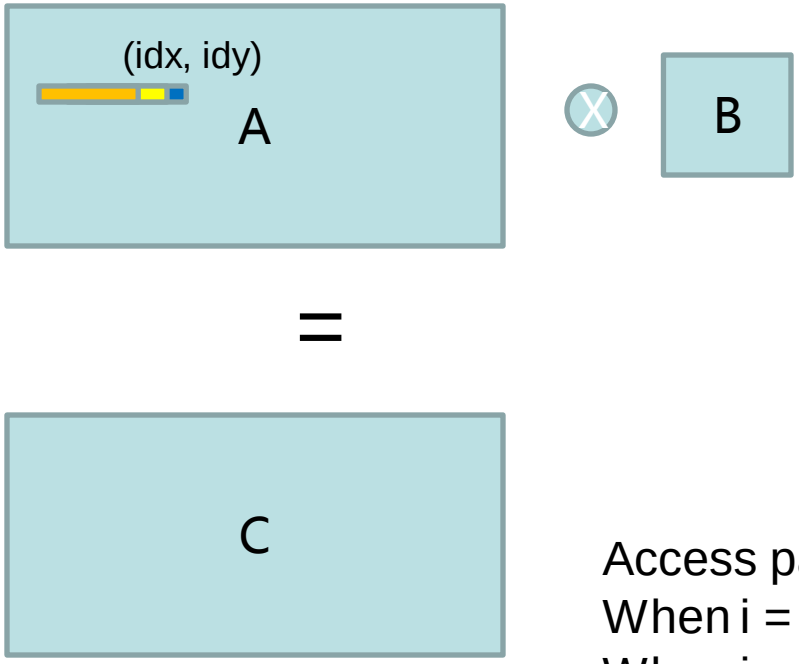



```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j] [i];
      sum += a*b;
 }
```

Inner loop of convolution

Access pattern of A[idy-j][idx-i]
When i = 0, A[idy-j][idx]
When i = 1, A[idy-j][idx-1]

Checking coalesced memory accesses

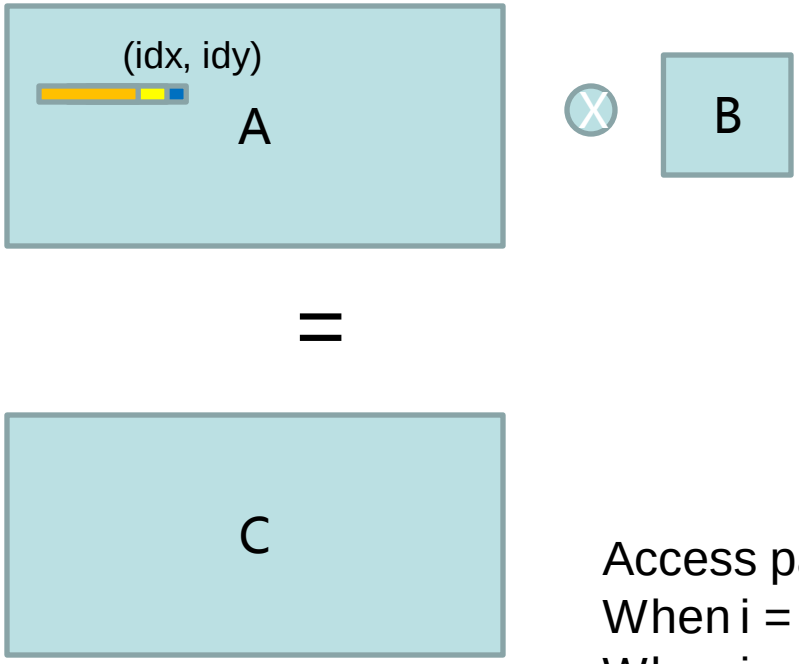



```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j] [i];
      sum += a*b;
 }
```

Inner loop of convolution

Access pattern of A[idy-j][idx-i]
When i = 0, A[idy-j][idx]
When i = 1, A[idy-j][idx-1]
When i = 2, A[idy-j][idx-2]

Checking coalesced memory accesses

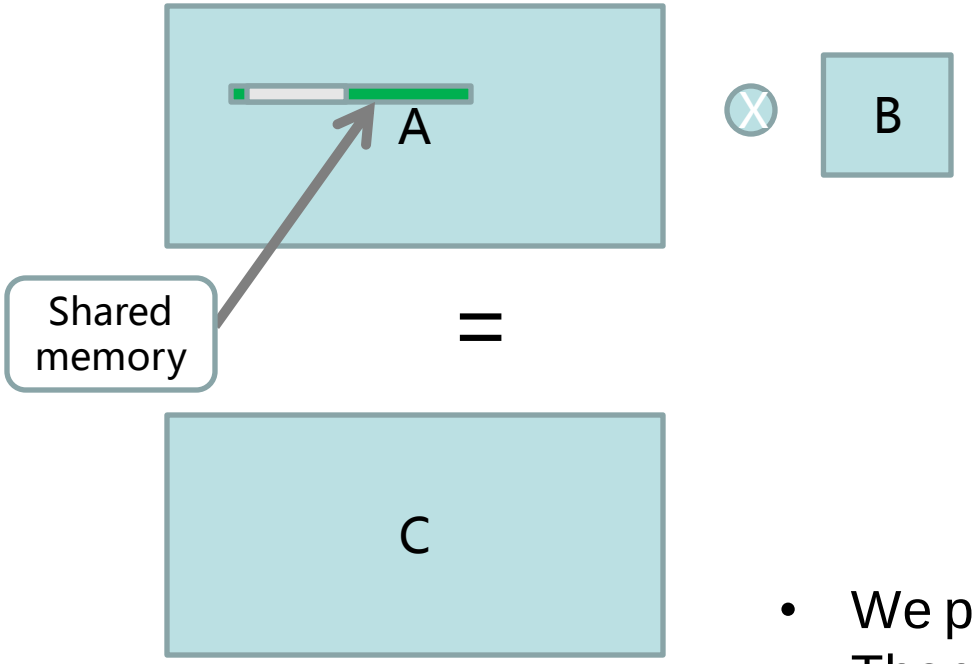



```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j] [i];
      sum += a*b;
 }
```

Inner loop of convolution

Access pattern of A[idy-j][idx-i] for the warp
When i = 0, A[idy-j][idx]
When i = 1, A[idy-j][idx-1]
When i = 2, A[idy-j][idx-2]
…
When i = 7, A[idy-j][idx-7]
Therefore, it is not coalesced. The warp accesses the data: A[idy-j][idx-7:idx+31]

Convert to coalesced accesses




```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j] [i];
      sum += a*b;
 }
```

Inner loop of convolution

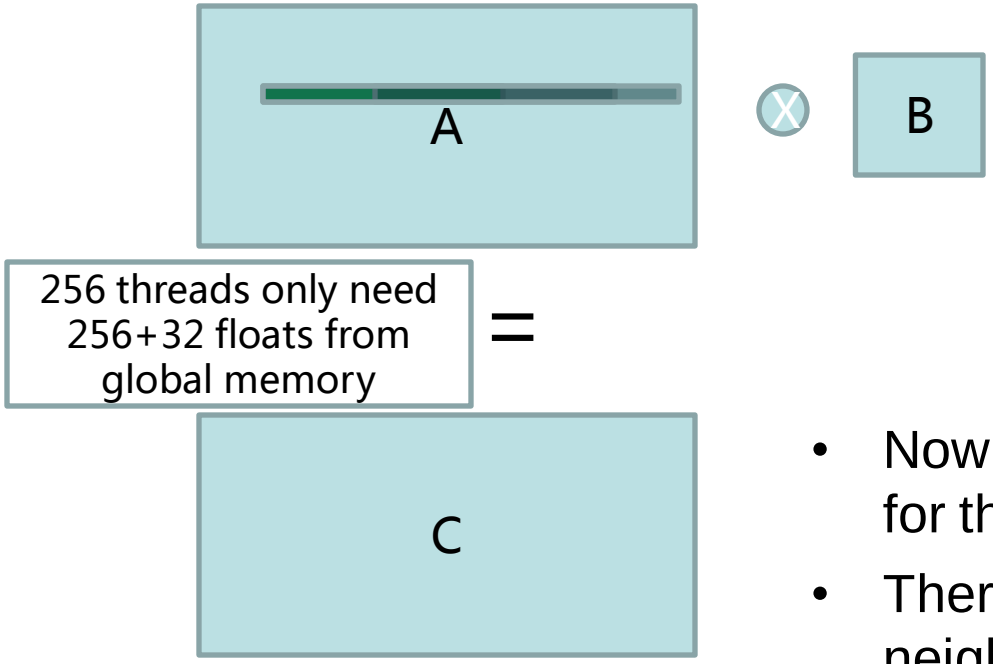
- We preload data into shared memory. Then access it from shared memory. One warp (32 threads) loads 64 floats into shared memory

Coalesced memory access

```
__shared__ float shared_0[64];
shared_0[tidx]=A[idy-j][idx-32];
shared_0[tidx+32]=A[idy-j][idx];        // load data into shared memory
__syncthreads();
for (i=0; i<8; i=i+1)
{
        float a=shared_0[tidx+32]; // access data from shared memory
        float b=B[j][i];
        sum+=(a*b);
}
__syncthreads();
```

- 32 threads (one warp) in one thread block
- Each warp access 64 elements A[idy-j][idx-tidx-32 : idx-tidx+31]
- (idx – tidx) is the start position of the thread block

Convolution: thread block merge




```
for (i=0; i<8; i=i+1) {
      float a;
      float b;
      a = A[idy-j][idx-i];
      b = B[j] [i];
      sum += a*b;
 }
```

Inner loop of convolution

- Now one warp needs to load 64 floats for the inner loop
- There are some overlap between neighboring warps/thread blocks

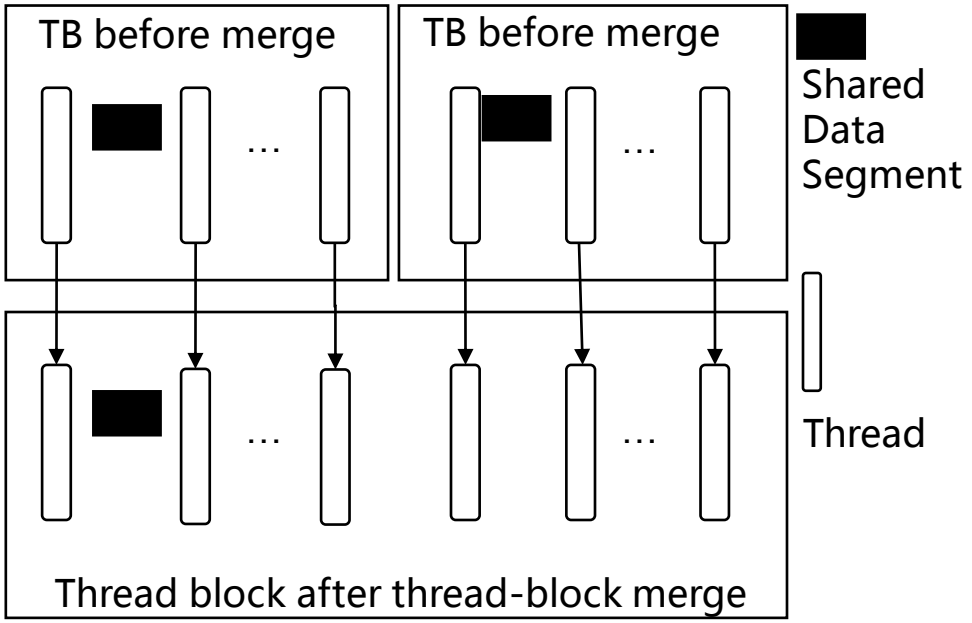
  A[idy-j][idx-tidx-32 : idx-tidx+31]

- If we put more warps into one thread blocks, they can share the overlap part and reduce global memory access

Thread block merge

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

- Advantage:
Don’ t increase register pressure
- Disadvantage :
Data must be in shared memory (slower than register)



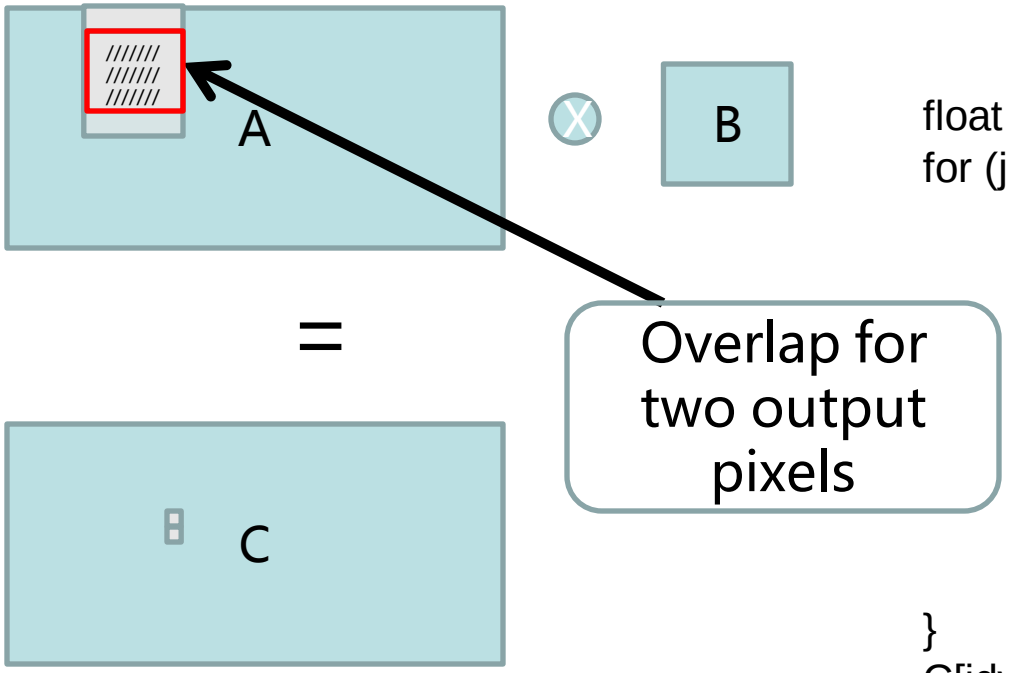
Improve memory reuse by merging neighboring thread blocks

Code after thread block merge

```
__shared__ float shared_0[256+32];
if (tidx<32) shared_0[tidx]=A[idy-j][idx-32];      // only first warp executes
shared_0[tidx+32]=A[idy-j][idx];
__syncthreads();
for (i=0; i<8; i=i+1)
{
        float a=shared_0[tidx+32];
        float b=B[j][i];
        sum+=(a*b);
}
__syncthreads();
```

- 256 threads in one thread block

Case study: Convolution

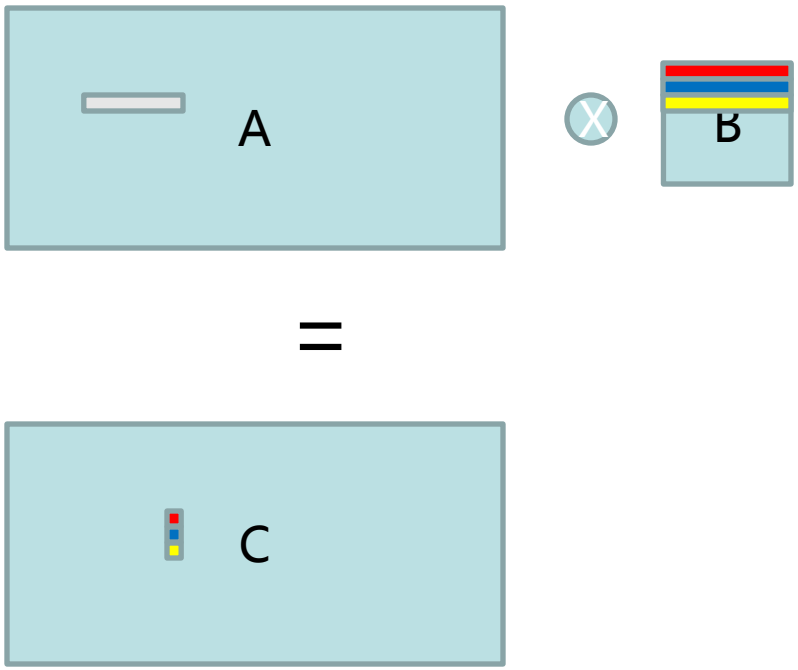



```
float sum = 0;
for (j=0; j<8; j=j+1) {
    for (i=0; i<8; i=i+1) {
        float a;
        float b;
        a = A[idy-j][idx-i];
        b = B[j] [i];
        sum += a*b;
    }
}
C[idy][idx] = sum;
```

Outer loop of Convolution

- The neighboring threads in Y direction have overlaps in A
- If we let one thread compute two output pixels in Y direction, we can reduce the data access of A.

Convolution: thread merge



- When we load 8 pixels from A (shared memory)
  - We can do inner loop for one output pixel
  - Or two pixels
  - Three, or more
- So after we load data A from shared memory, we can keep it in the register to do more ALU computation

Code after thread merge

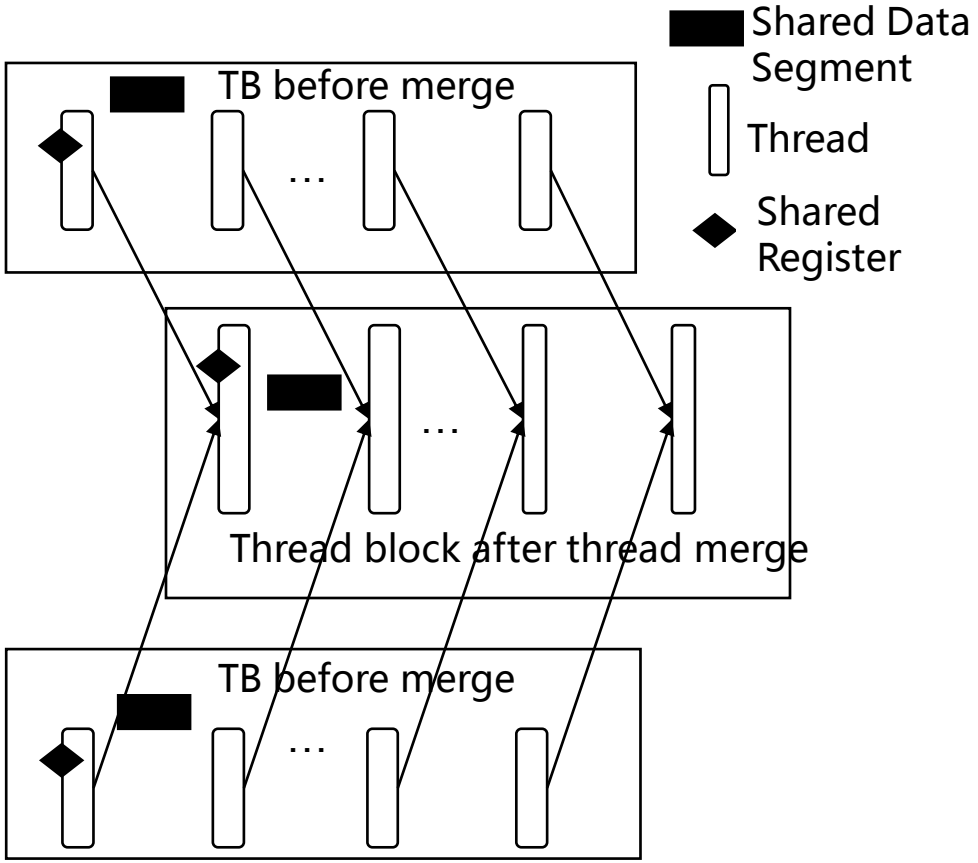
```
float sum_0 = 0; sum_1 = 0;
for (j=0; j<8; j=j+1) {
        __shared__ float shared_0[256+32];
        if (tidx<32) shared_0[tidx]=A[idy-j][idx-32];
        shared_0[tidx+32]=A[idy-j][idx];
        __syncthreads();
        for (i=0; i<8; i=i+1)
        {
                float a=shared_0[tidx+32];
                float b_0=B[j][i];
                float b_1=B[j+1][i];            // we also compute another output pixel
                sum_0+=(a*b_0);                 // code for boundary check is ignored
                sum_1+=(a*b_1);
        }
         __syncthreads();
}
C[2*idy][idx] = sum; C[2*idy+1][idx] = sum;
```

- One thread computes two output pixels

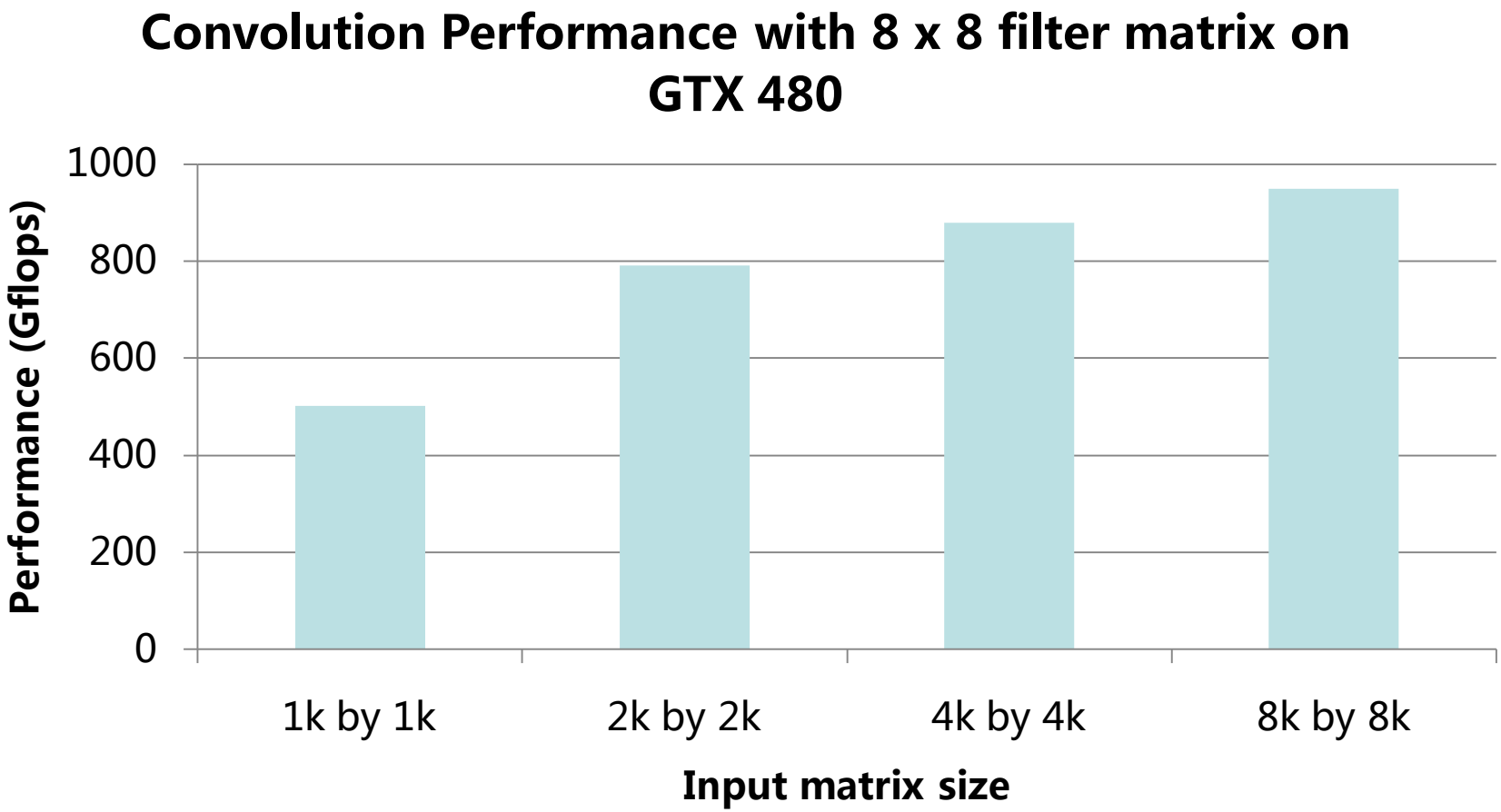
Thread merge

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

- Advantage :
Data can be in the register or shared memory
- Disadvantage :
Increase the register pressure for single thread



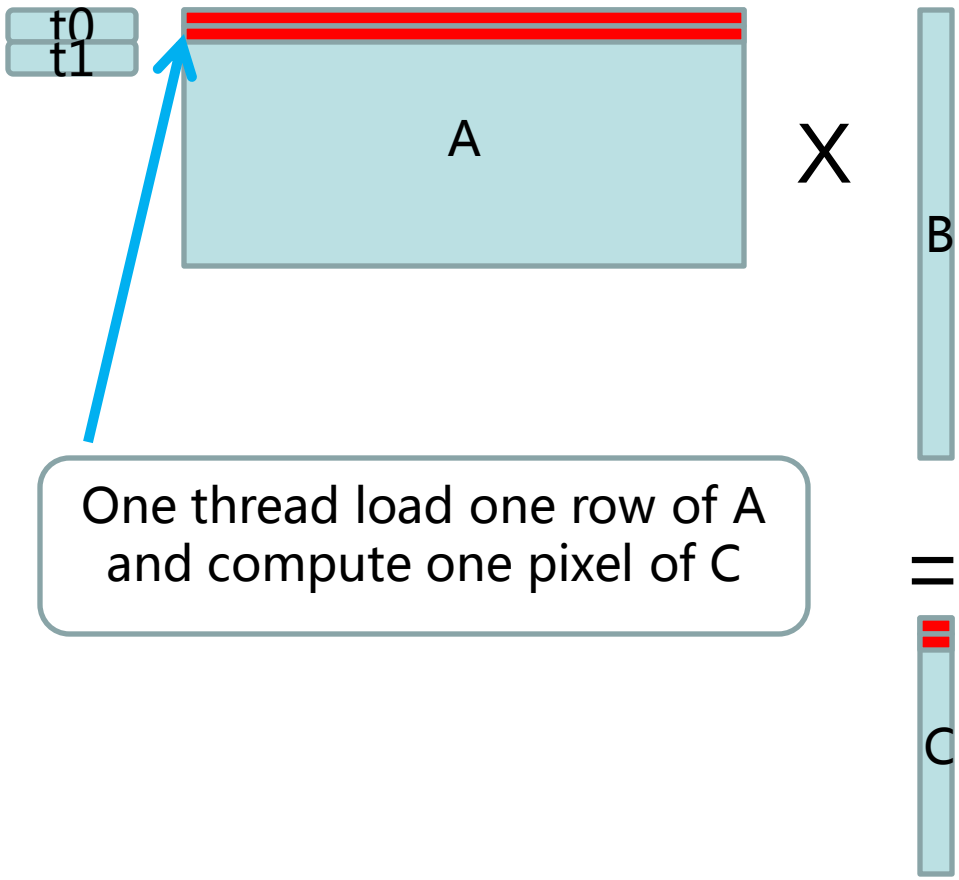
Improve memory reuse by merging threads from neighboring thread blocks.



- 70% of theoretic computation power (1.35Tflops) of GTX 480

128 threads in one thread block and one thread computes 184 output pixels.

Case study: matrix vector multiplication

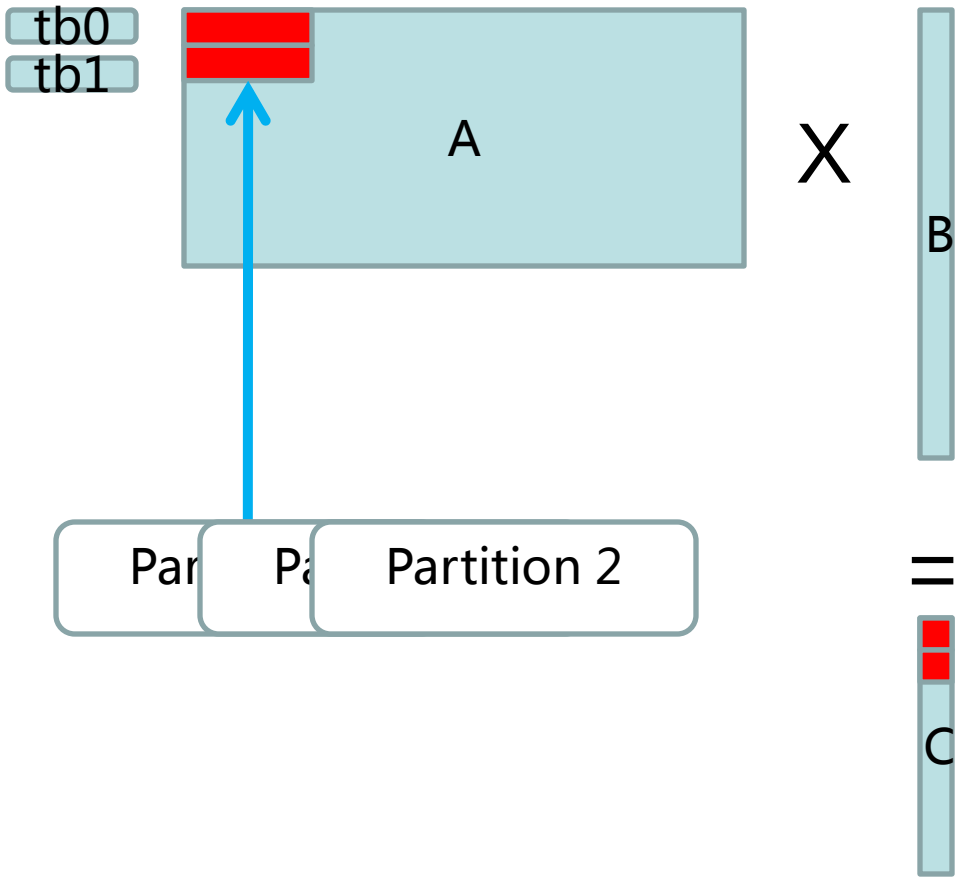


One thread load one row of A and compute one pixel of C

```
float sum = 0;
for (i=0; i<w; i=i+1) {
    float a;
    float b;
    a = A[idx][i];
    b = B[i];
    sum += a*b;
}
C[idx] = sum;
```

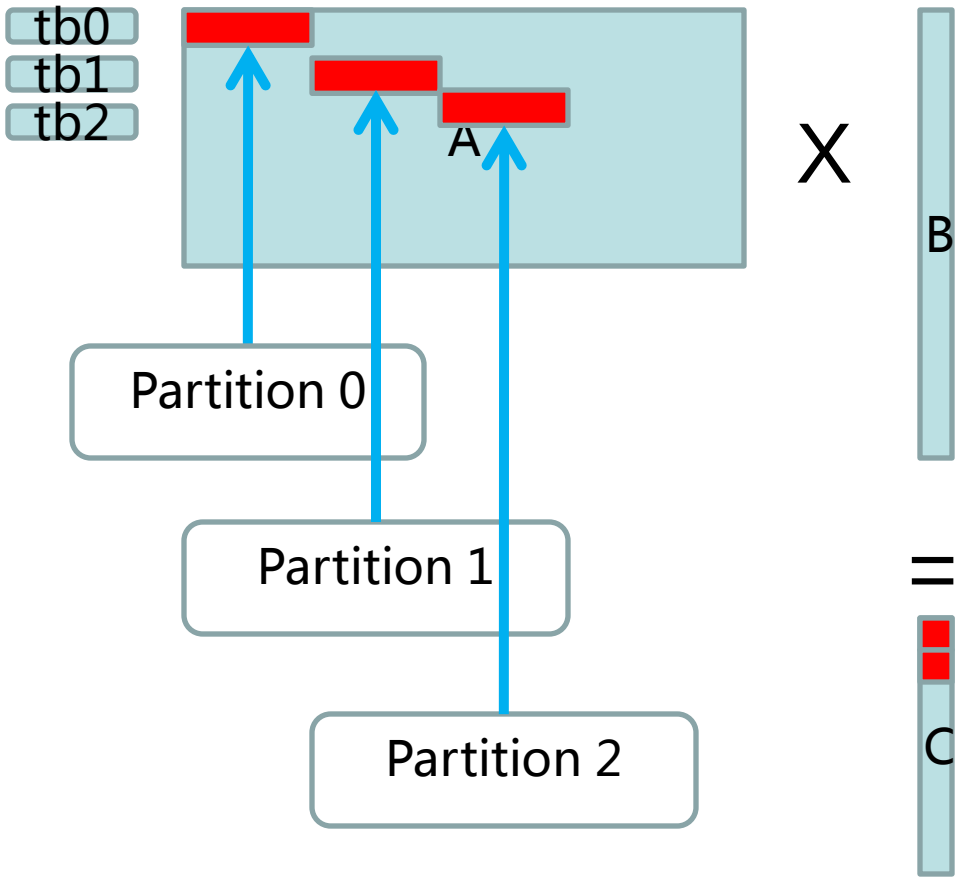
Naïve version of MV

Partition camping



- If the width of A is multiple of partition size
- All thread blocks start from same partition

Eliminating Partition camping



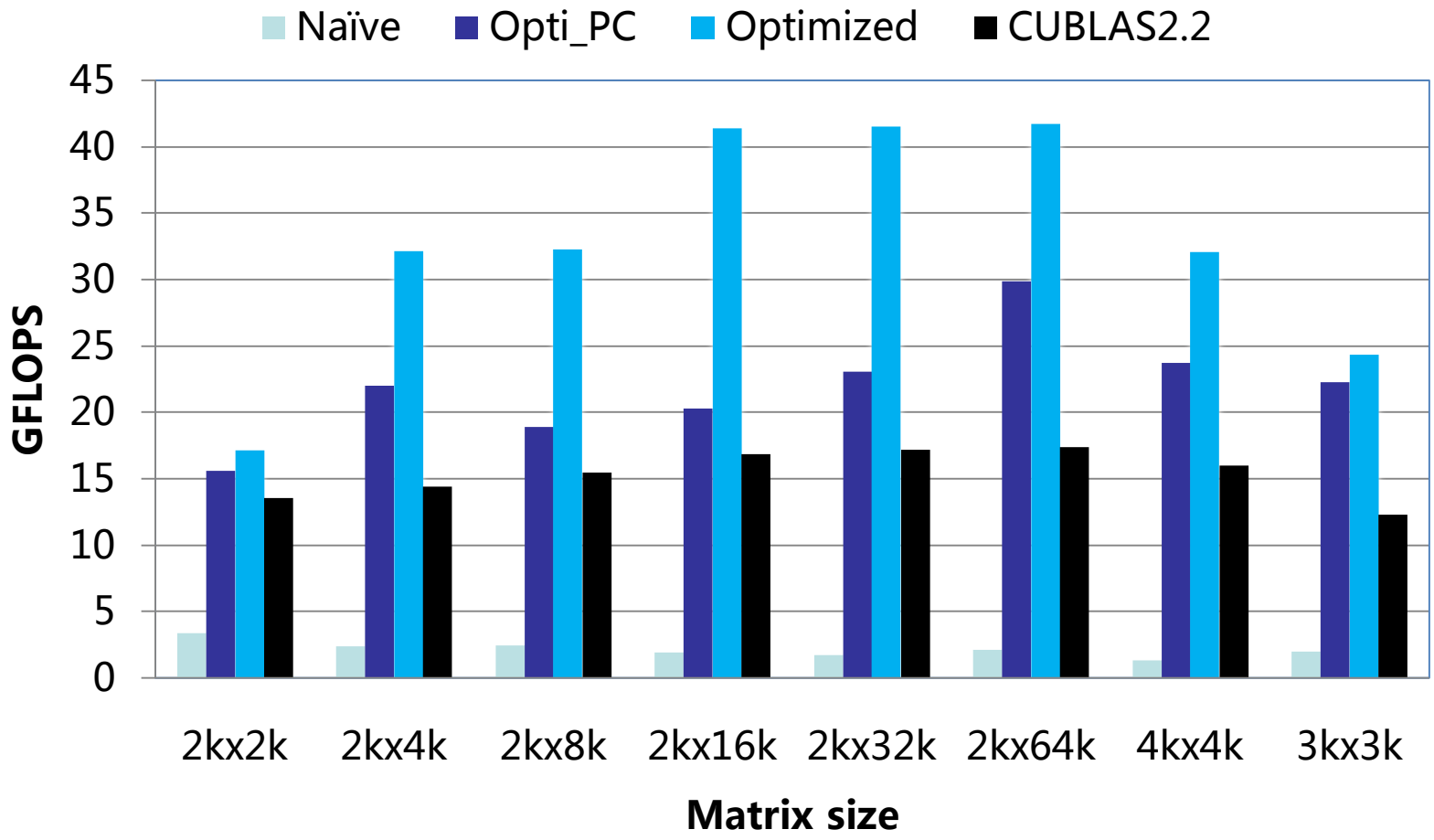
- Let different thread blocks have different start points

Code to eliminate partition camping

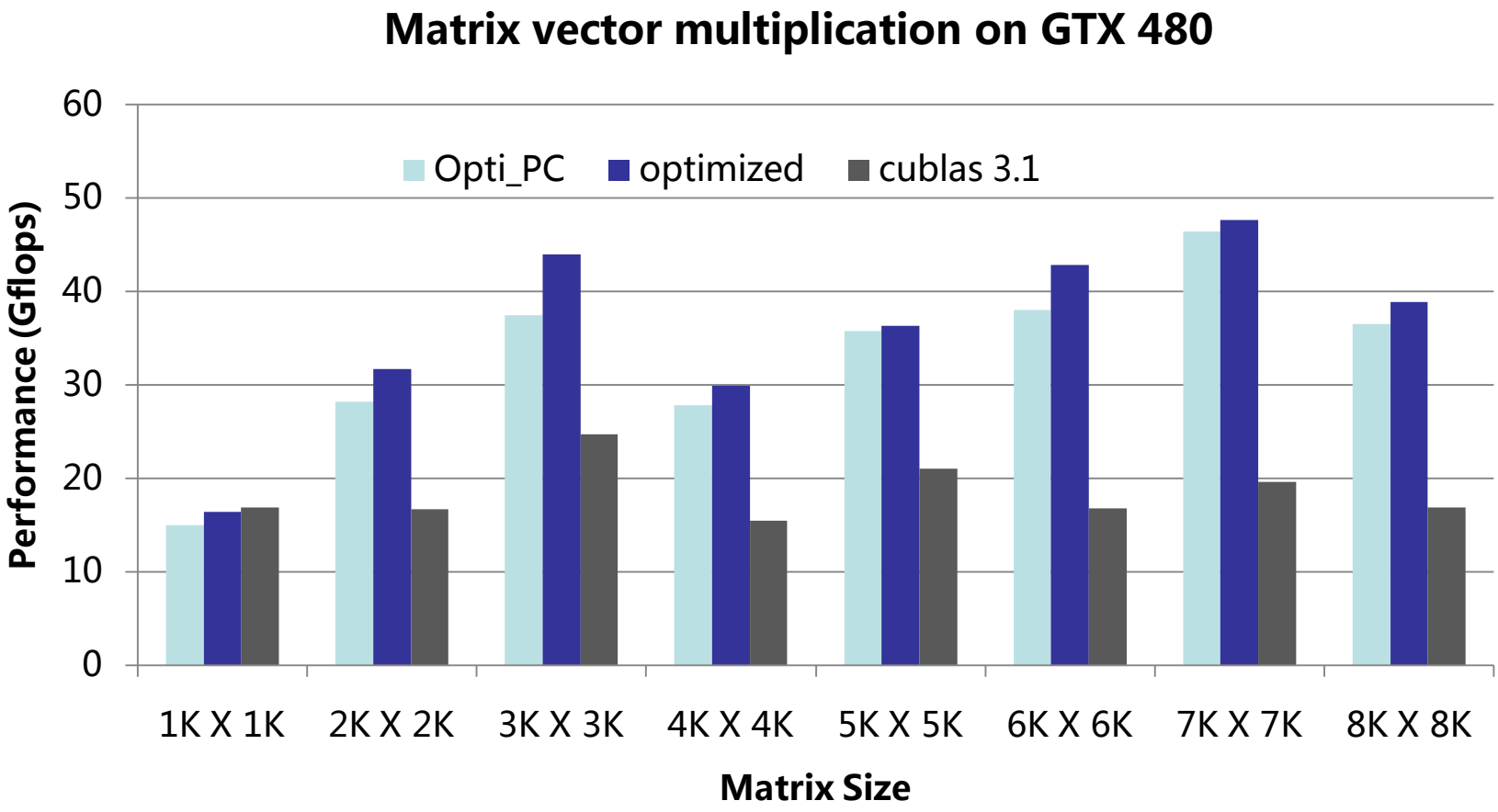
```
int start = (blockIdx.x*16);   // different start points for different thread blocks
for (i=0; i<w; i=(i+16))
{
    int k=((start+i)%w);
    float a;
    float b;
    a = A[idx][k];
    b = B[k];
    sum += a*b;
}
C[idx]=sum;
```

- The un-optimized kernel is used to illustrate the code to remove partition camping
- Optimized kernel has 32 threads in one thread block and uses shared memory to avoid un-coalesced memory access

Matrix vector multiplication on GTX 280

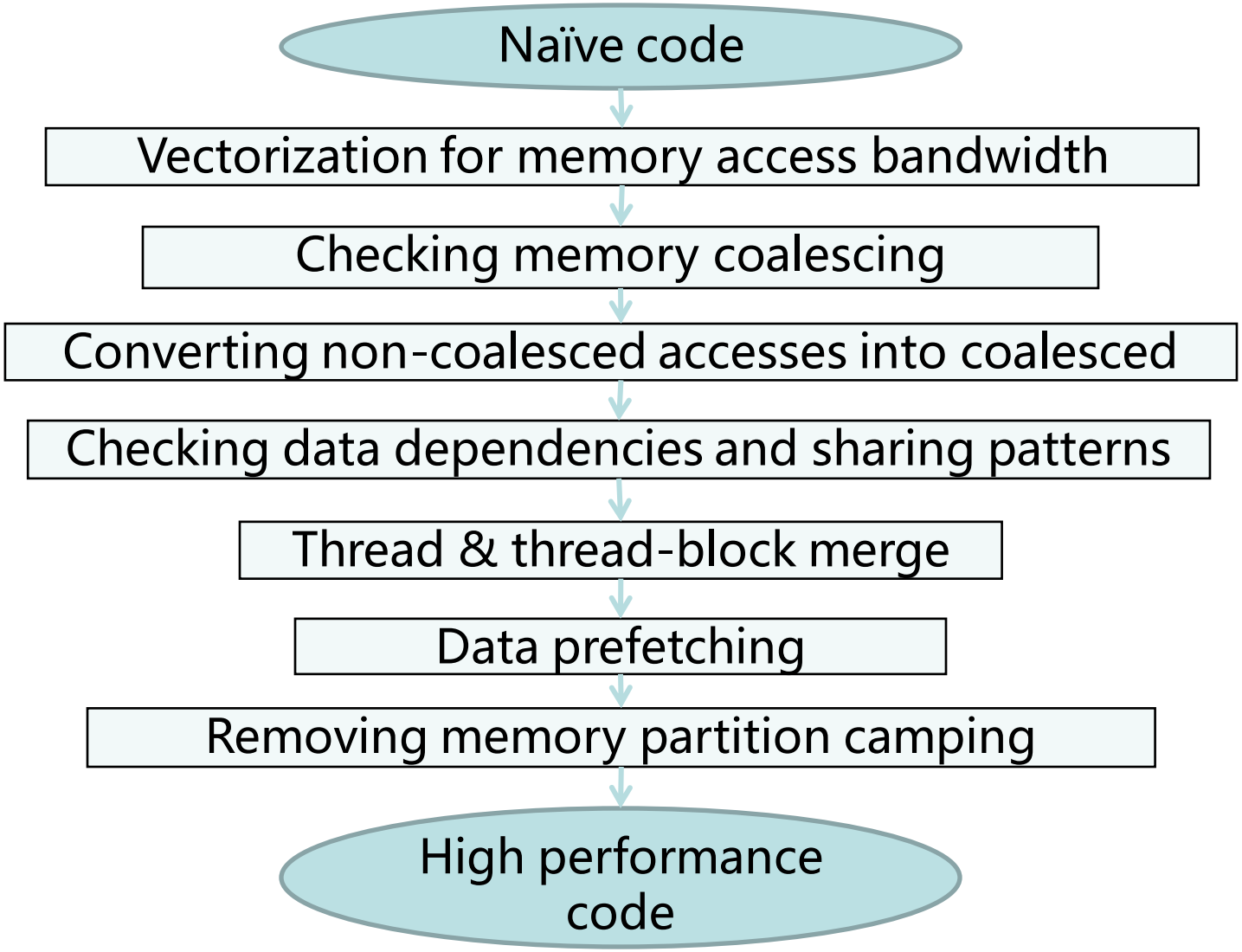


Opti_PC : the optimized kernel without partition camping elimination



- Opti_PC : the optimized kernel without partition camping elimination.
- Partition camping elimination benefits 3K and 6K more because GTX 480 has 6 partitions.

Compiling for High Performance GPGPU Code: http://code.google.com/p/gpgpucompiler/



Outline

- Hardware abstraction
- A systematic approach to developing high performance GPGPU programs
- Optimization techniques with case studies
  - Coalesced memory accesses
  - Data reuse through thread (block) merge
  - Eliminating partition conflicts
  - Leveraging constant cache
- Conclusions

Leveraging constant cache (GTX 480)

r0 = r1 + r2*shared[k];
One second 1T/4bytes = 0.25T float
0.25T * 2flops = 500 GFlop

- Register
  - Benefit: fastest, no l
  - Limitation: no sharing between threads
- Constant cache
  - Benefit: up to 2TBytes/S
  - Limitation: 64kB const memory on GTX 480, sequential broadcast access
- Shared memory
  - Benefit: sharing in block with index
  - Limitation: up to 1TBytes/S
- Texture cache
  - Benefit: 2D cache automatically
  - Limitation: up to 334GBytes/S

Case study: Matrix Multiplication with Constant memory

- C = A * B

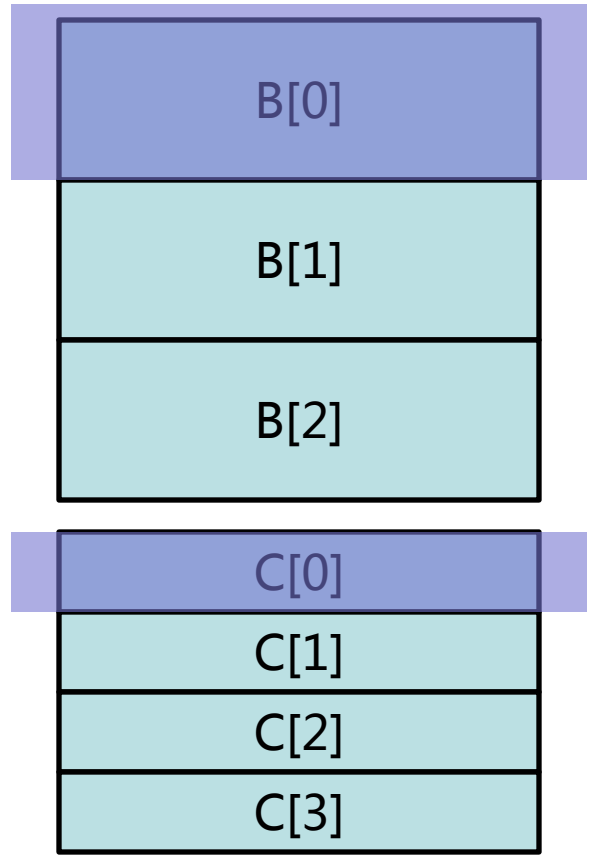
```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication (one output per thread)

- All threads with the same idy access input A same location sequentially.
  - From A[idy][0] to A[idy][w-1]
- How about we put A into constant memory

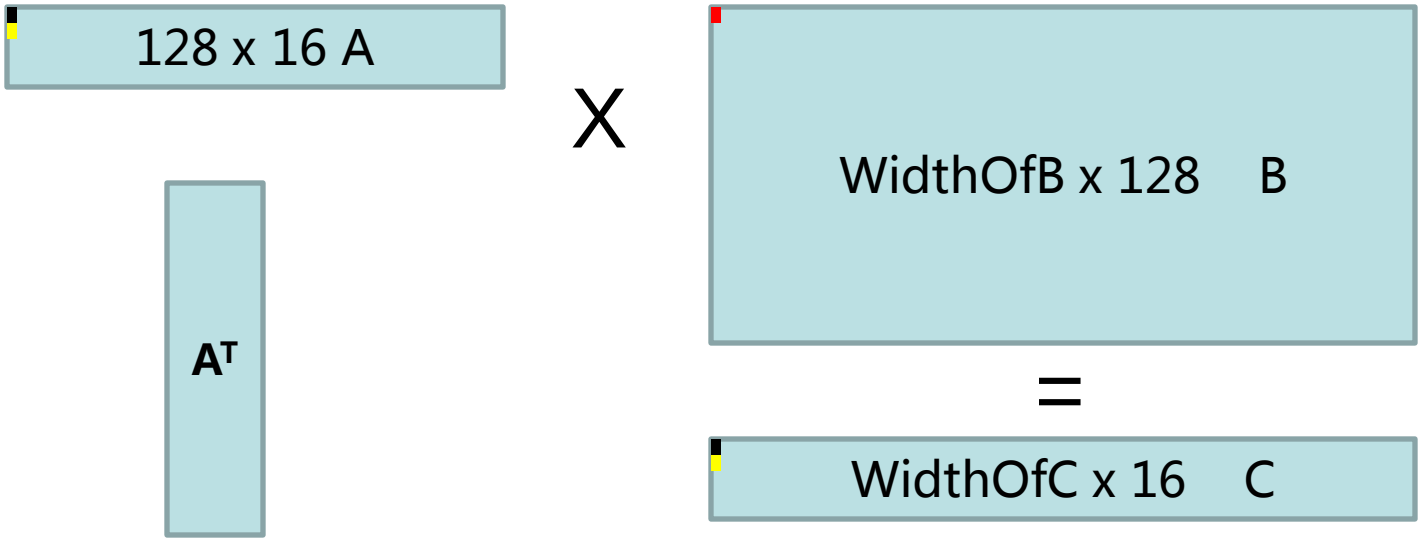
Matrix Multiplication (Tiled)

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][0]



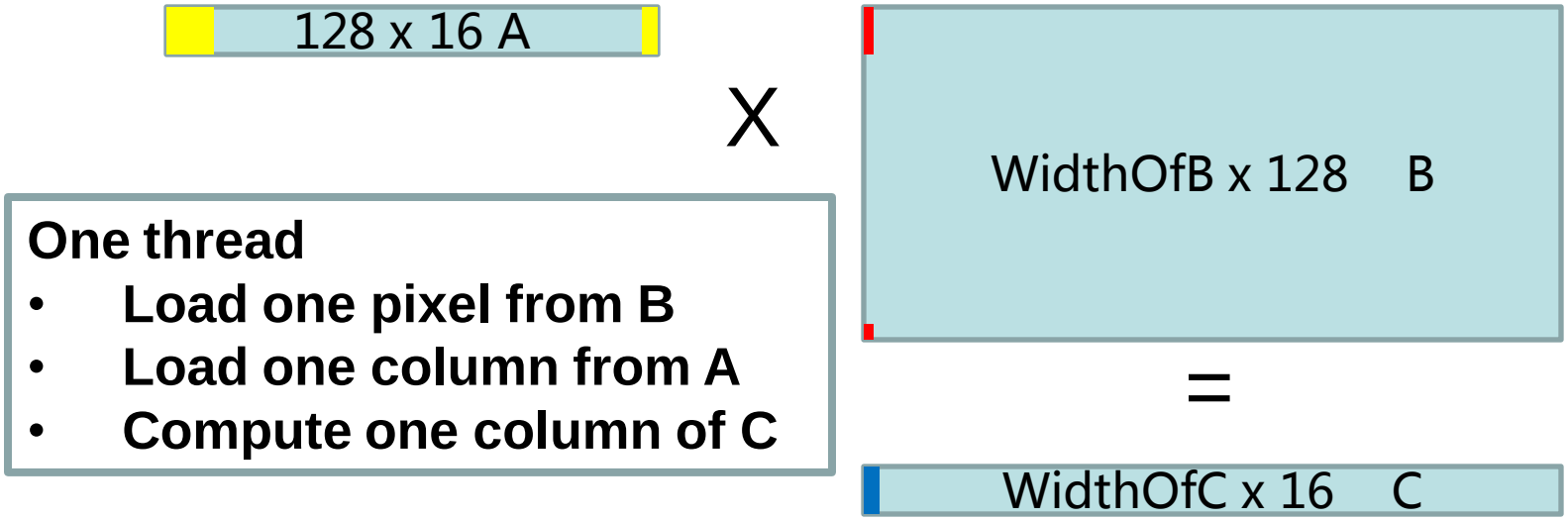
- C[i] = A[i][0]*B[0] + A[i][1]*B[1]+A[i][2]*B[2]
- C[0], C[1], C[2], C[3] can be computed concurrently
- Let’ s put A[i][j] into constant memory

Efficient constant memory accesses



- When we load one pixel from B
- We can compute one output pixel
- Or two, up to 16 so that we can use more computation to overlap memory access B
- But column access in constant memory is not efficient

Matrix Multiplication



One thread

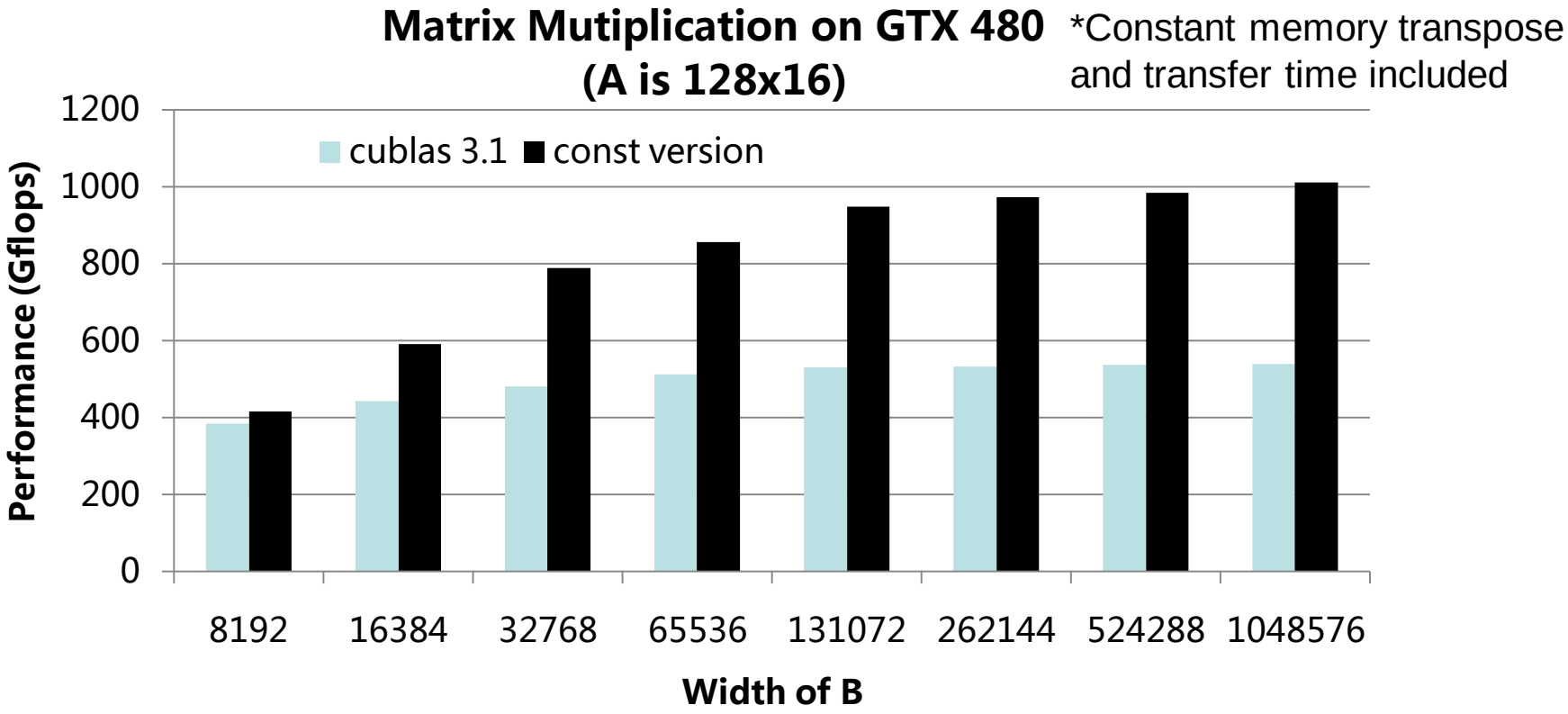
- **Load one pixel from B**
- **Load one column from A**
- **Compute one column of C**

- A is 128 x 16
  - We can put A into const memory (column major)
  - Load a float from B, we can do 16 mad to overlap the memory request of B
  - The width of B determine the overall thread number.

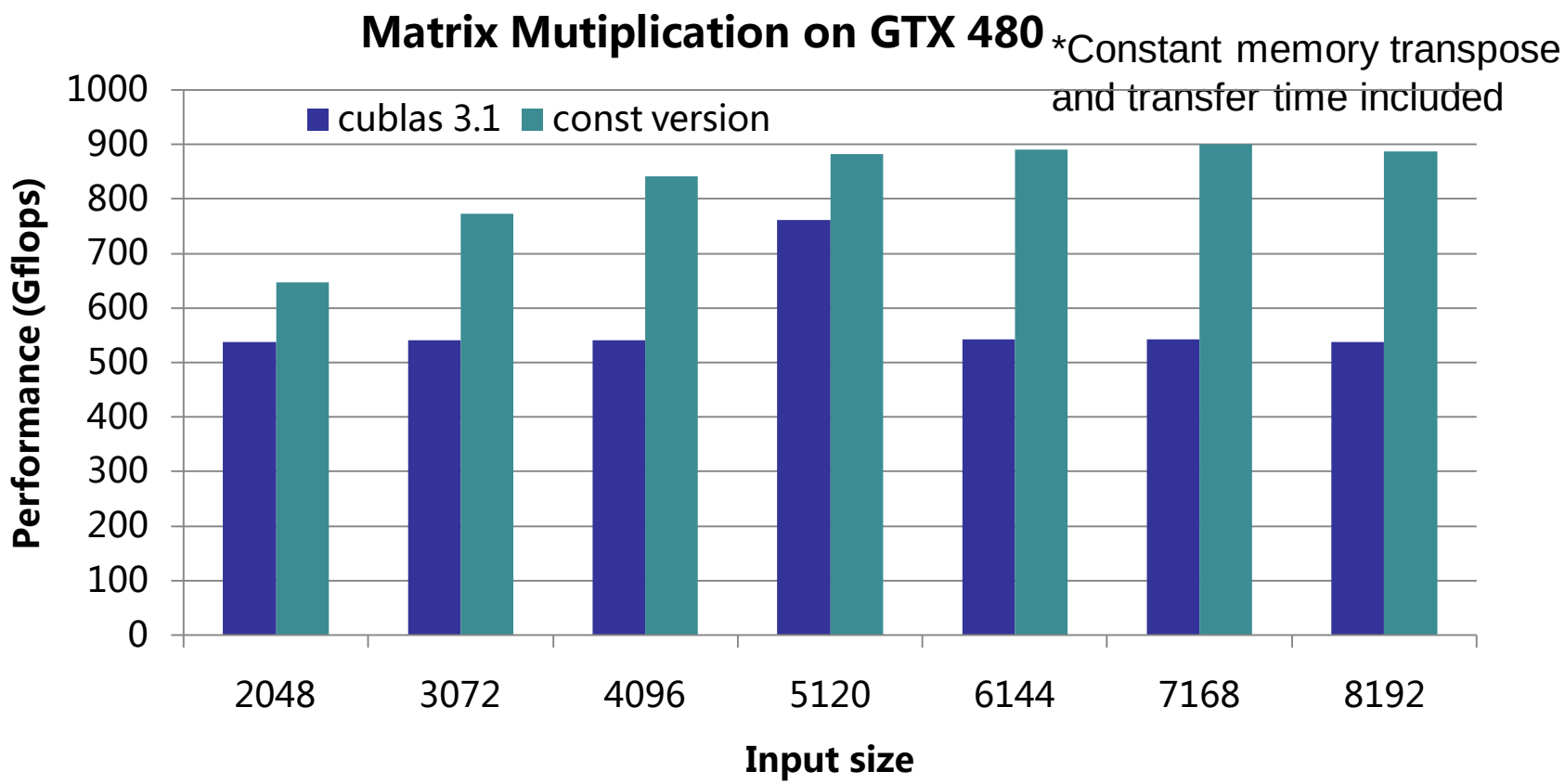
Kernel code when A is 128 x 16

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
float sum[16];
for (int i=0; i<128; i++) {
        float b = B[i] [idx];
        for (int j=0; j<16; j++) {
                sum[j] += b*CONSTA_A[i*16+j];
              // A is in constant memory
        }
}
for (int j=0; j<16; j++) {
        C[j] [idx] = sum[j];
}
```

- Each thread computes 16 output pixels
- Thread block size: 256



- Up to 1.8 times Speedup on CUBLAS 3.1
- 75% of theoretical computation power (1.35Tflops) of GTX 480



- Width, Height of A and B are the same as the input size
- Up to 1.65X speedups over CUBLAS 3.1
- 67% of theoretical computation power (1.35Tflops) of GTX 480

Conclusion

- A systematic way to optimize GPGPU programs
  - Naïve kernel based on simplified hardware abstraction
  - Optimizations
    - Coalesced memory accesses
    - Data reuse through thread (block) merge
    - Eliminating partition conflicts
    - Leveraging different types of caches
- We implement a source to source compiler to perform the optimizations automatically.

http://code.google.com/p/gpgpucompiler/