GPU Computing with CUDA
Lab 4 - Efficient AA^T

Christopher Cooper
Boston University

August, 2011
UTFSM, Valparaíso, Chile

Objectives

- Implement an efficient AA^T multiplication considering
  - Tiling
  - Coalesced memory accesses
- One thread will perform one dot product for one element of the resulting matrix

Efficient AA^T

- Naive approach

```
float sum = 0.0f;
for (int k=0; k<W; k++)
  sum += A.elements[j*MAX+k]*A.elements[i*MAX+k];
```

- Problems

  - Redundant global memory accesses

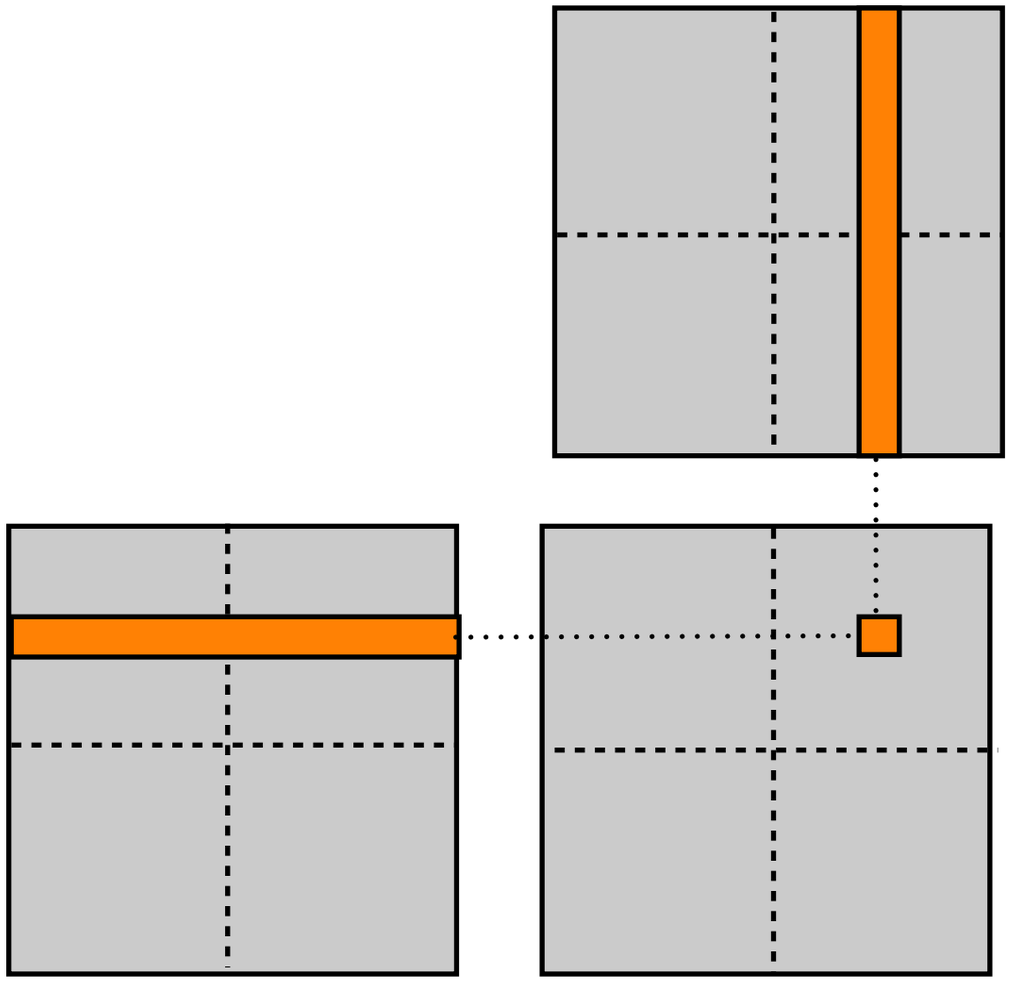
  - Uncoalesced reads for transpose

Efficient AA^T

- Use shared memory
  - Reduces redundant reads
  - Transposing in shared memory has no coalescing penalty
  - Try making your reads to shared memory coalesced
- Be aware that the transpose of a whole matrix can be obtained by transposing in shared memory inside the block and then transposing the whole blocks

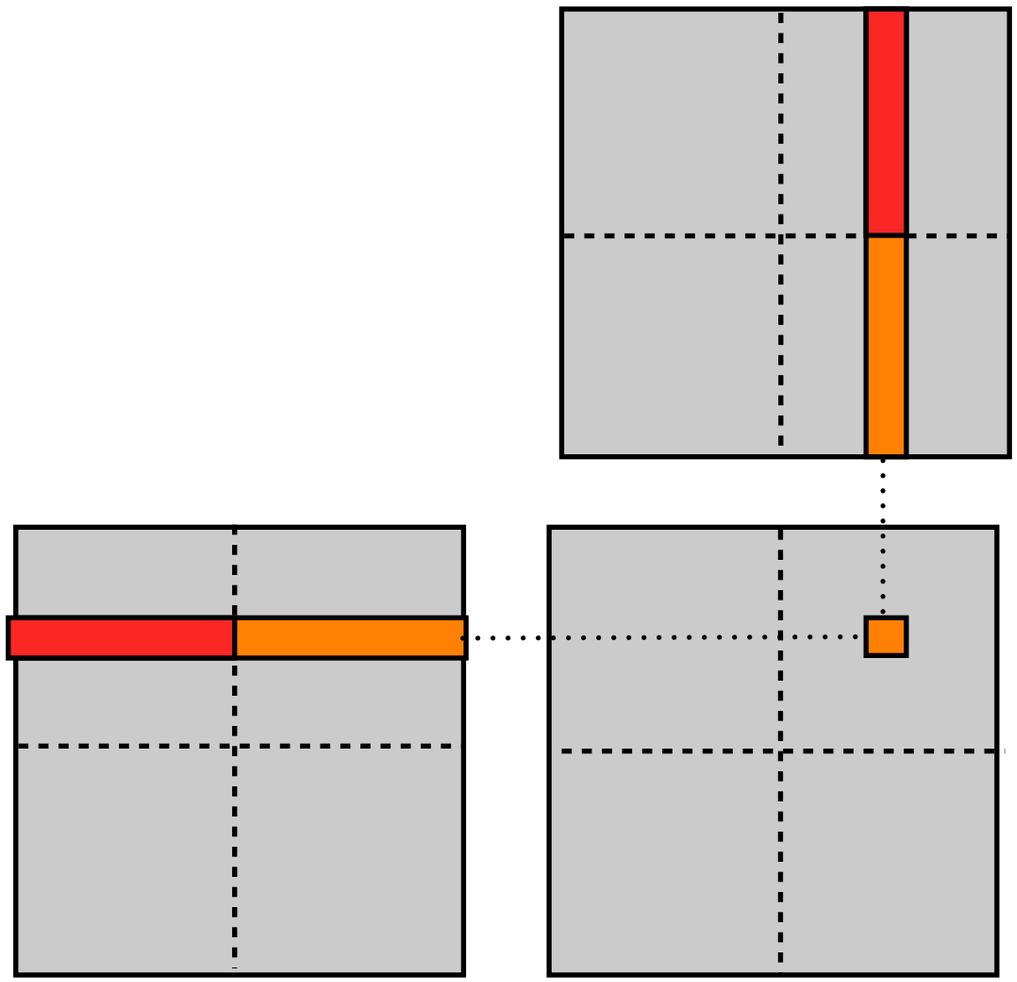
Efficient AA^T

- Conventional matrix matrix



Efficient AA^T

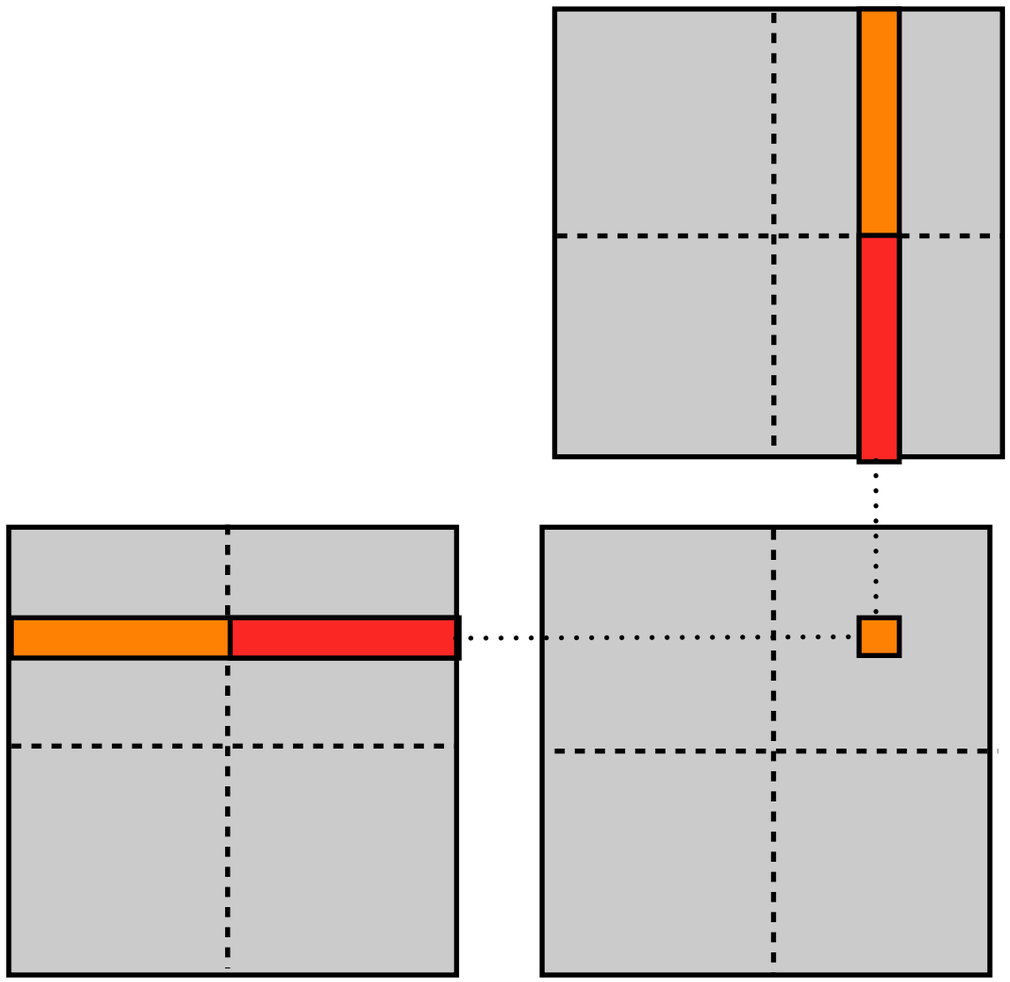
- Conventional matrix matrix



Iteration 1

Efficient AA^T

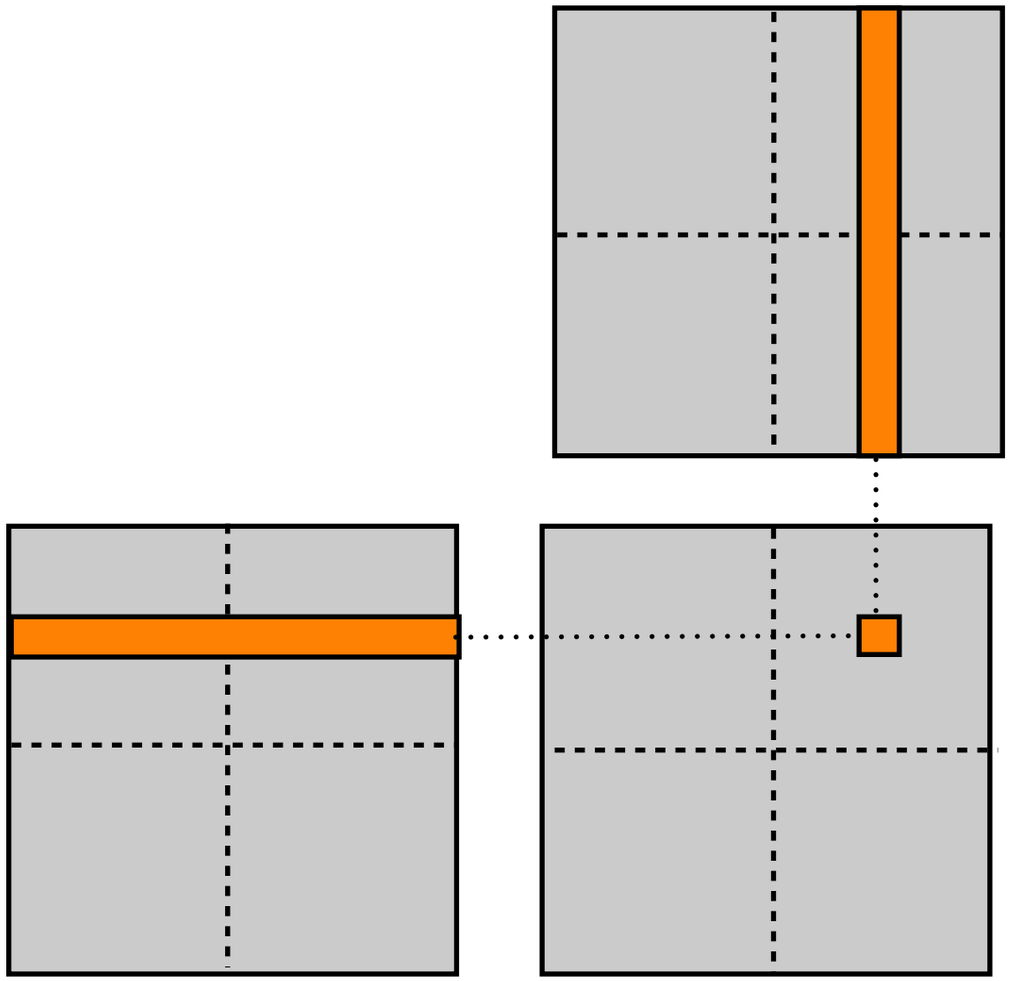
- Conventional matrix matrix



Iteration 2

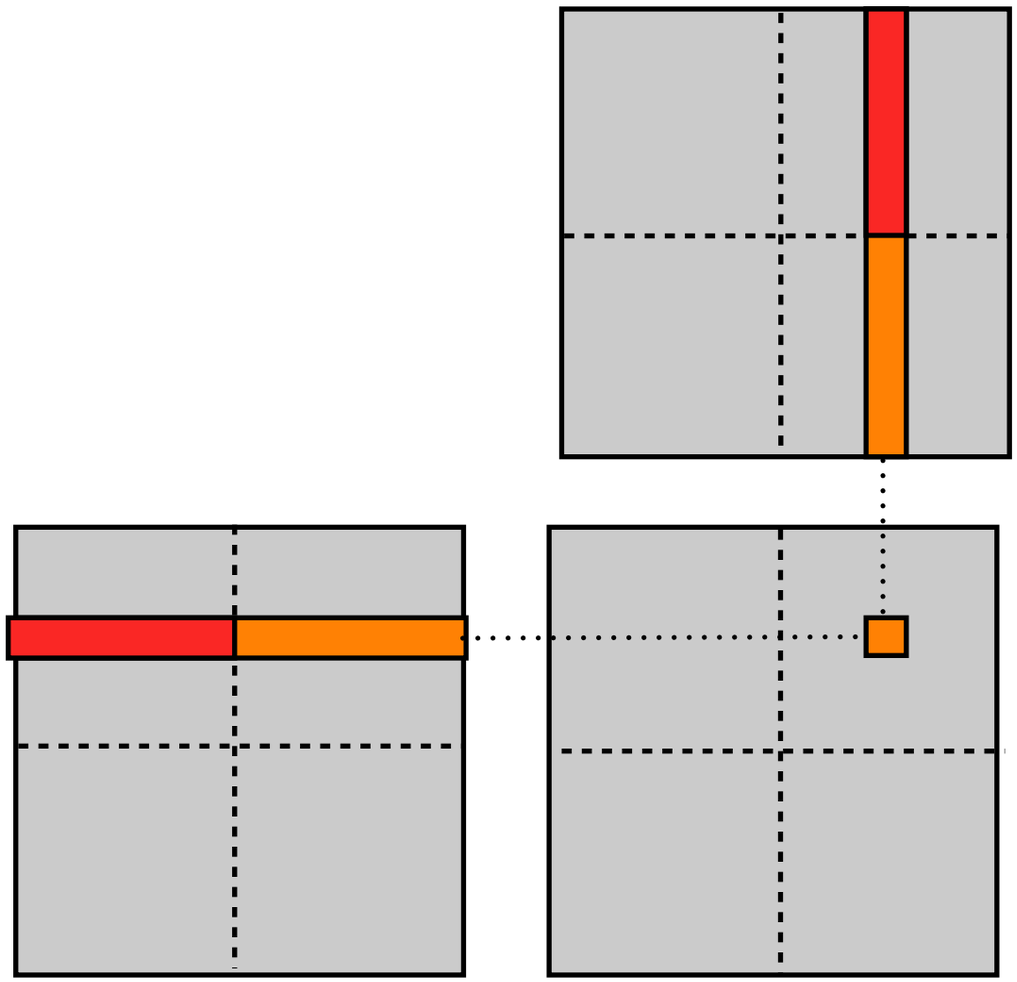
Efficient AA^T

- Conventional matrix matrix



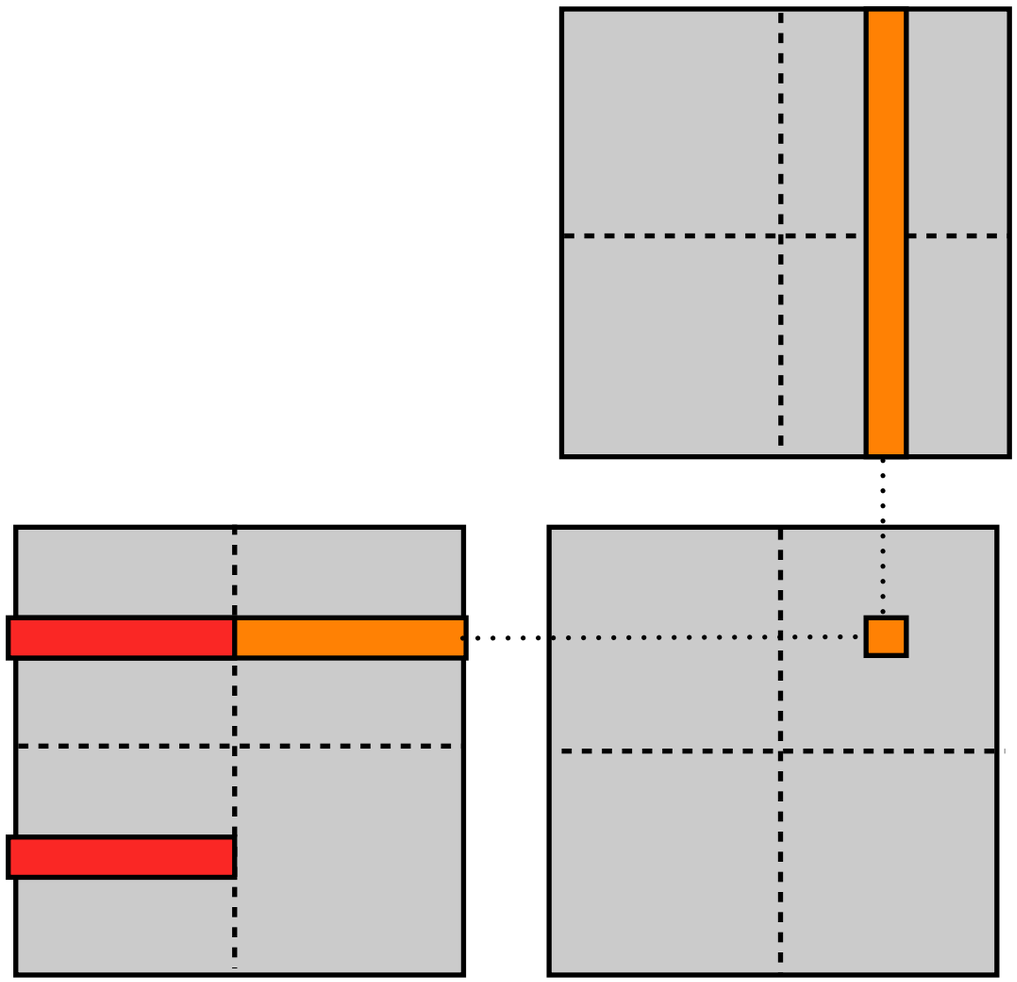
Efficient AA^T

- AA^T case



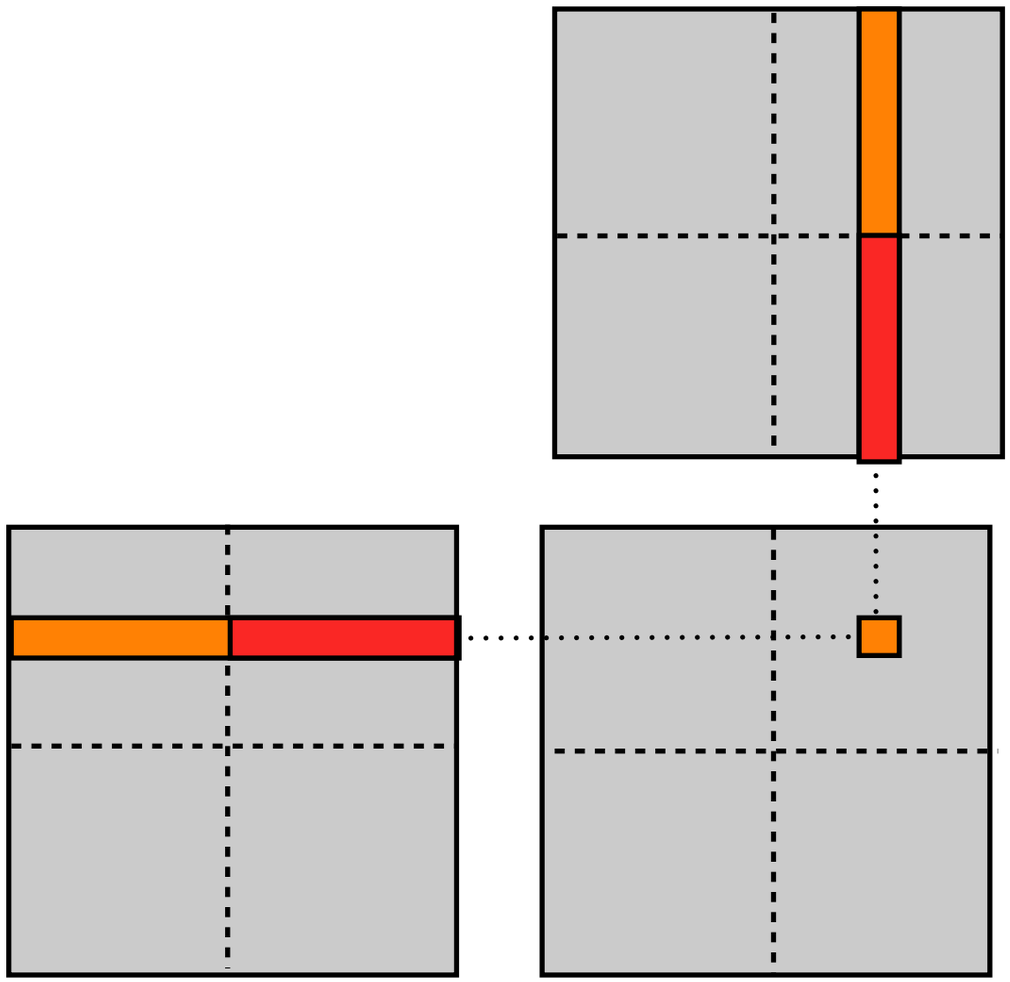
Efficient AA^T

- AA^T case



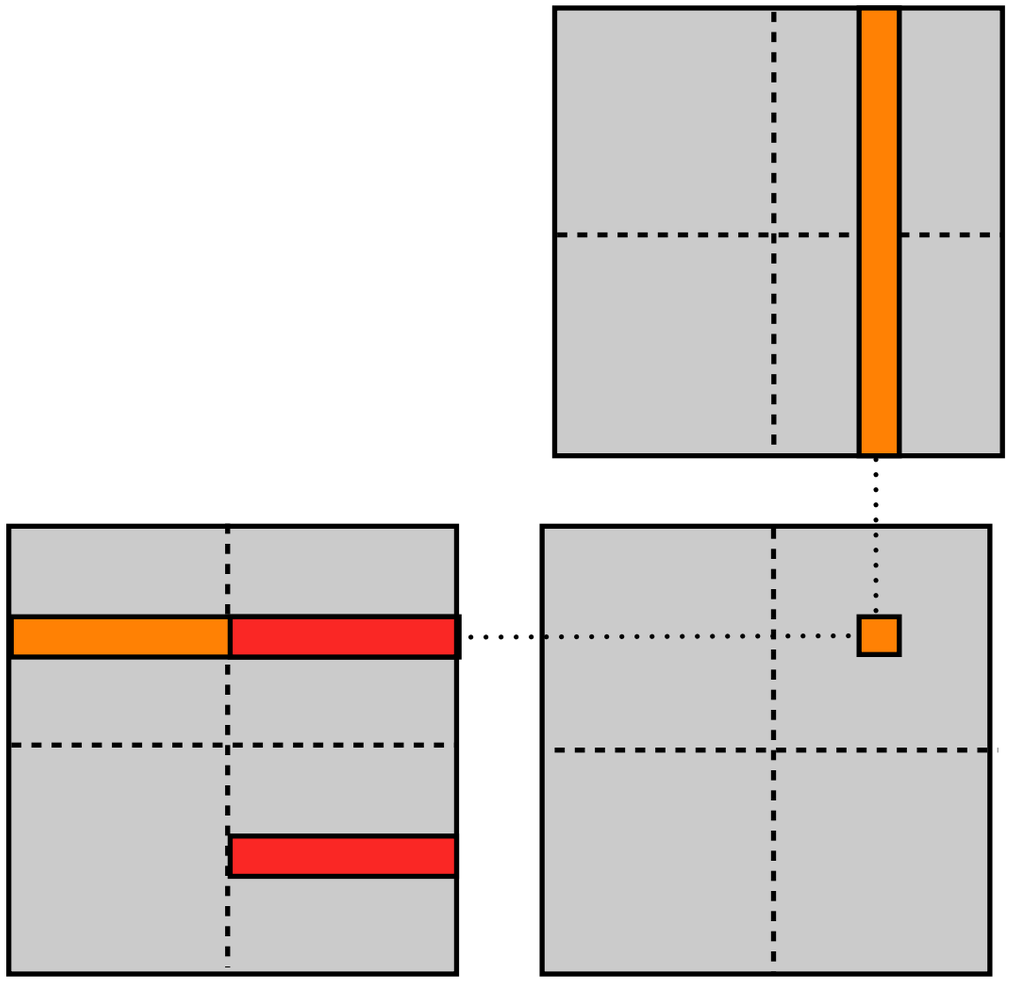
Efficient AA^T

- AA^T case



Efficient AA^T

- AA^T case



Efficient AA^T

- My results (2048x2048)
  - Naive
    - 2*2048+1 = 4097 loads per thread
    - 2*2048 = 4096 operations per thread
    - Kernel time = 2.11s (C2050) 8.1s (GTX295)

Stops being bandwidth limited

  - Tiled
    - 2*2048/16 + 1 = 257 loads per thread
    - 2*2048 = 4096 operations
    - Kernel time = 0.34s (C2050) 0.577s (GTX295)