Deep Learning Optimization - GPU Architecture & CNN Acceleration

March 6 and 8, 2023

Eunhyeok Park

Class Overview

- GPU architecture & programming model
  - VS CPU architecture
  - SIMT parallel execution
  - Characteristics for optimization

- Convolutional operation implementation
  - With matrix multiplication: cuBLAS, cuDNN
  - Winograd convolution for 3x3 acceleration
  - Low/mixed-precision acceleration with TensorCore

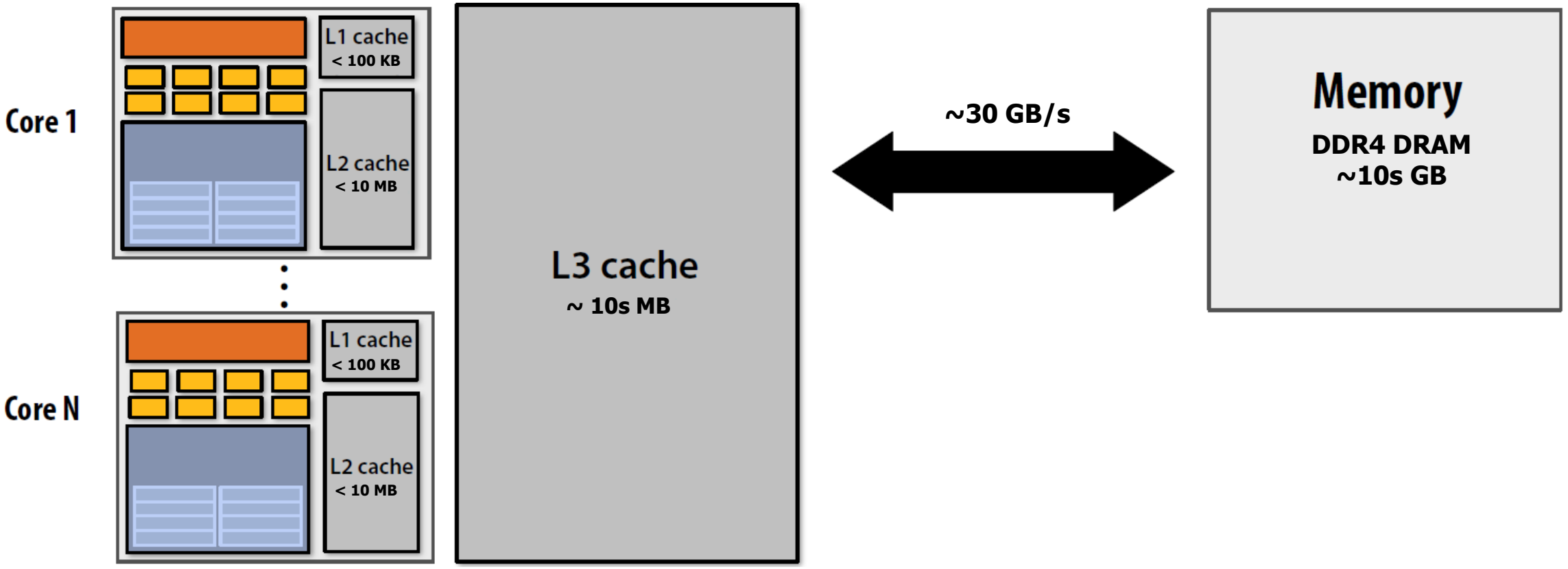
CPU vs GPU?

- https://www.youtube.com/watch?v=-P28LKWTzrI

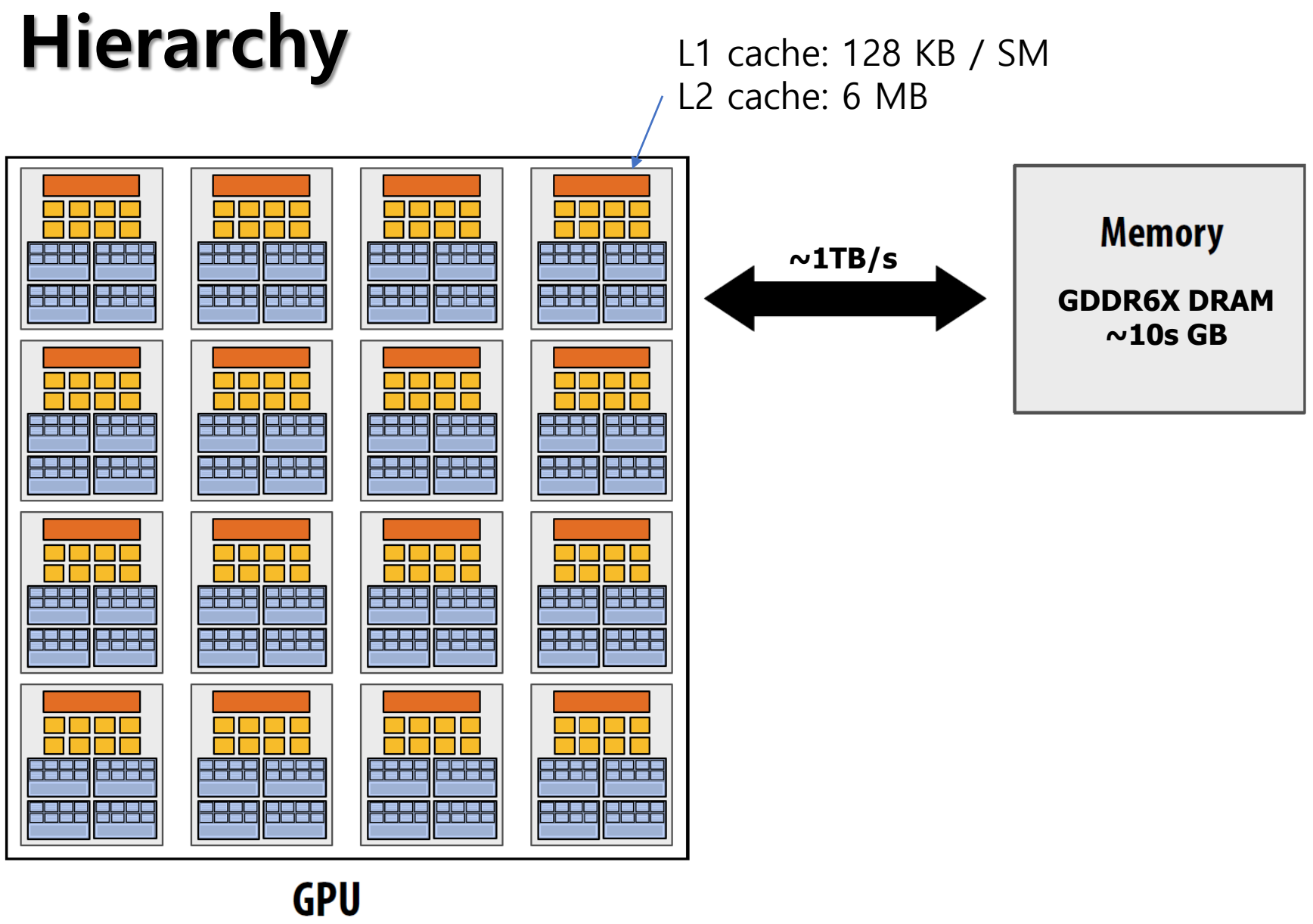
- CPU
  - Has 10s of extremely fast cores
  - Optimized for sequential execution
  - Exploit large cache to minimize latency

- GPU
  - Has 1000s of moderately fast core
  - Optimized for data-parallel execution
  - Designed to hide latency / maximize throughput

CPU Hierarchy



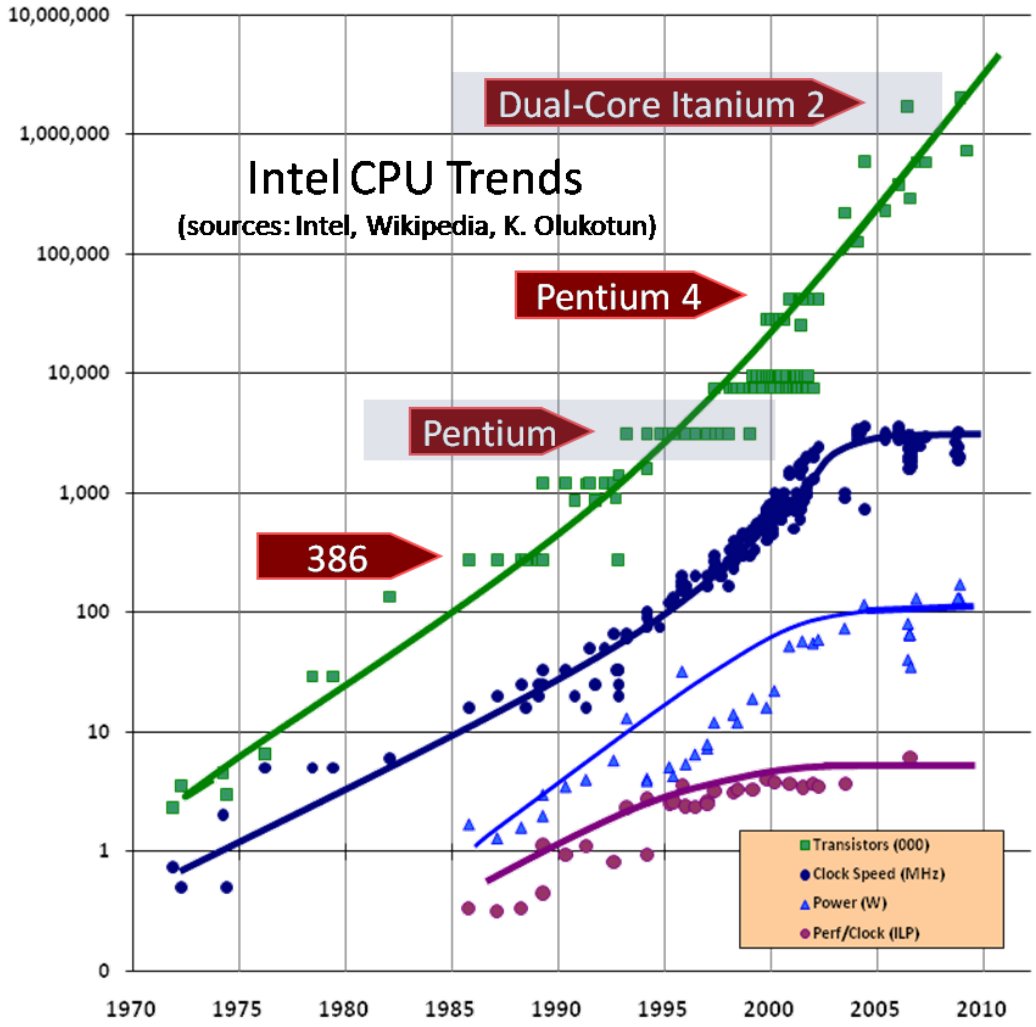
GPU Hierarchy



Why GPU? - 1

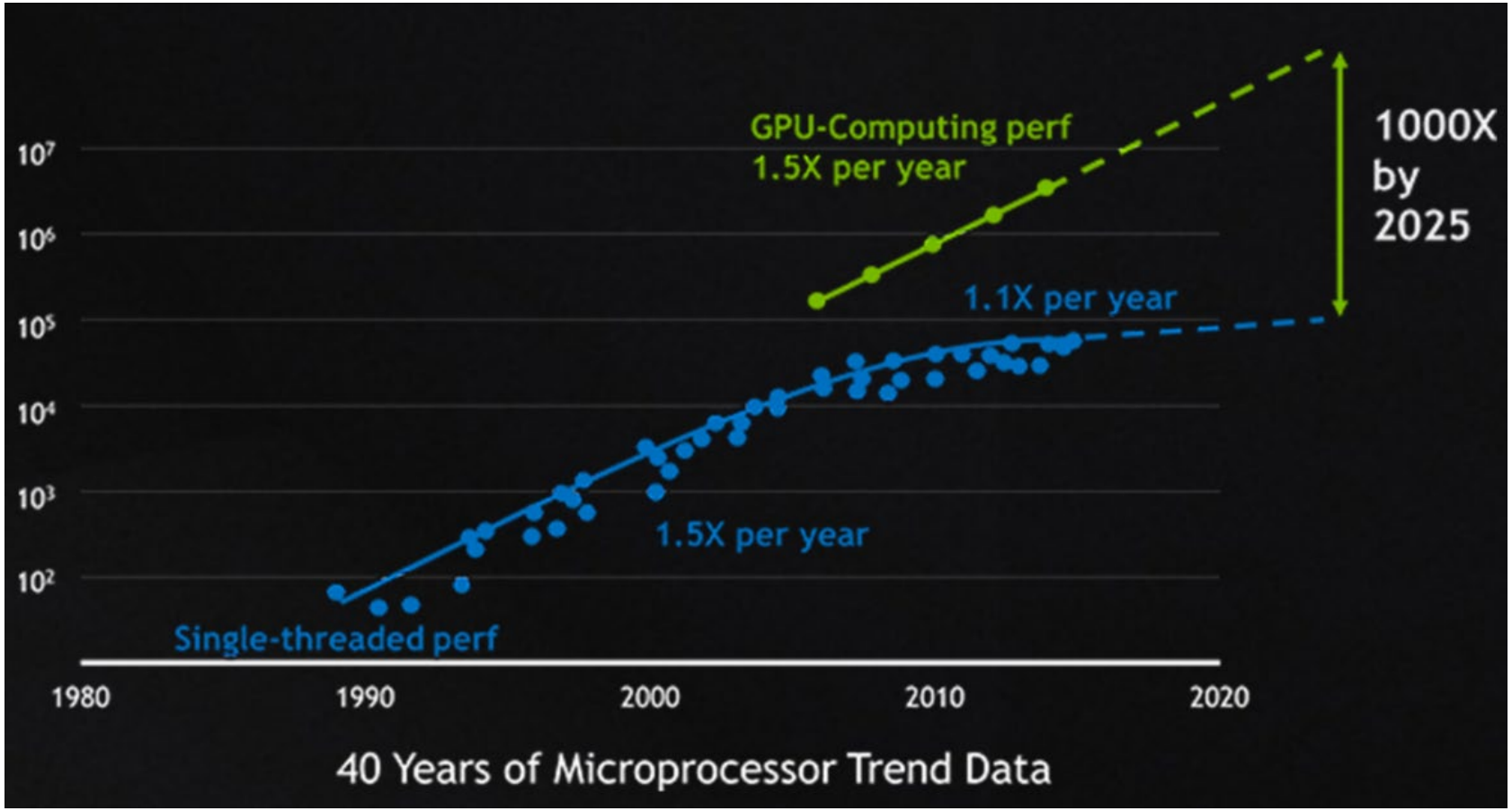
- Before early of 2010
  - Exponential performance improvement
  - Technology scaling
  - Instruction-level parallelism
    - Superscalar execution

- Now
  - Limitation of technology scaling
  - Saturated architecture improvement
  - ~ 3 % improvement / year/core



[The Free Lunch Is Over, Herb Sutter]

Why GPU? - 2



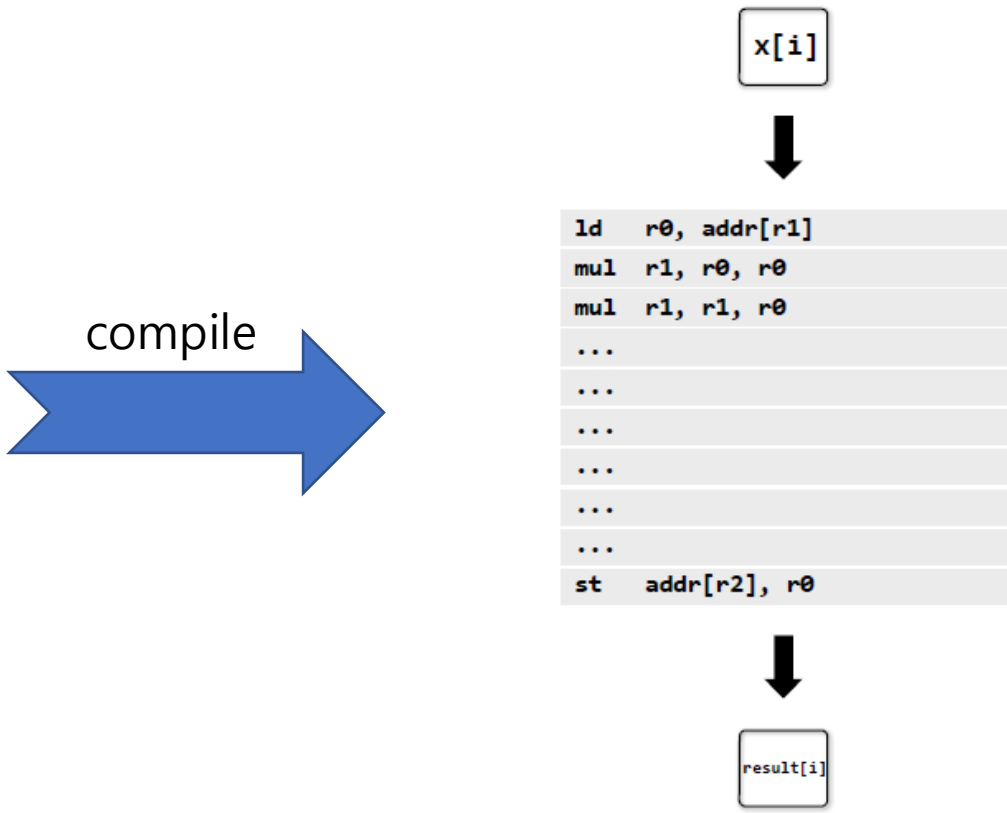
- VS CPU design
  - Increase # of simplified cores instead of increasing functionality of a single core
  - Optimize dataflow/memory hierarchy for parallel applications

[https://www.nvidia.com/en-gb/about-nvidia/ai-computing/]

Example Program

- Example: Taylor expansion for $\sin(x)$, x can be a vector or multiple scalar data

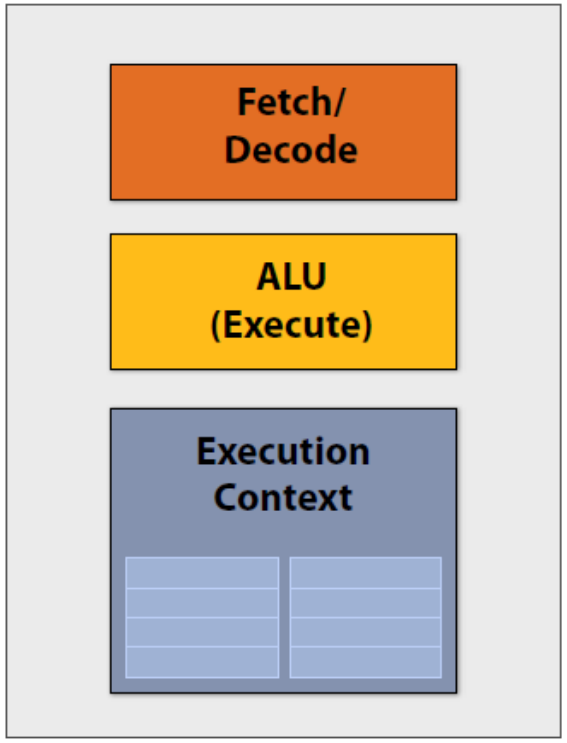
```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

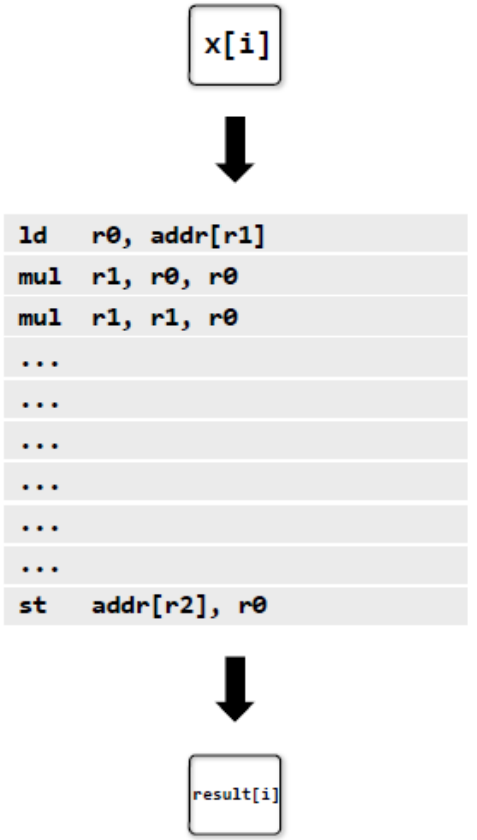


[Kayvon]

Running on Single CPU

- Basic processor: execute one instruction per cycle

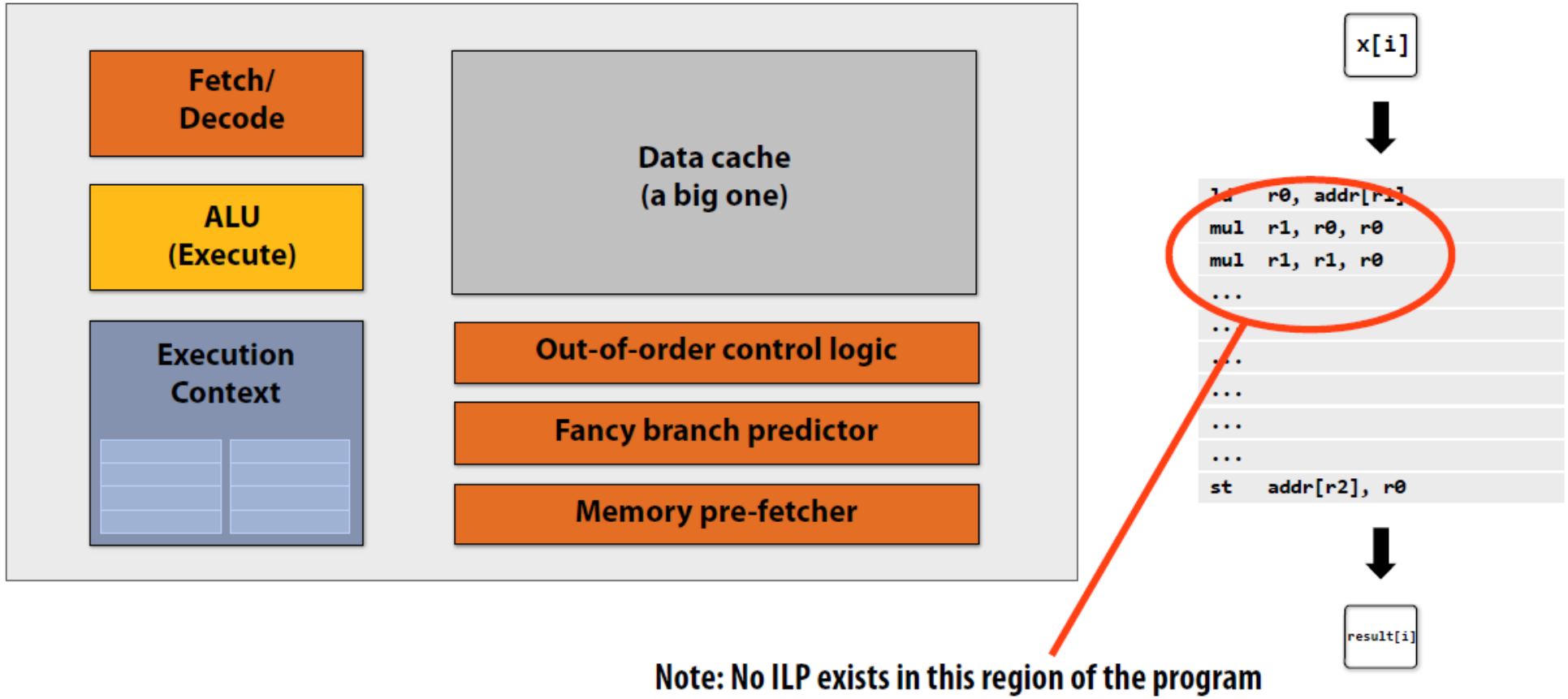




[Kayvon]

Running on Single CPU w/ Superscalar?

- Instruction level parallelism (ILP) increases hardware complexity significantly
- Is it helpful to increase the performance of this example?

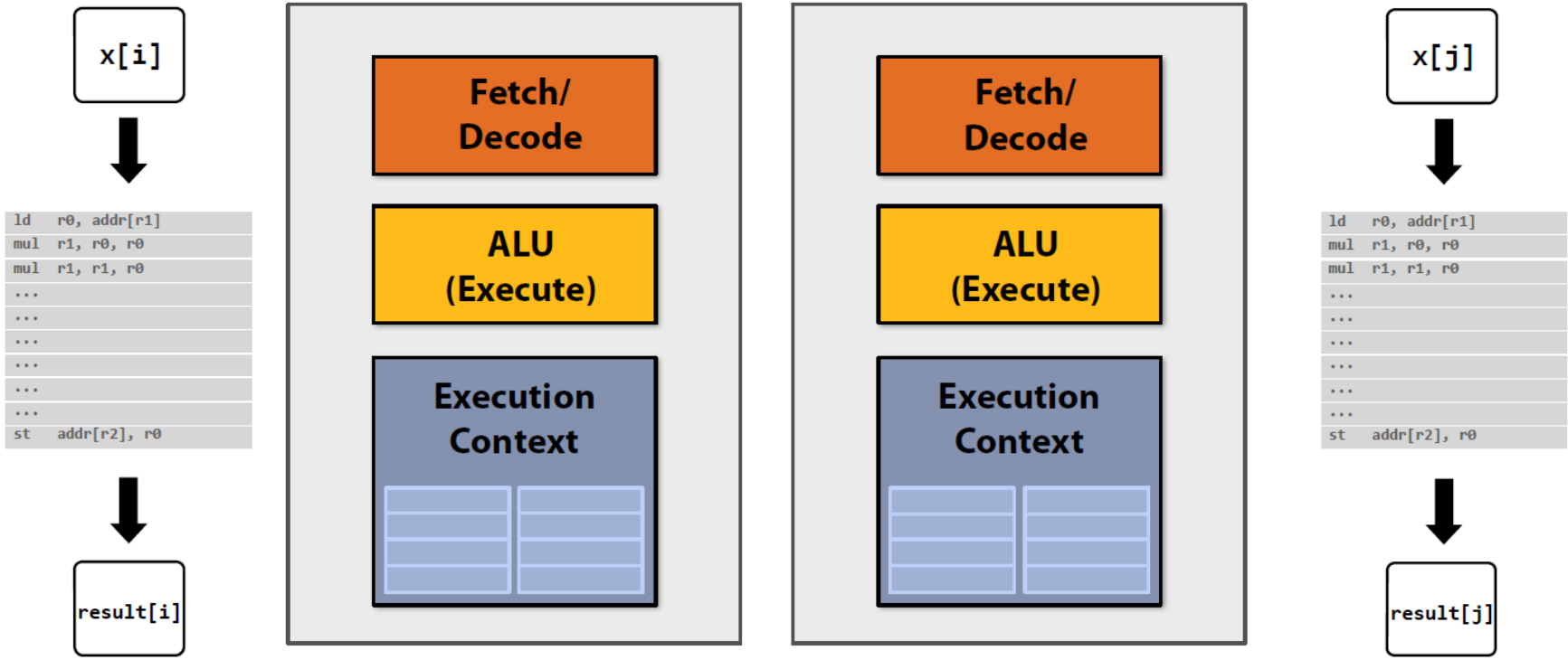


[Kayvon]

Idea of Dual Core

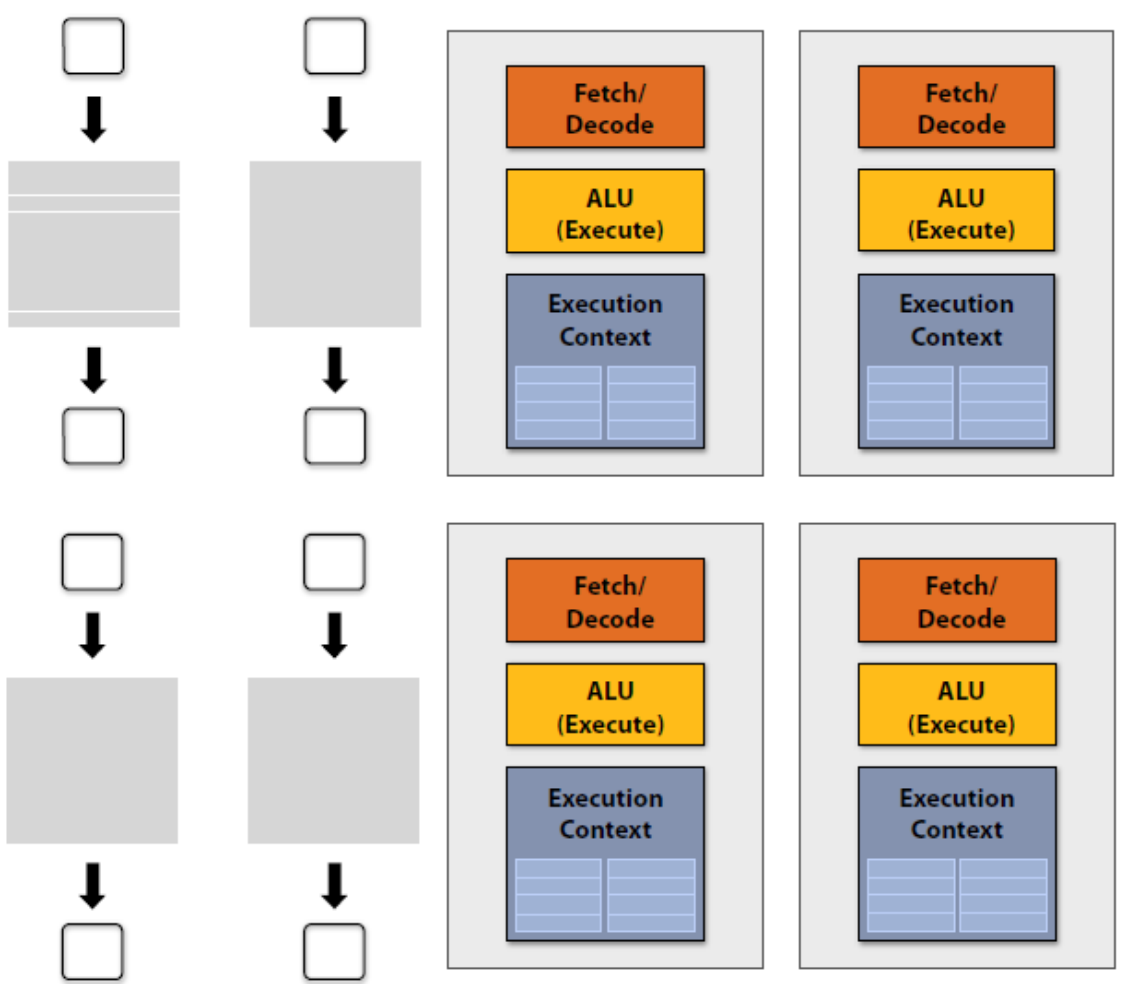
- Instead of using one powerful core, how about exploiting parallelism with multiple moderate cores?

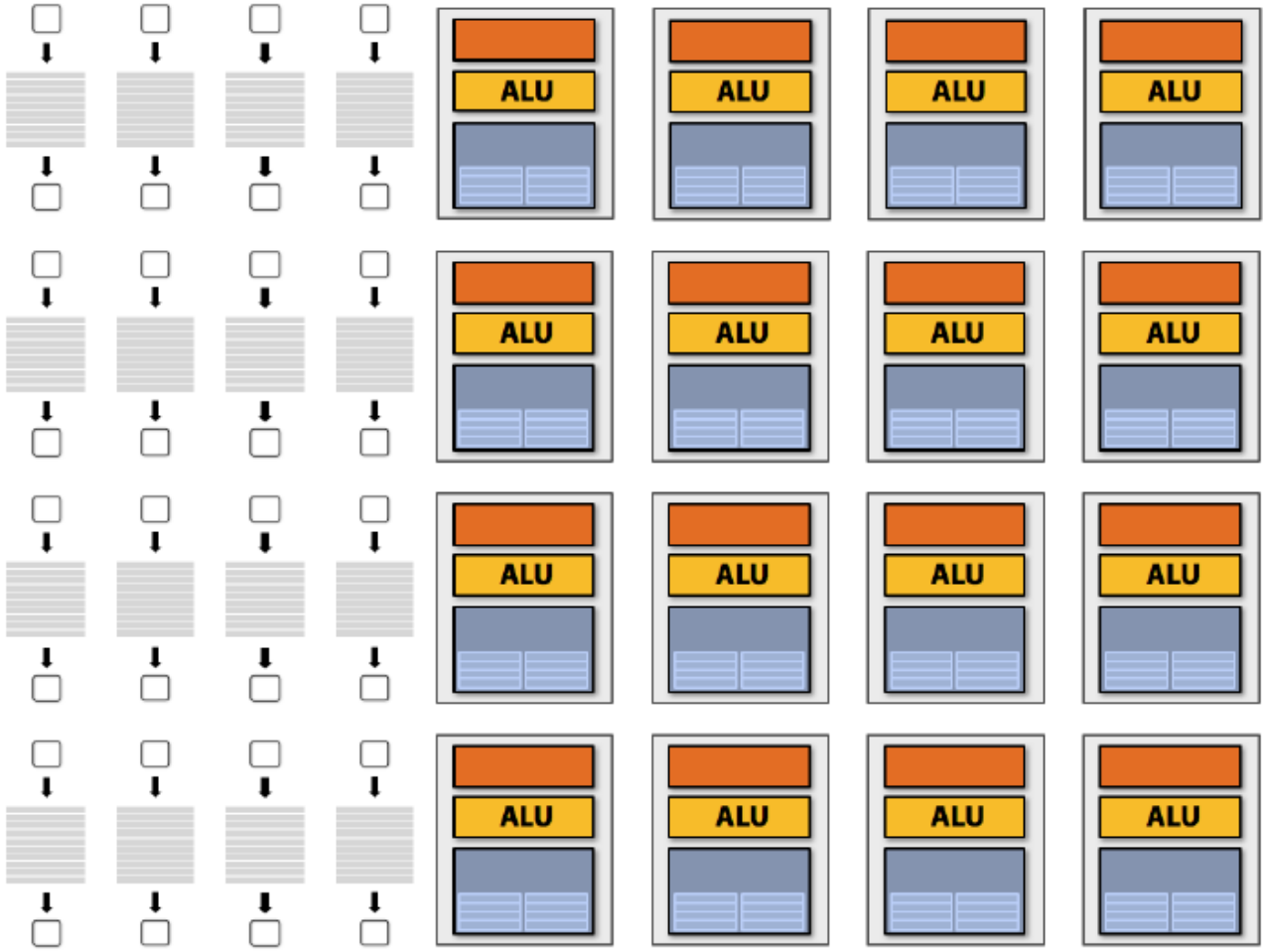
```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```



[Kayvon]

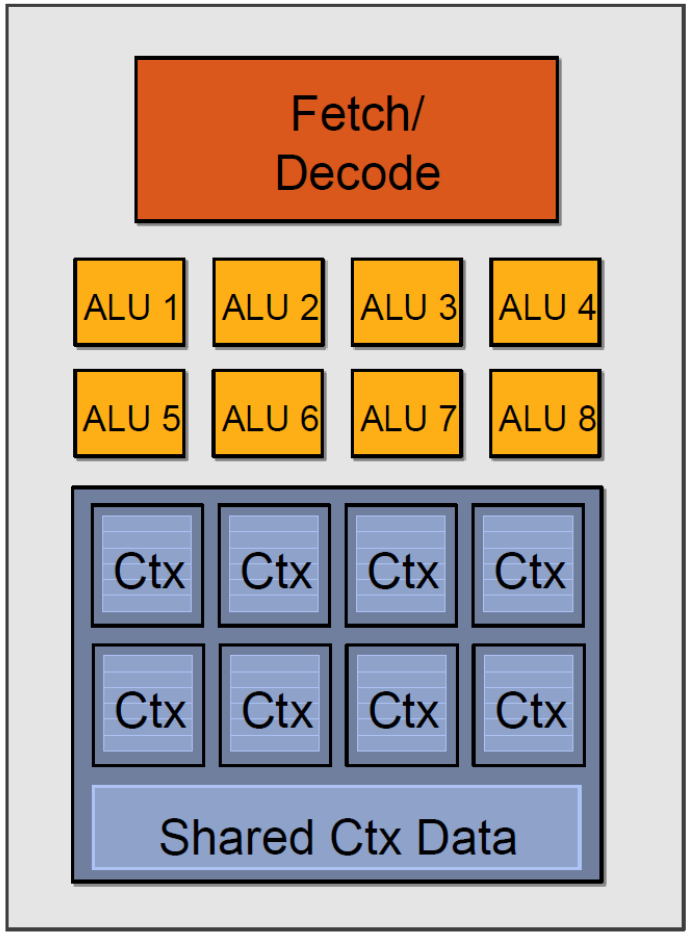
Multi-core Computation





[Kayvon]

SIMD (Single Instruction Multiple Data)



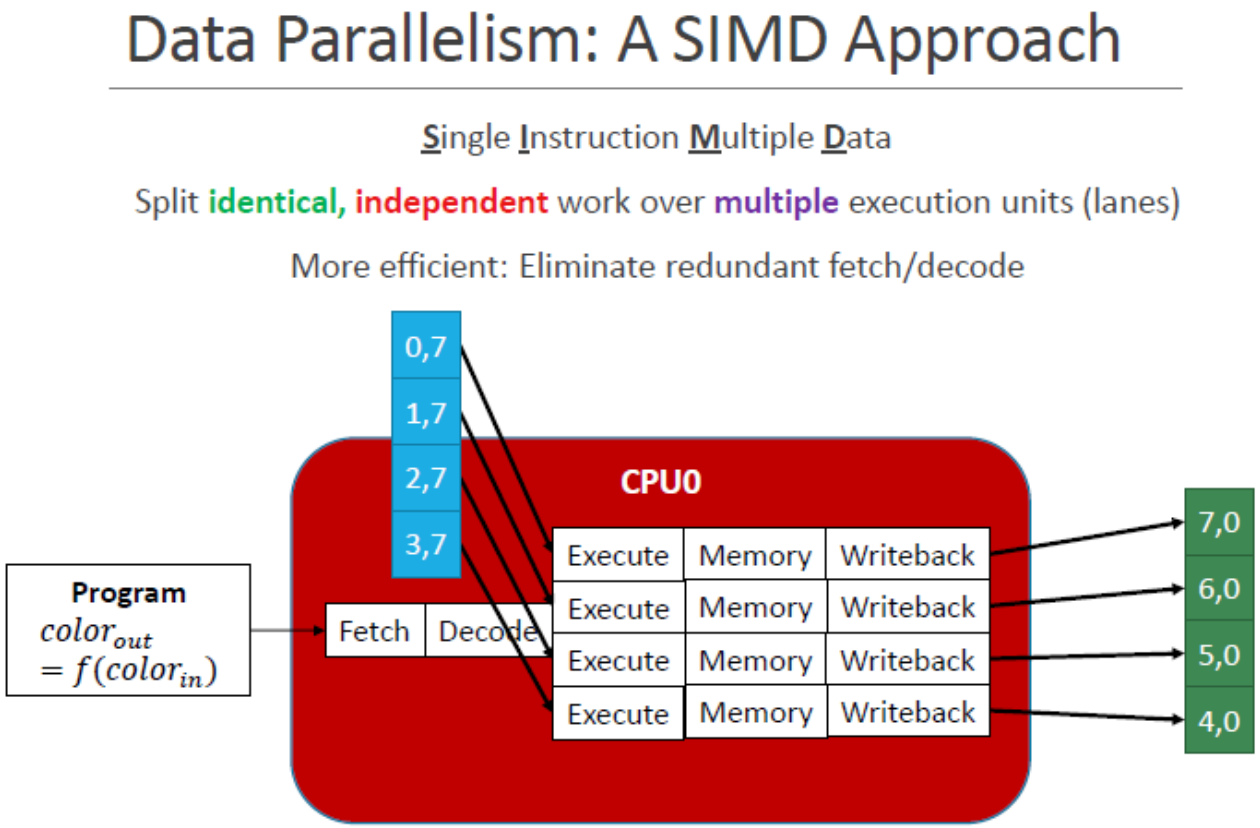
Idea

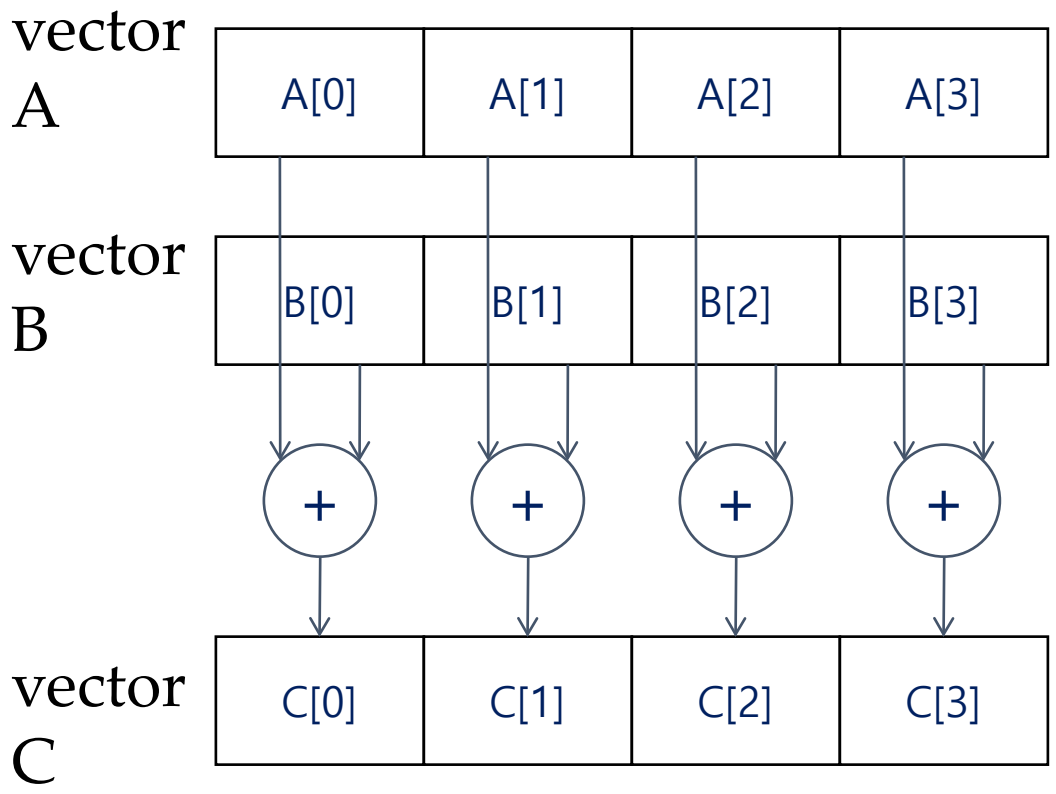
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

[AMD]
[S. Yoo]

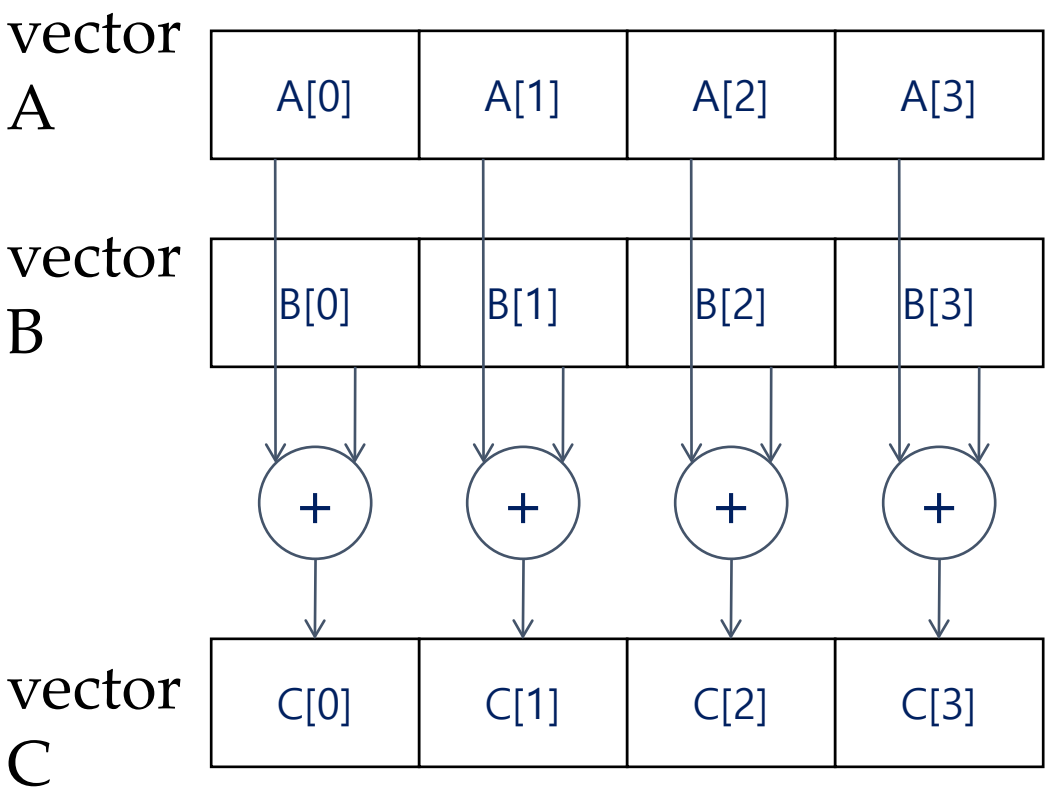
On SIMD Processor …





[AMD]
[S. Yoo]

SIMD or Vector (Addition)

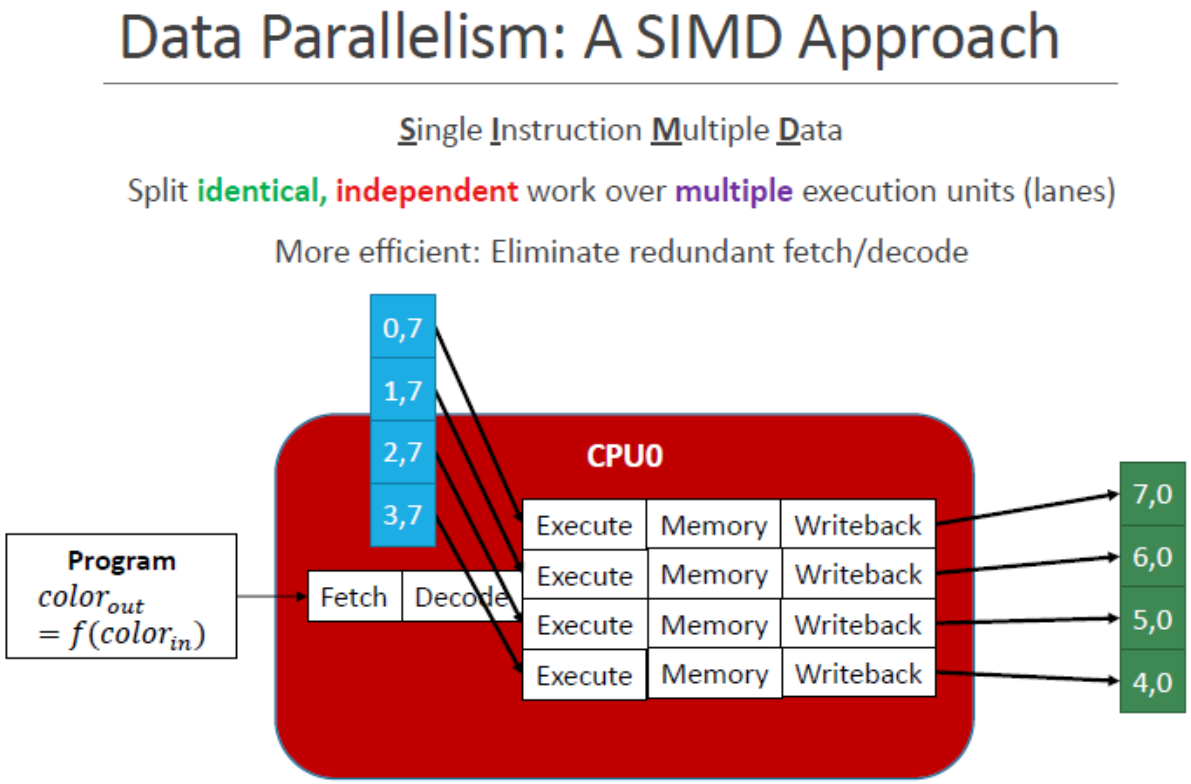


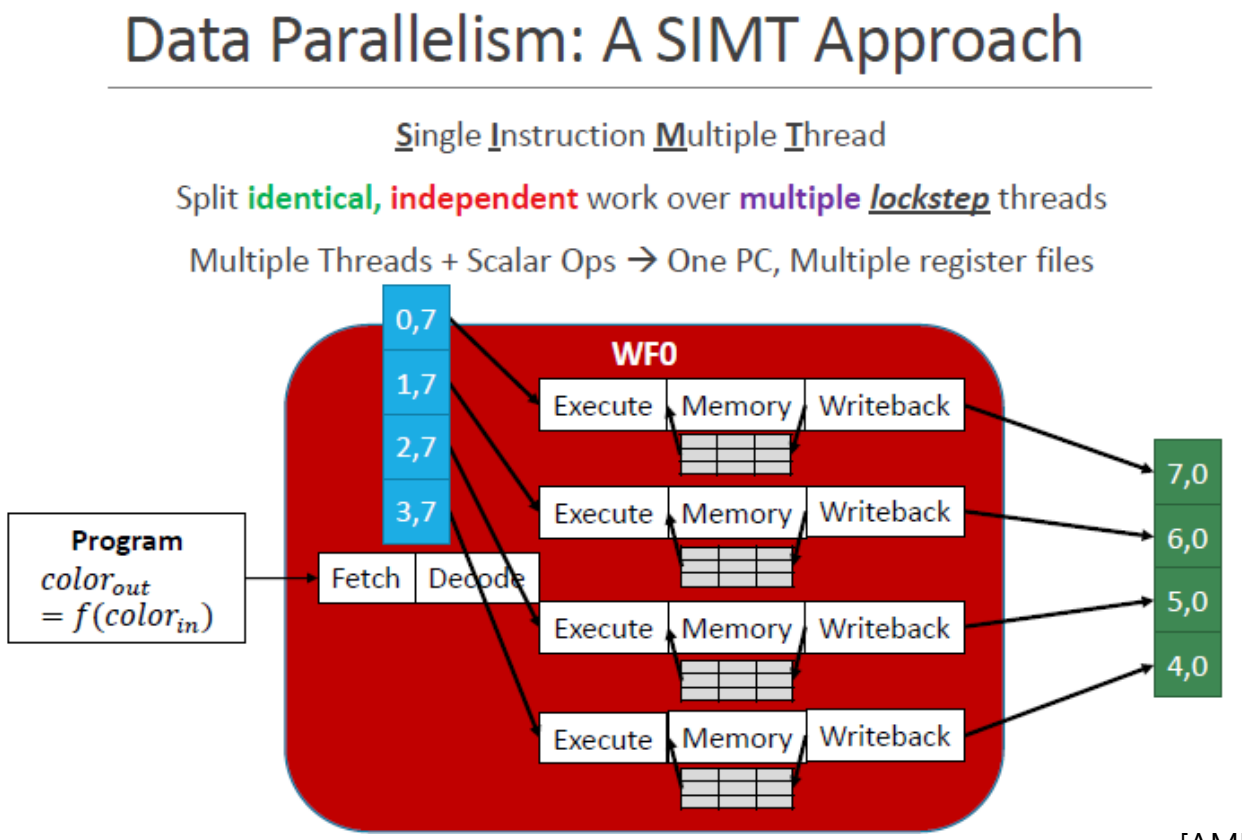

```
int main()
{
  union f4vector a, b, c;

  a.f[0] = 1; a.f[1] = 2; a.f[2] = 3; a.f[3] = 4;
  b.f[0] = 5; b.f[1] = 6; b.f[2] = 7; b.f[3] = 8;

  c.v = a.v + b.v;

  printf("%f, %f, %f, %f\n", c.f[0], c.f[1], c.f[2], c.f[3]);
}
```

```
$ gcc  -ggdb -march=pentium3 -mcpu=pentium3    -c -o example1.o example1.c
$ gcc  -lm example1.o -o example1
$ objdump -dS ./example1.o  | grep -4 c.v | tail -5
  c.v = a.v + b.v;
  8b:   0f 28 45 e8             movaps 0xffffffe8(%ebp),%xmm0
  8f:   0f 58 45 d8             addps  0xffffffd8(%ebp),%xmm0
  93:   0f 29 45 c8             movaps %xmm0,0xffffffc8(%ebp)

  printf("%f, %f, %f, %f\n", c.f[0], c.f[1], c.f[2], c.f[3]);
```

SIMT on GPU

- SIMT is more flexible than SIMD

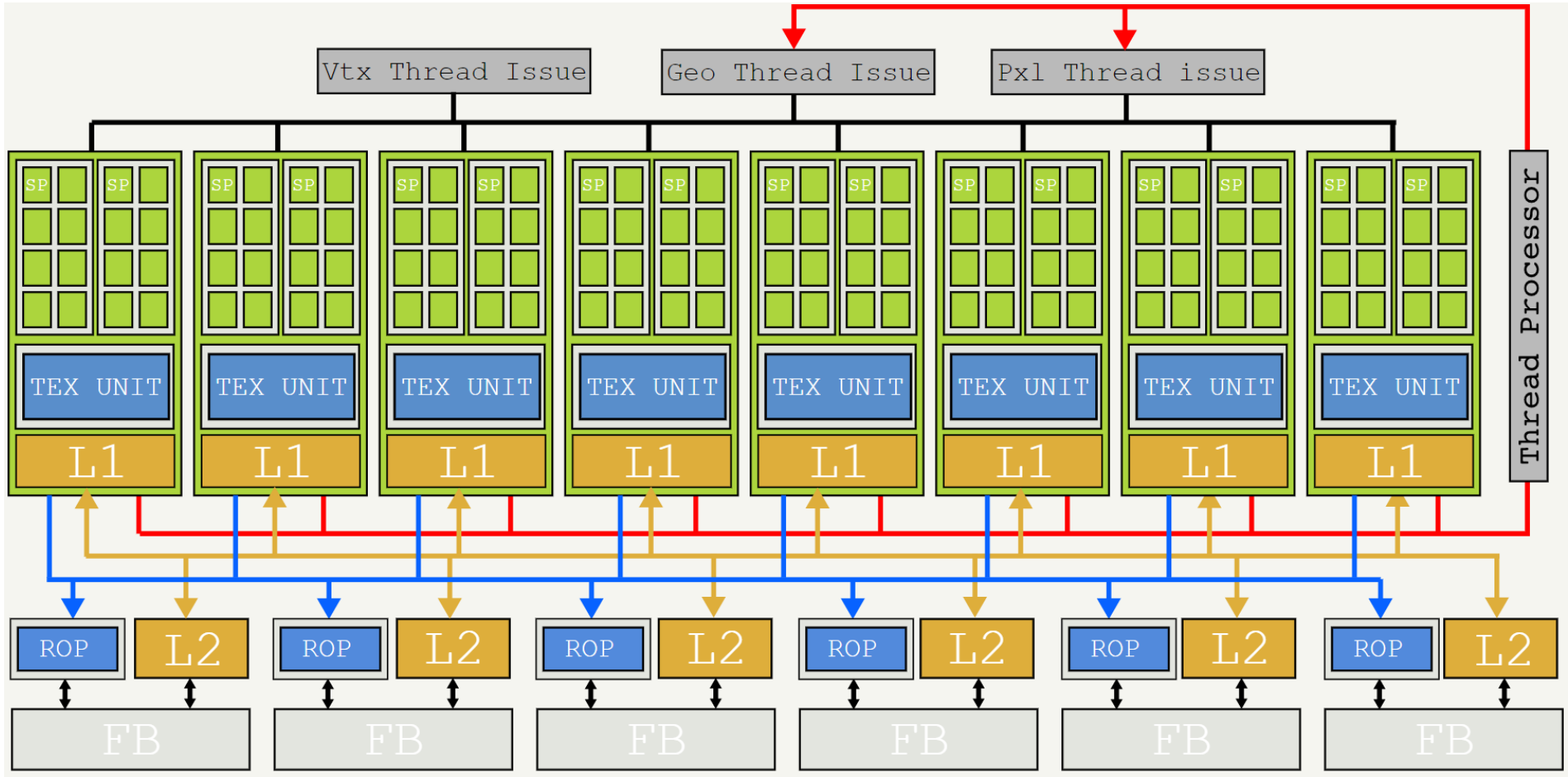


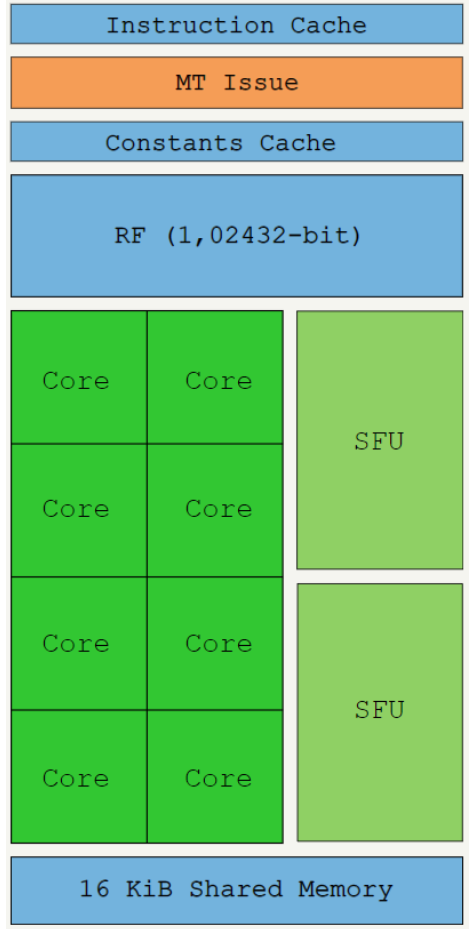


[AMD]
[S. Yoo]

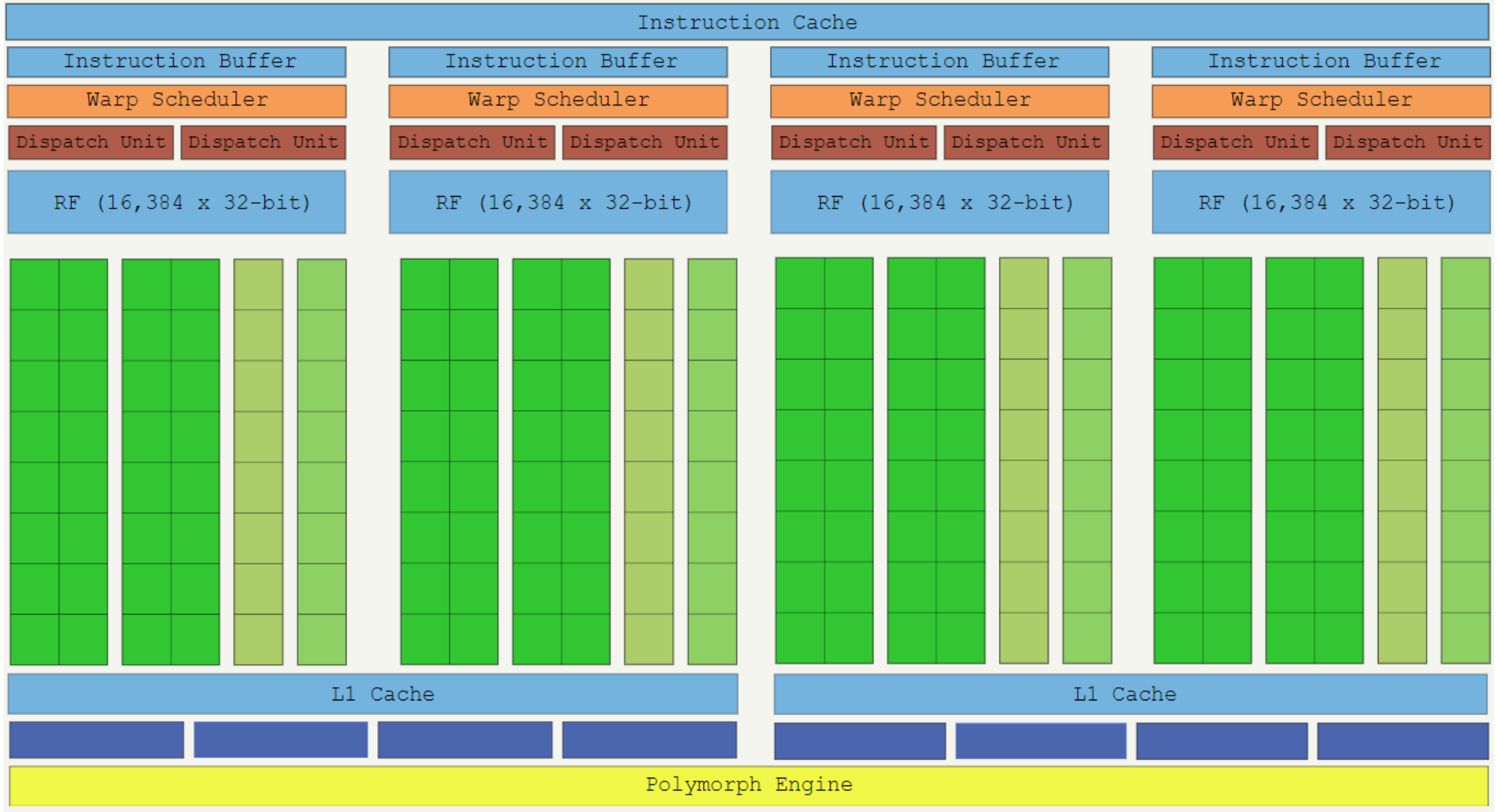
GPU Architecture

NVIDIA GPU – Tesla Architecture (2006)

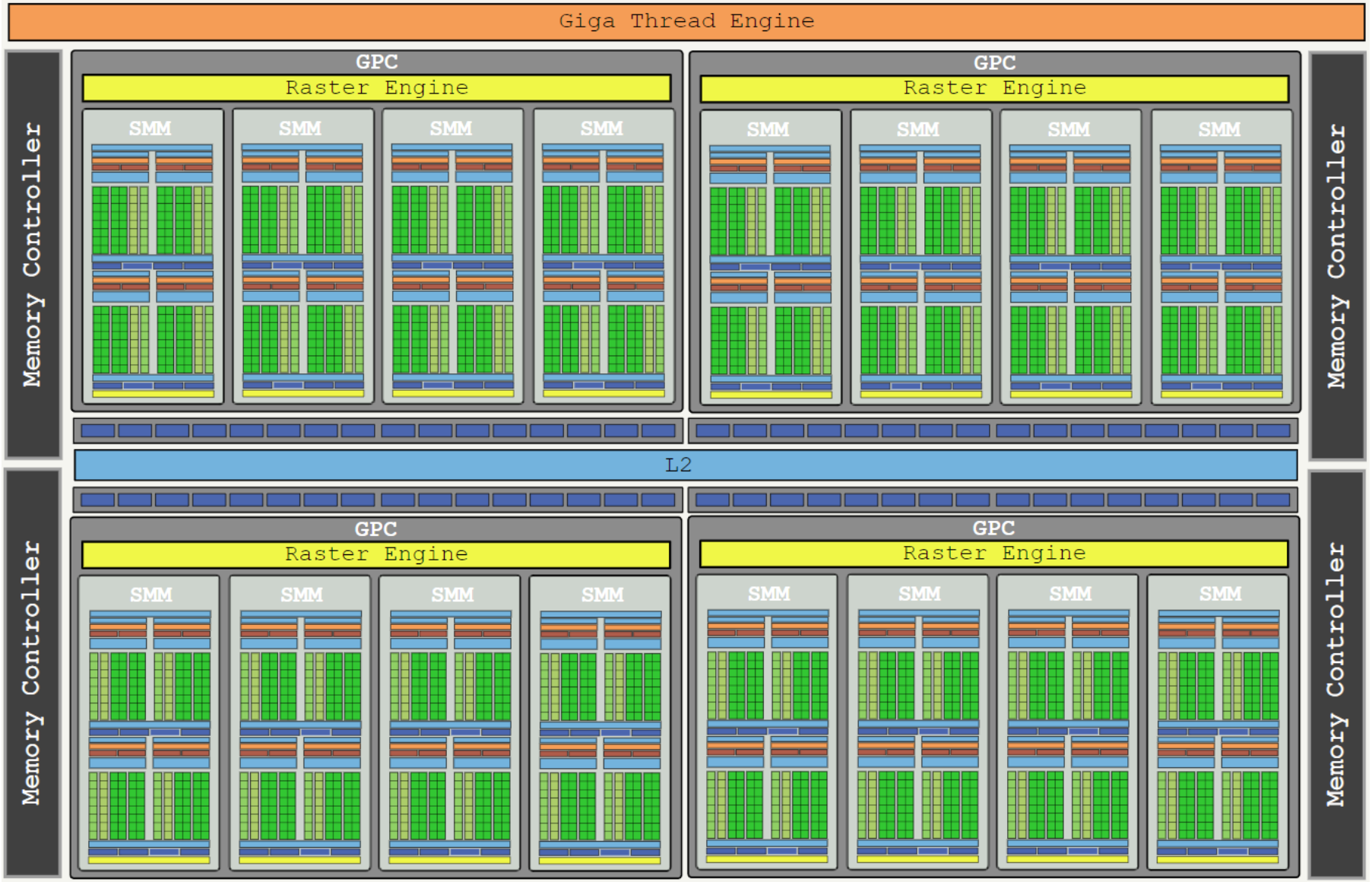




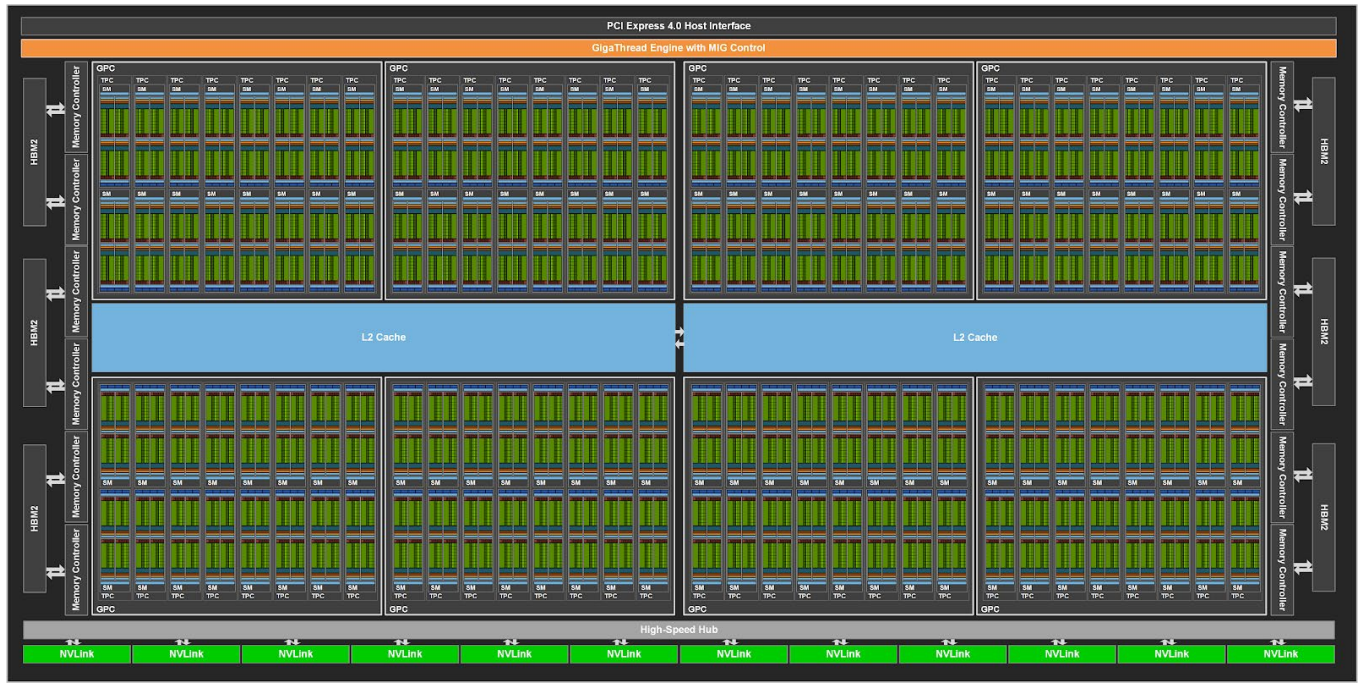
NVDIA GPU – Maxwell (2012)



NVDIA GPU – Maxwell (2012)



NVDIA GPU – Ampree (2020)

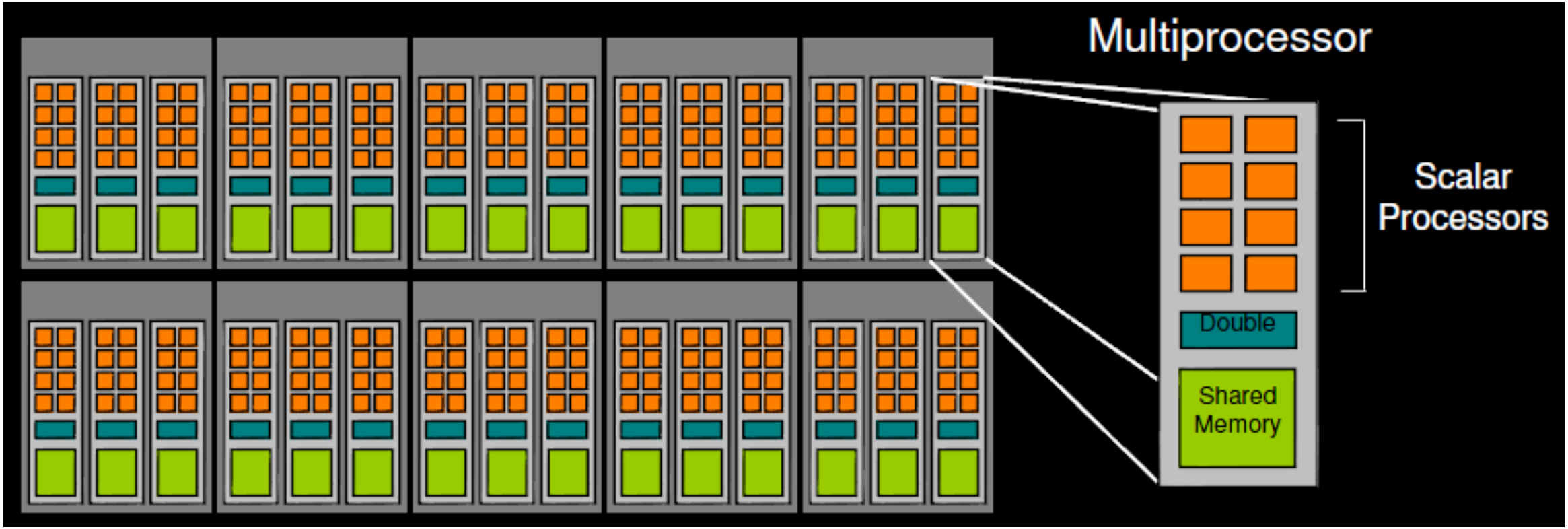


NVDIA GPU Related Materials

- GPU History
  - https://fabiensanglard.net/cuda/
  - https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

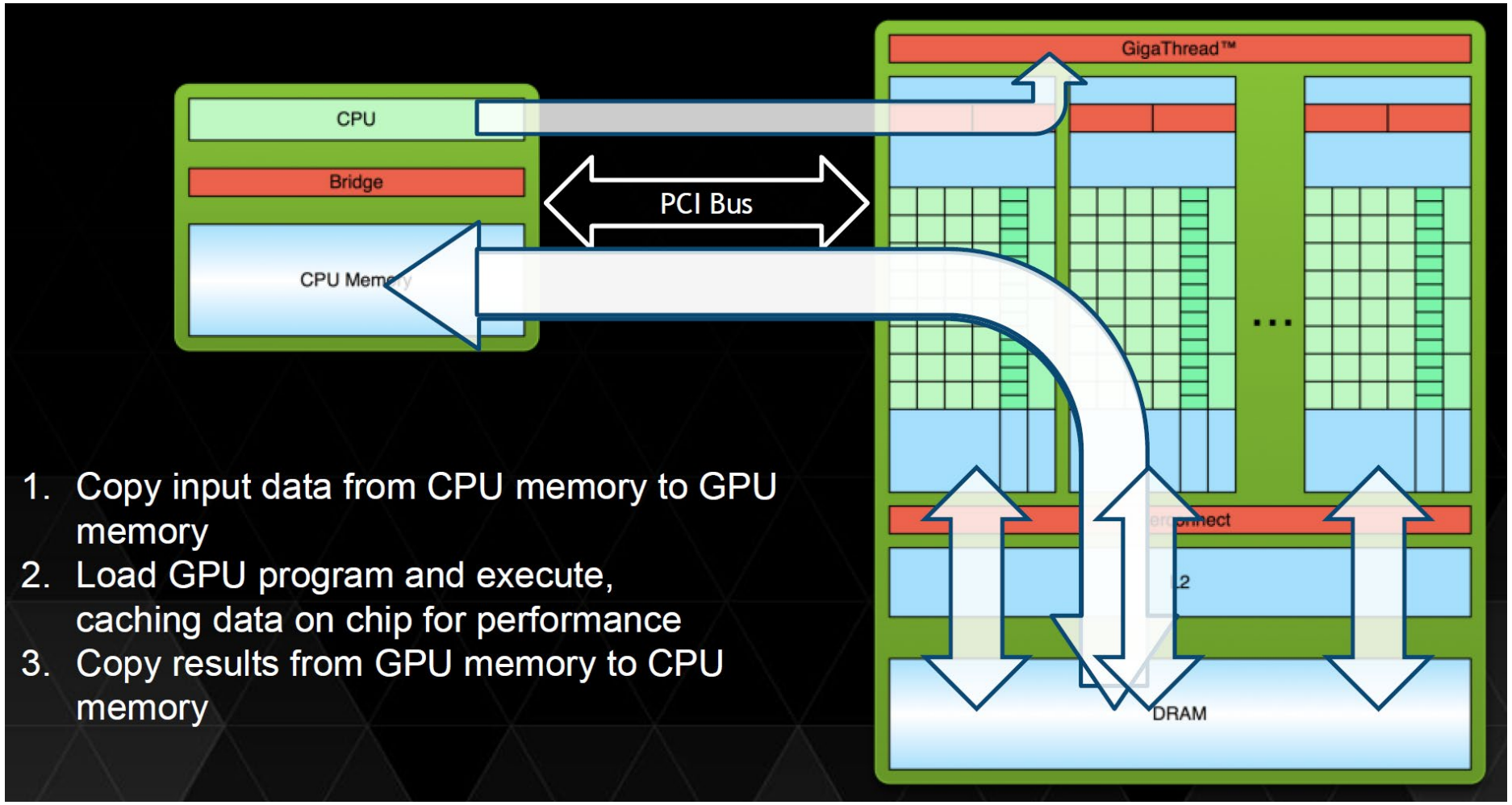
NVIDIA GPU Model

- Streaming multiprocessor (SM) has multiple scalar processor (called CUDA core)



[NVIDIA]

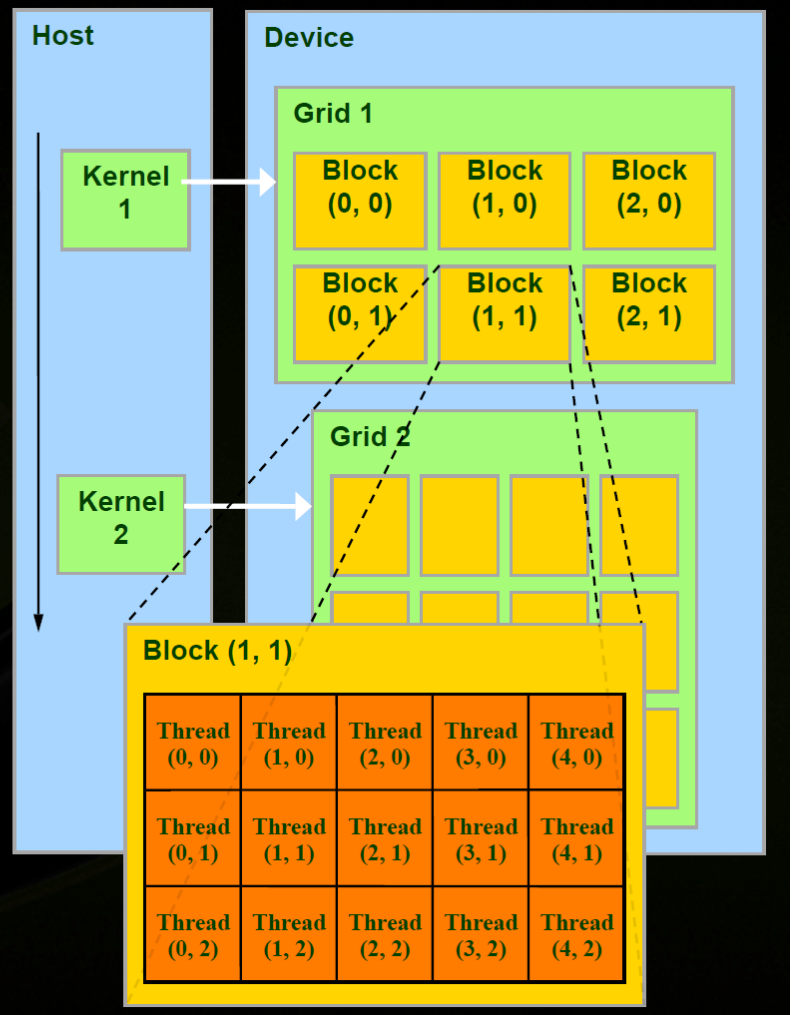
CUDA Programming Model



[NVIDIA]

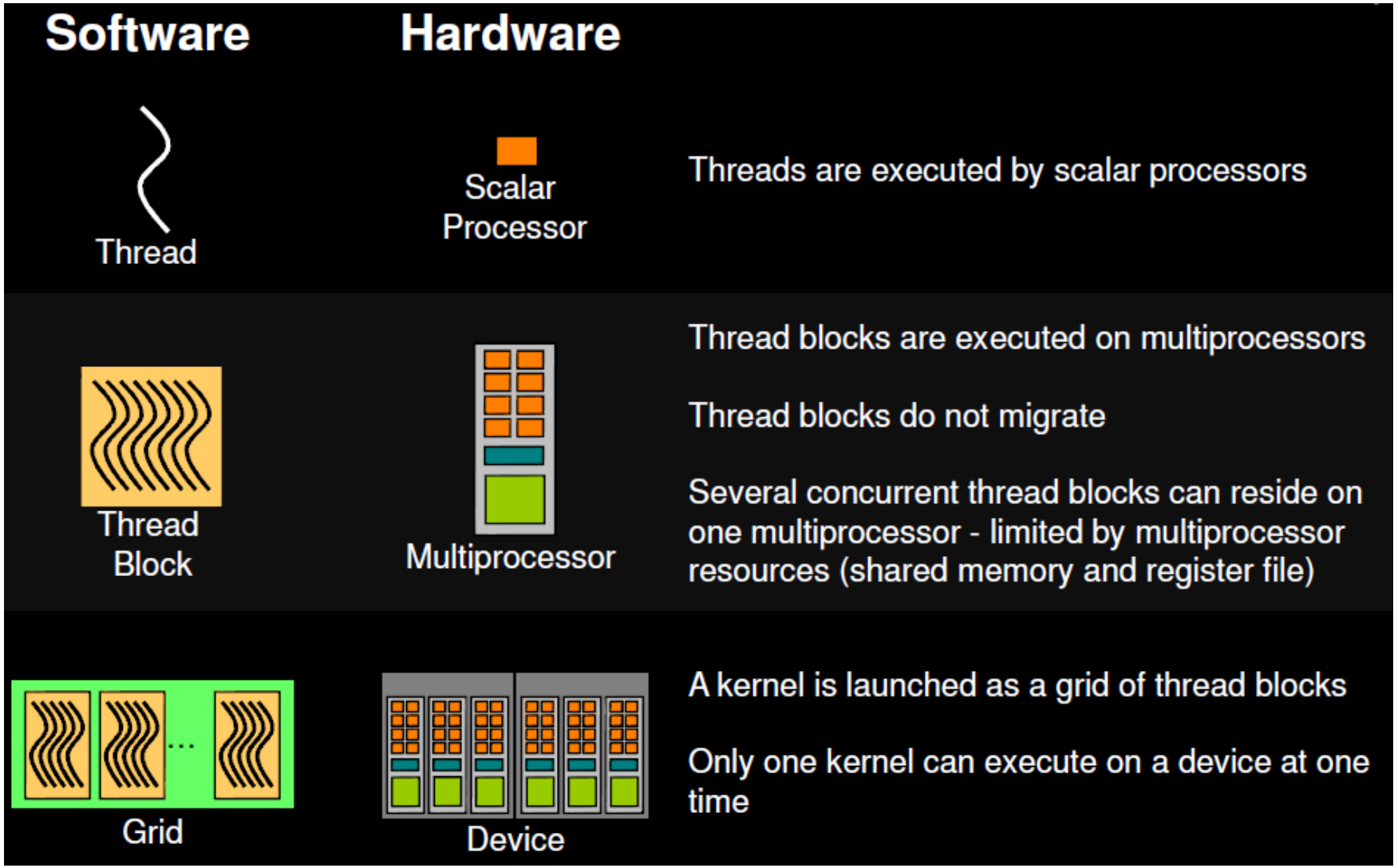
Grid, Block and Thread

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



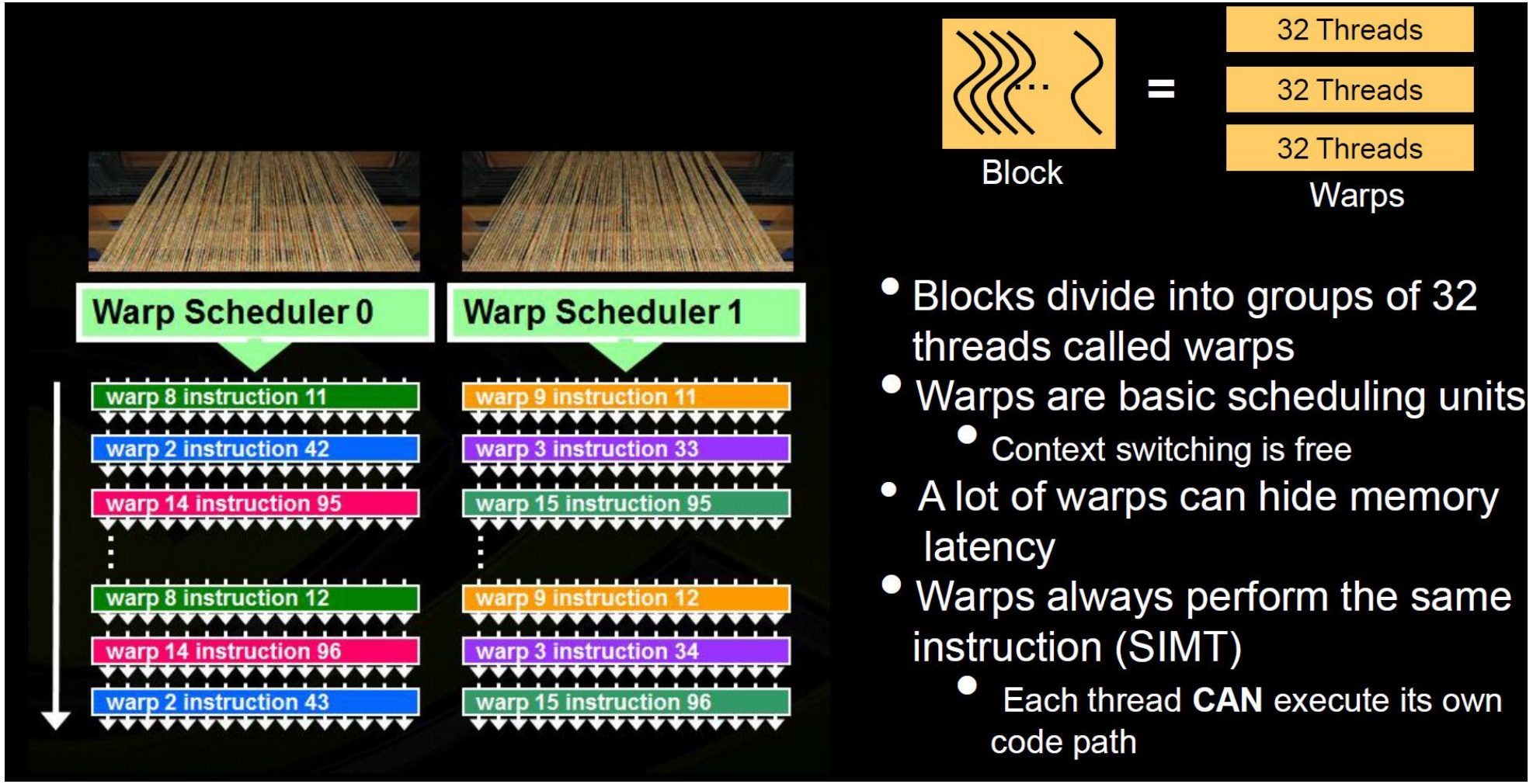
[NVIDIA]

Grid, Block and Thread



[NVIDIA]

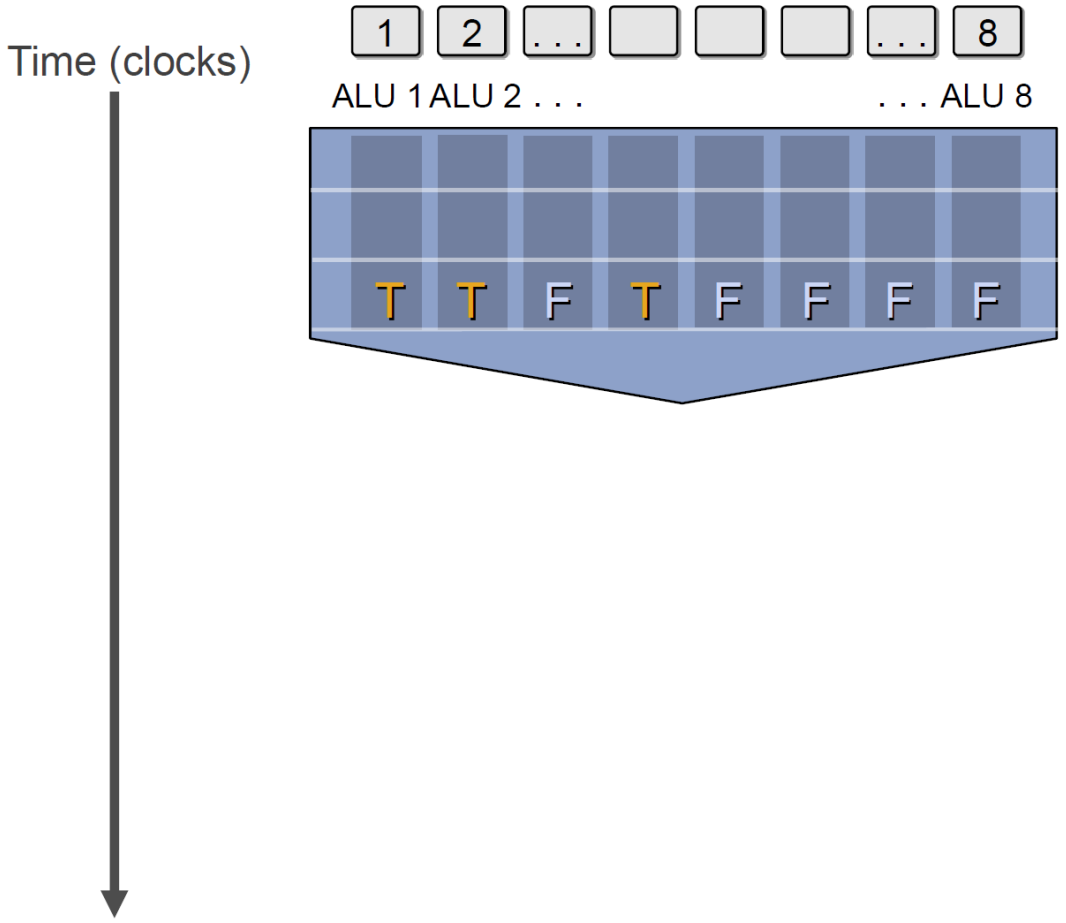
Understanding about Warp



- Blocks divide into groups of 32 threads called warps
- Warps are basic scheduling units
  - Context switching is free
- A lot of warps can hide memory latency
- Warps always perform the same instruction (SIMT)
  - Each thread **CAN** execute its own code path

[NVIDIA]

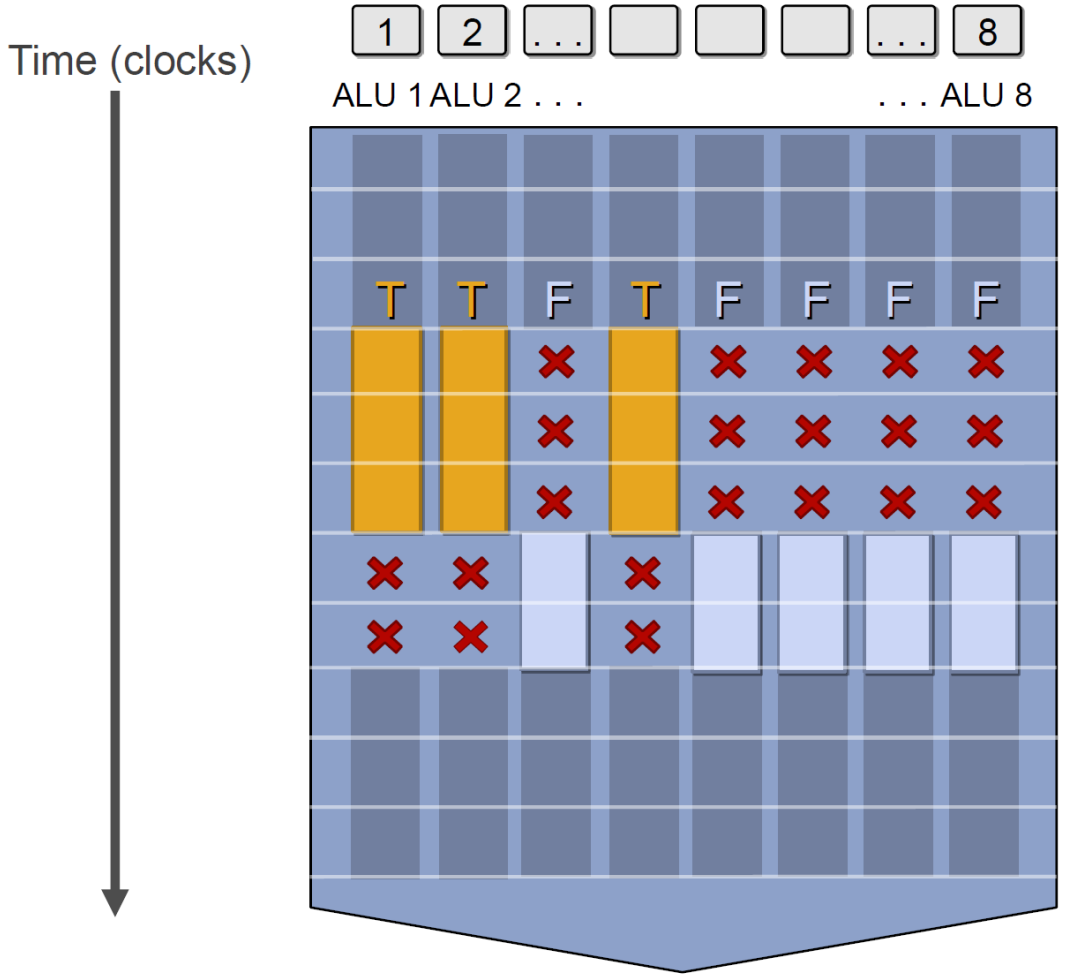
What happens on Branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

[AMD]
[S.Yoo]

Control Divergence Problem




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

- Advanced GPU architecture, Ampree, could have divergent PC counters

[AMD]
[S.Yoo]

CUDA Example: Vector Addition

GPU Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}
```

CUDA Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Host Code

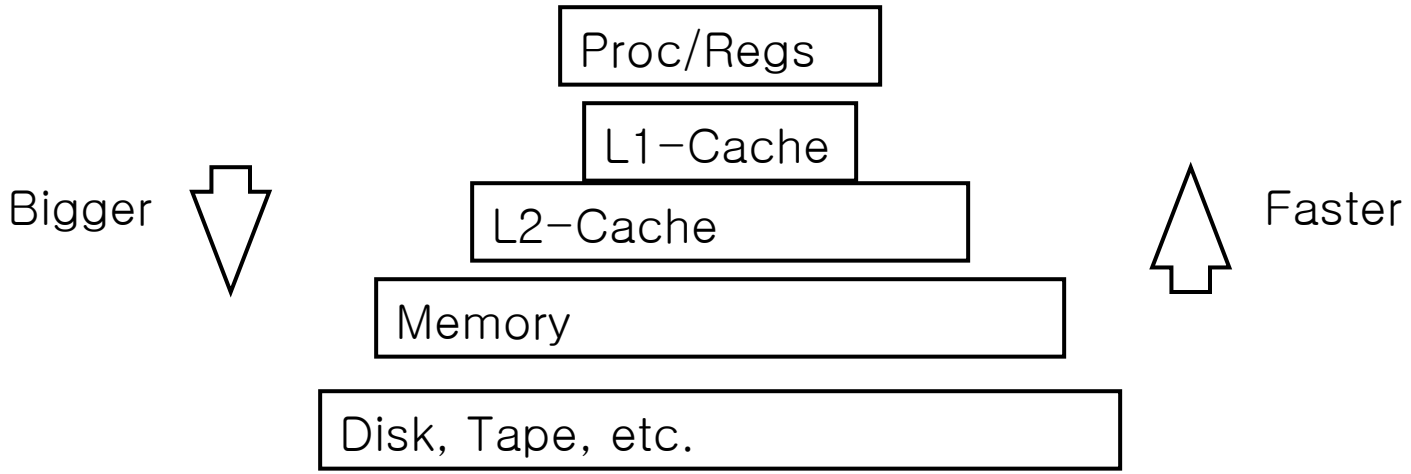
```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  vecAddKernnel<<<ceil(n/256),256>>>(A_d, B_d, C_d, n);
}
```

GPU Memory Hierarchy

- Register
  - Spills to local memory
- Caches
  - Shared memory
  - L1 cache
  - L2 cache
  - Constant memory
  - Texture memory
- Global Memory

What is a cache?

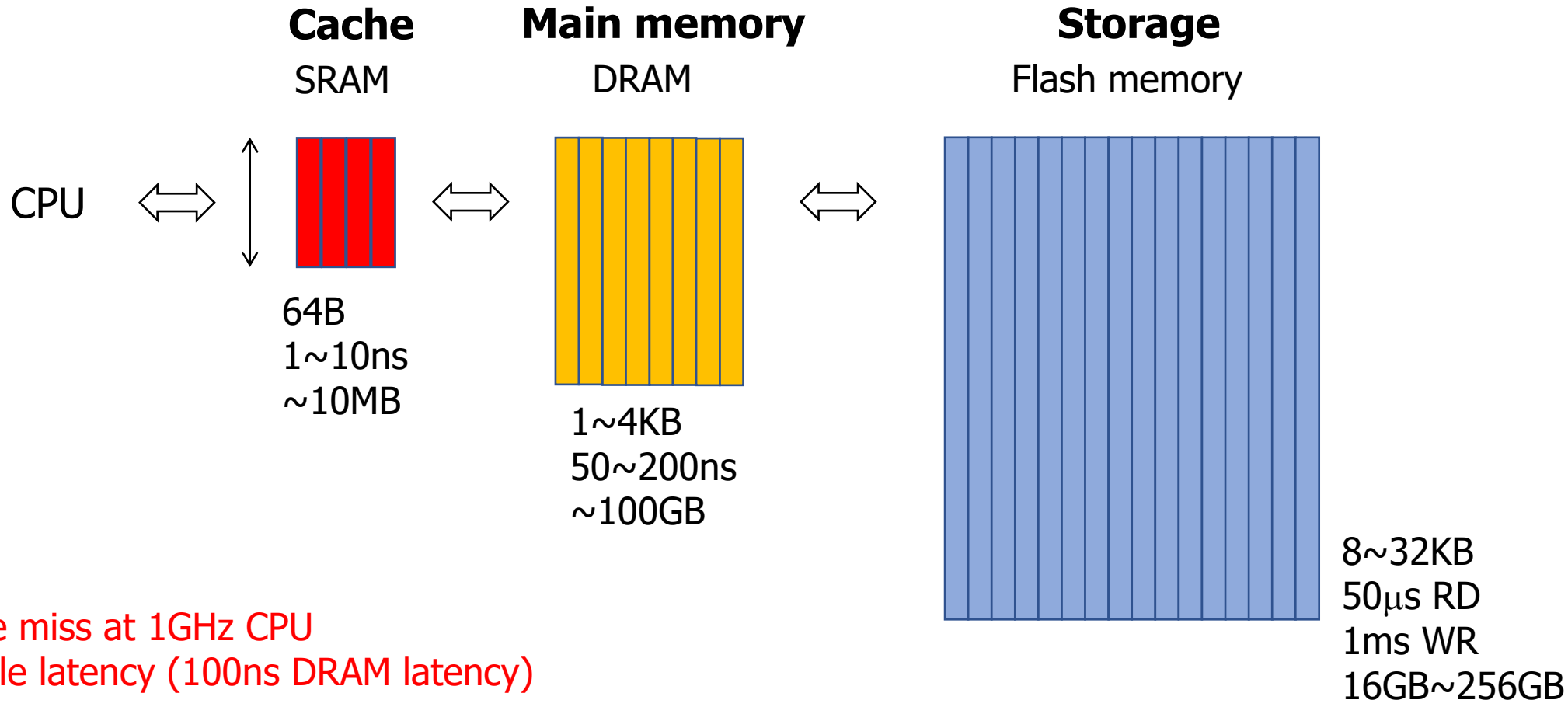
- Small, fast storage used to improve average access time to slow memory.
- Exploits spatial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers a cache on variables
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?



[J. Kubiatowicz, 2000]
[S.Yoo]

Cache, Main Memory and Storage

- Data granularity, latency and size

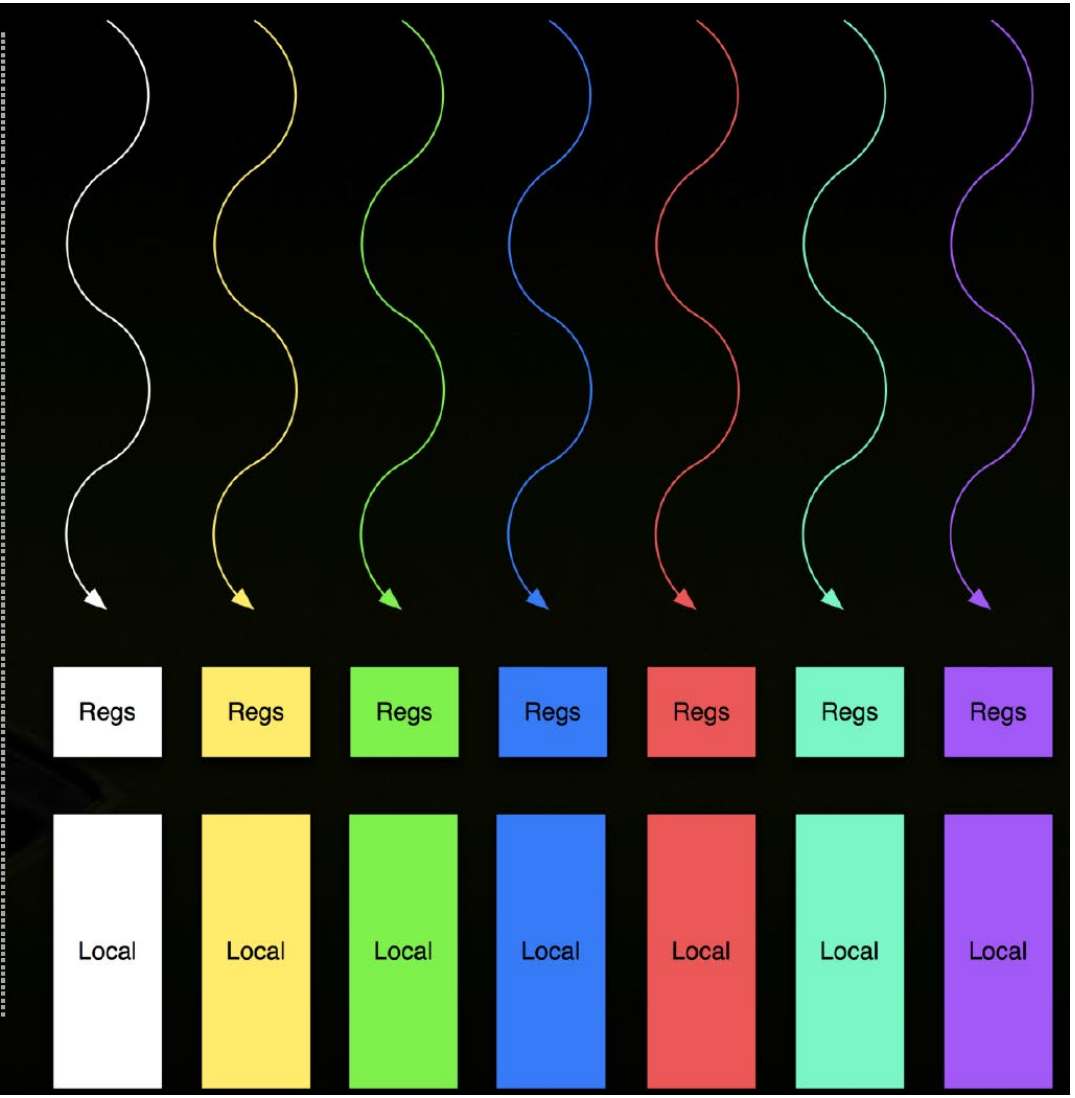


One cache miss at 1GHz CPU
= 100 cycle latency (100ns DRAM latency)

[S.Yoo]

GPU Memory Hierarchy Detail

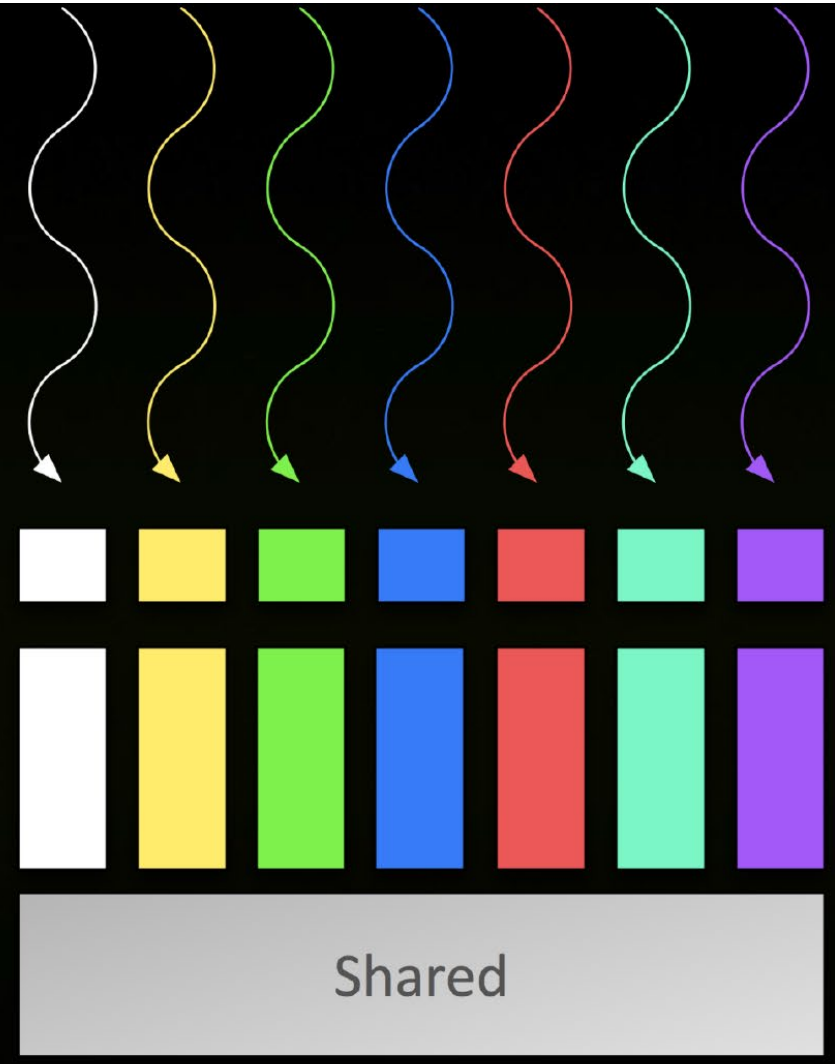
- Thread:
  - Registers
- Thread:
  - Local memory



[NVIDIA]

GPU Memory Hierarchy Detail - 2

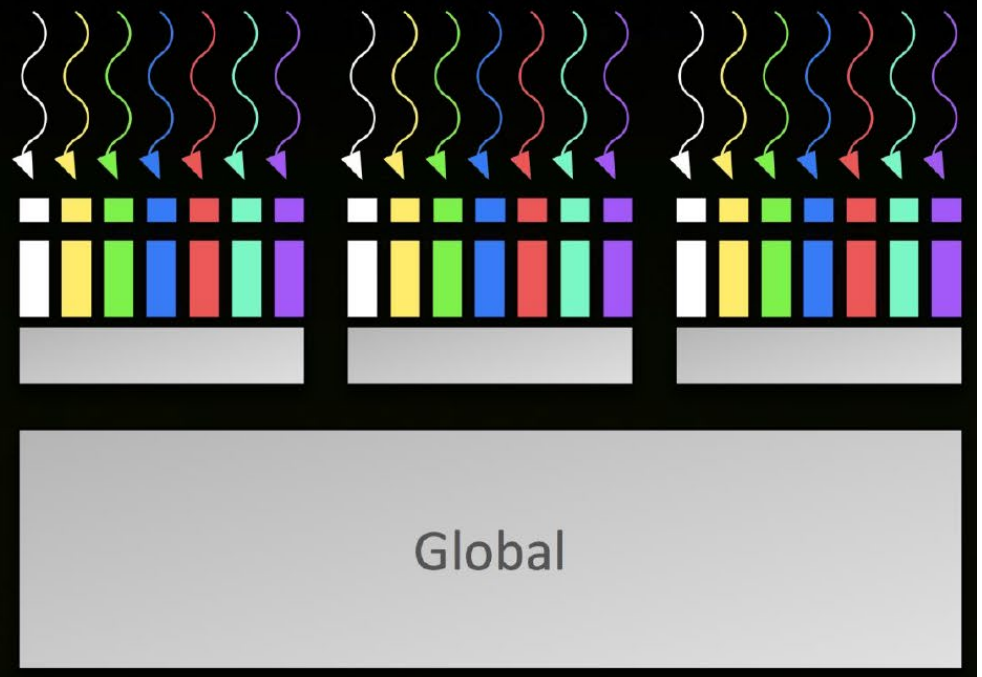
- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory



[NVIDIA]

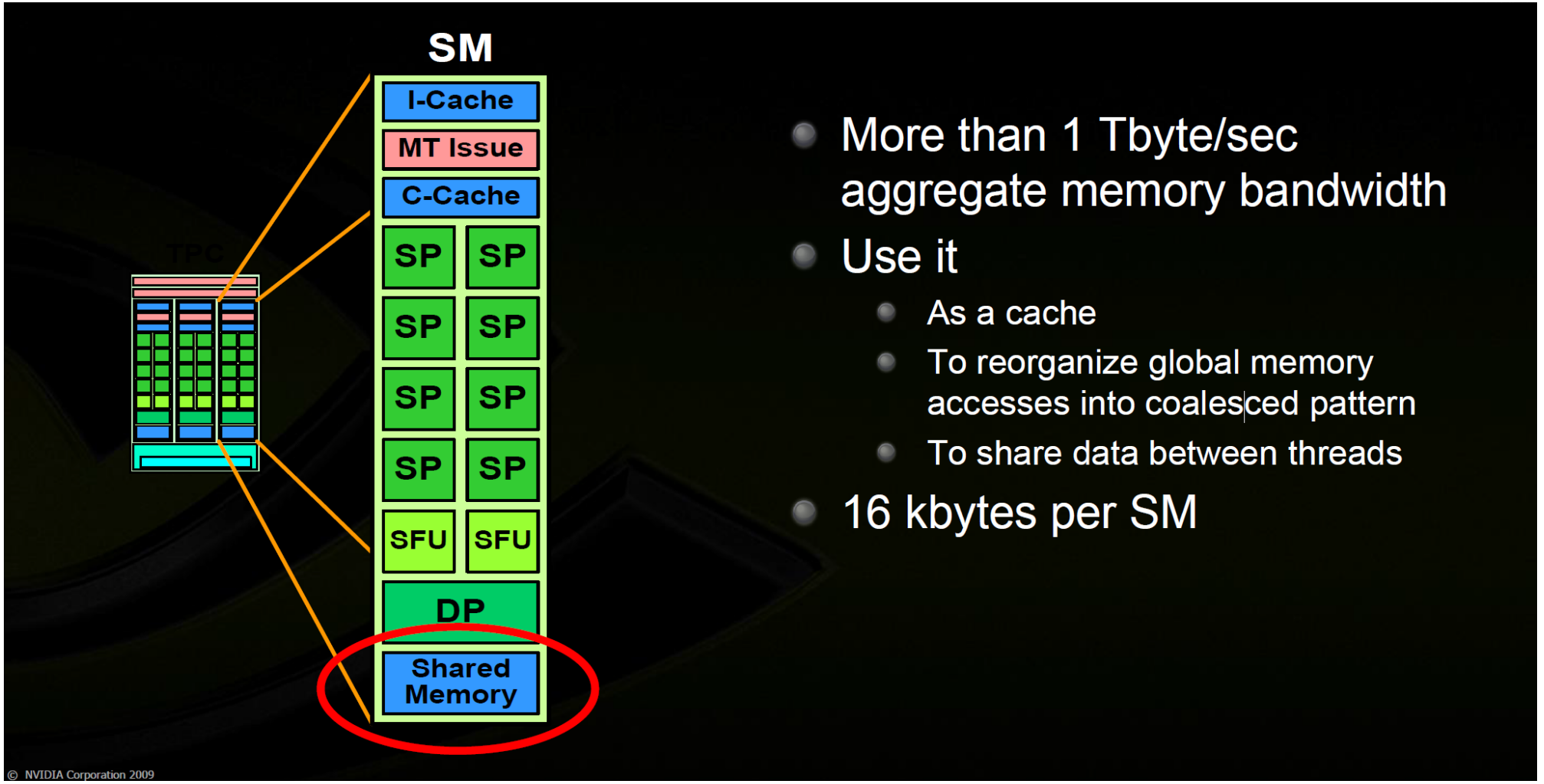
GPU Memory Hierarchy Detail - 3

- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



[NVIDIA]

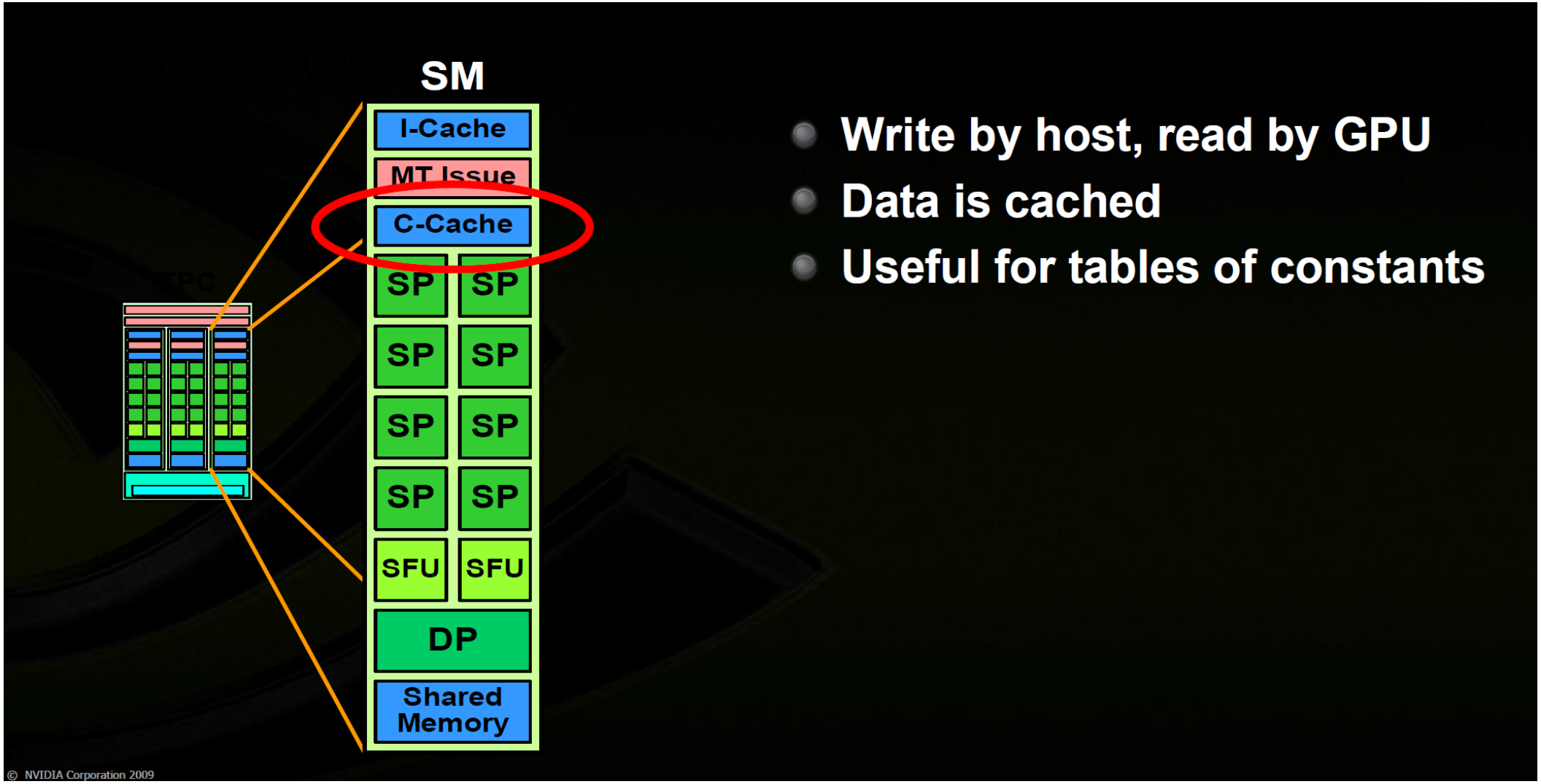
Shared Memory



- More than 1 Tbyte/sec aggregate memory bandwidth
- Use it
  - As a cache
  - To reorganize global memory accesses into coalesced pattern
  - To share data between threads
- 16 kbytes per SM

[NVIDIA]

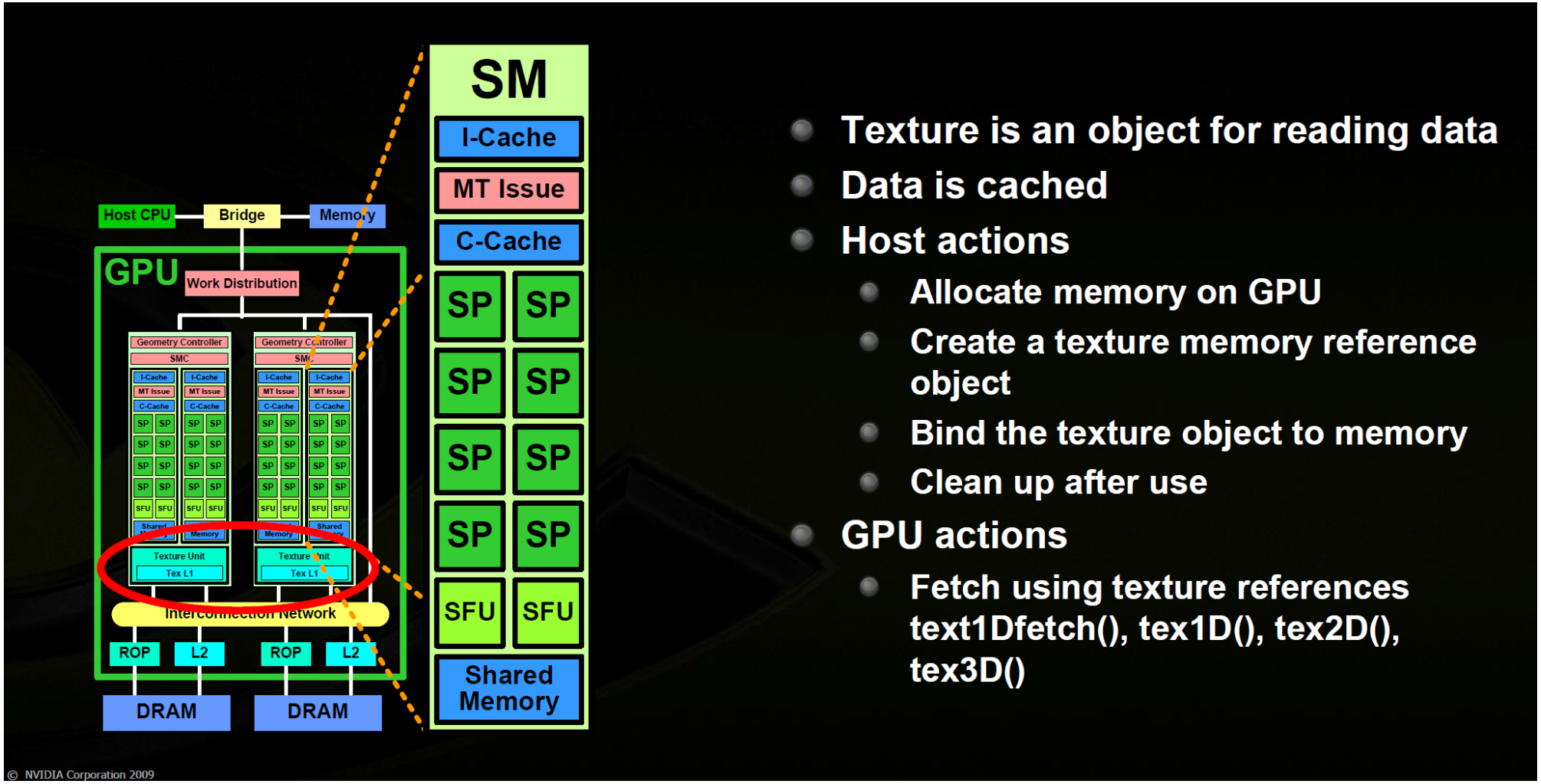
Constant Memory



- Write by host, read by GPU
- Data is cached
- Useful for tables of constants

[NVIDIA]

Texture Memory

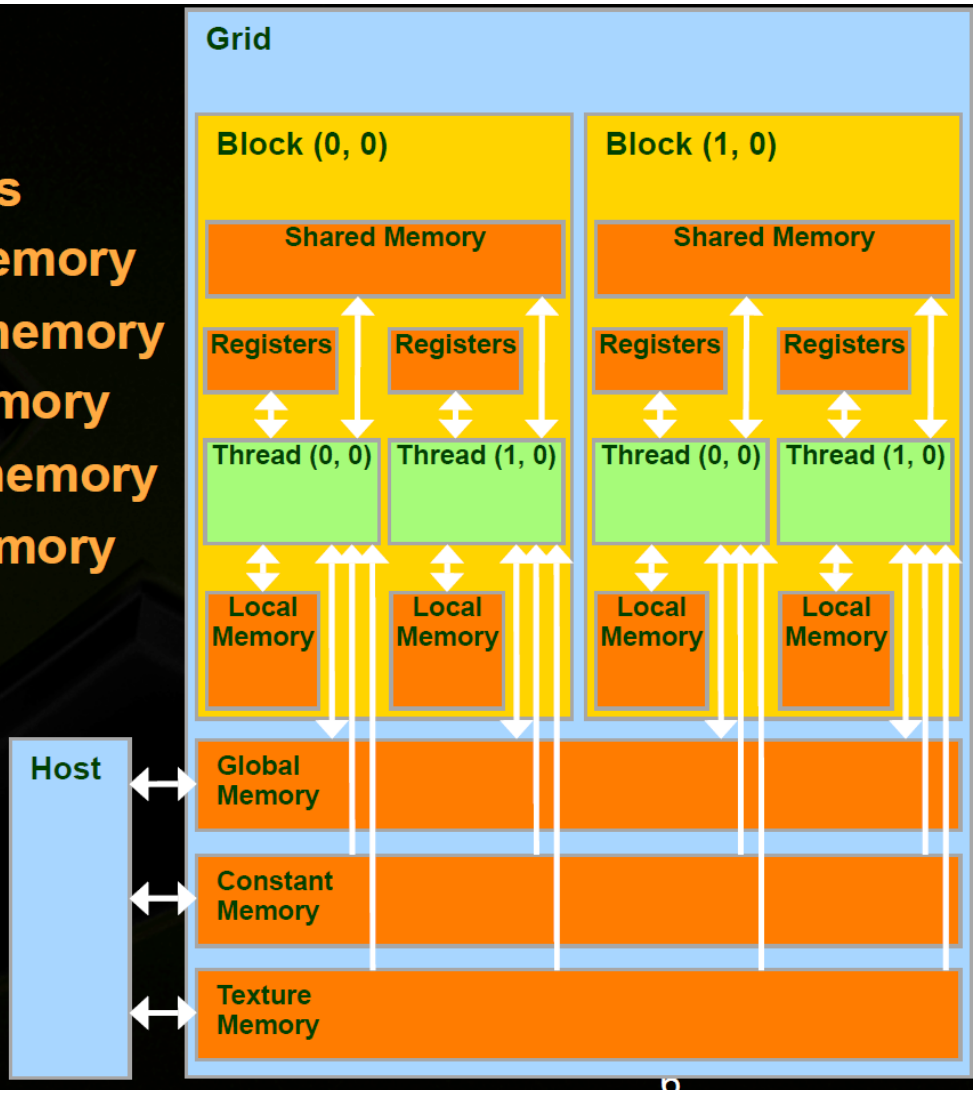


- Texture is an object for reading data
- Data is cached
- Host actions
  - Allocate memory on GPU
  - Create a texture memory reference object
  - Bind the texture object to memory
  - Clean up after use
- GPU actions
  - Fetch using texture references text1Dfetch(), tex1D(), tex2D(), tex3D()

[NVIDIA]

Memory Space I/O

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can read/write global, constant, and texture memory (stored in DRAM)



[NVIDIA]

Overview

Performance Optimization

Terminology Review

- **Thread**: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
  - Note difference from CPU threads: creation cost, resource
  - usage, and switching cost of GPU threads is much smaller
- **Warp**: a group of threads executed physically in parallel (SIMD)
  - Half-warp: the first or second half of a warp of threads
- **Thread Block**: a group of threads that are executed together and can share memory on a single multiprocessor
- **Grid**: a group of thread blocks that execute a single CUDA kernel logically in parallel on a single GPU

Optimization Agenda

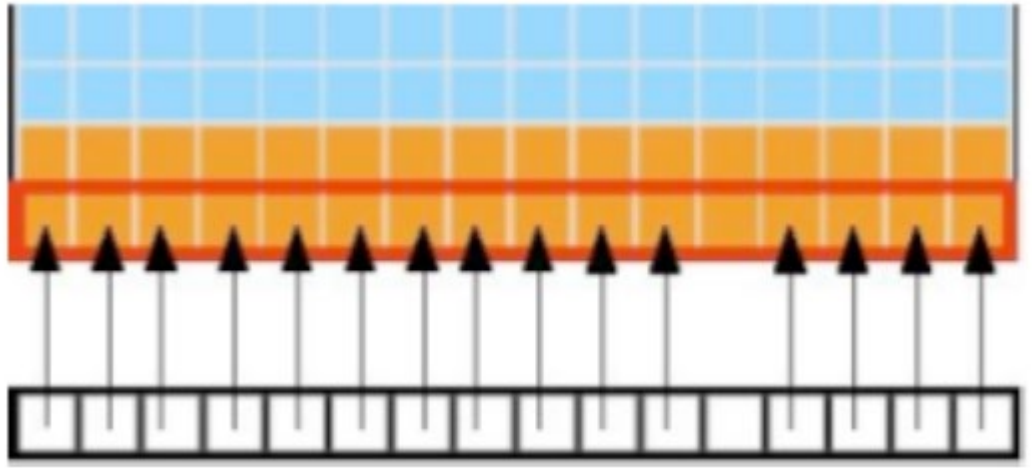
- Memory coalescing
- Data layout & reuse optimization
- Resource utilization maximization
  - Latency hiding based on thread switching
  - CUDA Resource constraints
- Host-Device interaction minimization

Global Memory Access on GPU

- Highest latency: ~100s of clock cycles
  - Could be performance bottleneck
  - Increase bandwidth utilization with memory coalescing / multi-bank cache

Cache & Main Memory Access on GPU

- If all the threads of the same warp access the same cache block
- In case of cache hit
  - If there is no bank conflict, one cycle L1 cache access
  - In case of bank conflicts, multiple cycles in L1 cache accesses
- In case of cache miss
  - Single access to L2 cache or main memory access

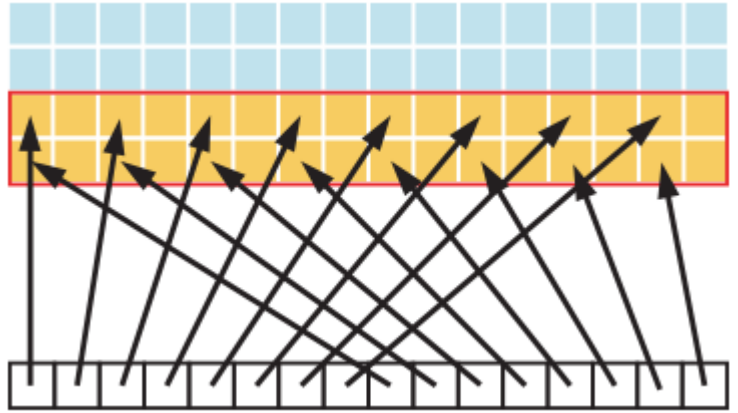


- However, sometimes non-caching could be beneficial
  - Non-caching reduce cache-line size

[S.Yoo]

Memory Request Coalescing by GPU

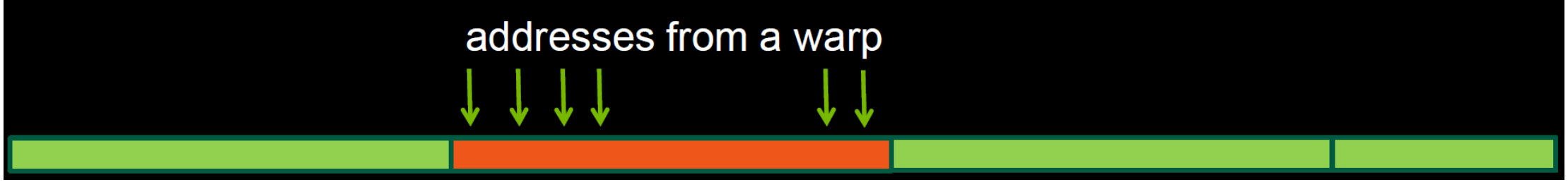
- If the threads of the same warp access N different cache blocks
- In case of cache hit
  - At least N cycles + bank conflicts determine total L1 cache access latency
- In case of cache miss
  - N accesses to L2 cache or main memory access instead of 32 cache miss requests
    - Memory request coalescing



[S.Yoo]

Memory Coalescing

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



[nvidia]

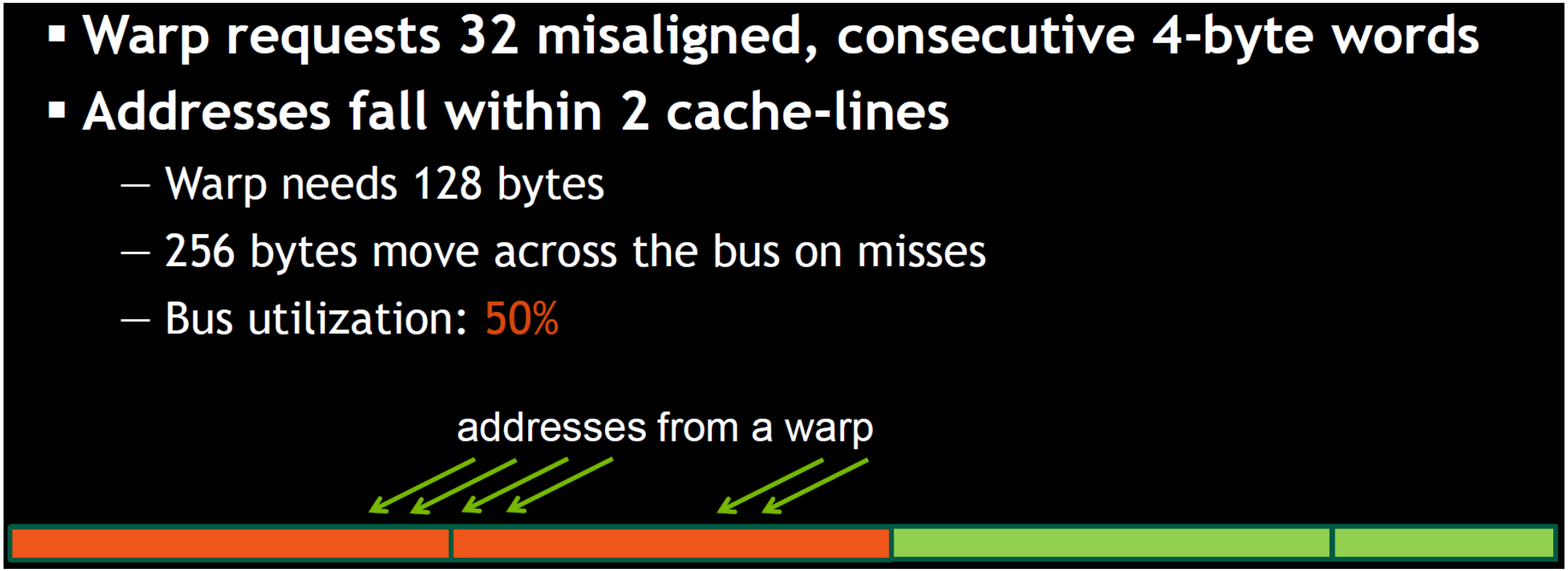
Memory Coalescing - 2

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

[nvidia]

Memory Coalescing - 3

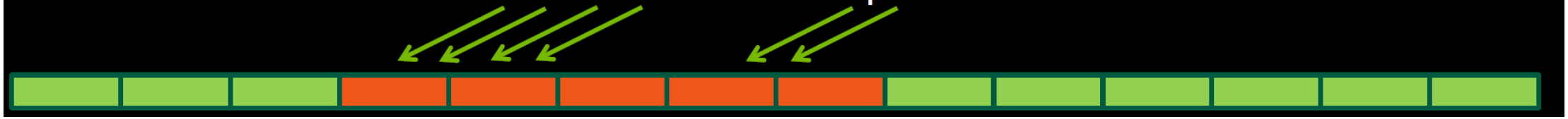
- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%



[nvidia]

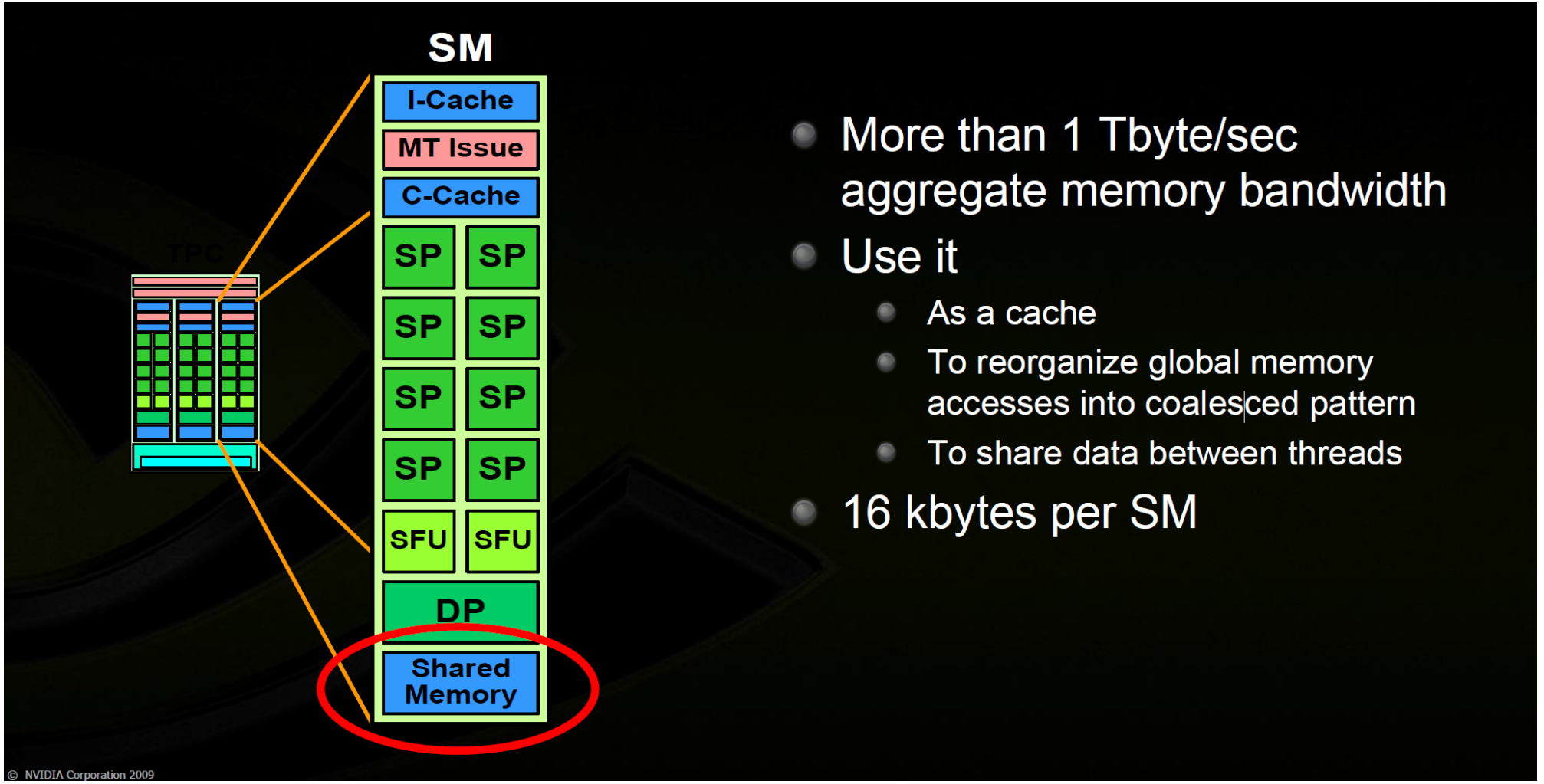
Memory Coalescing - 4

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



[nvidia]

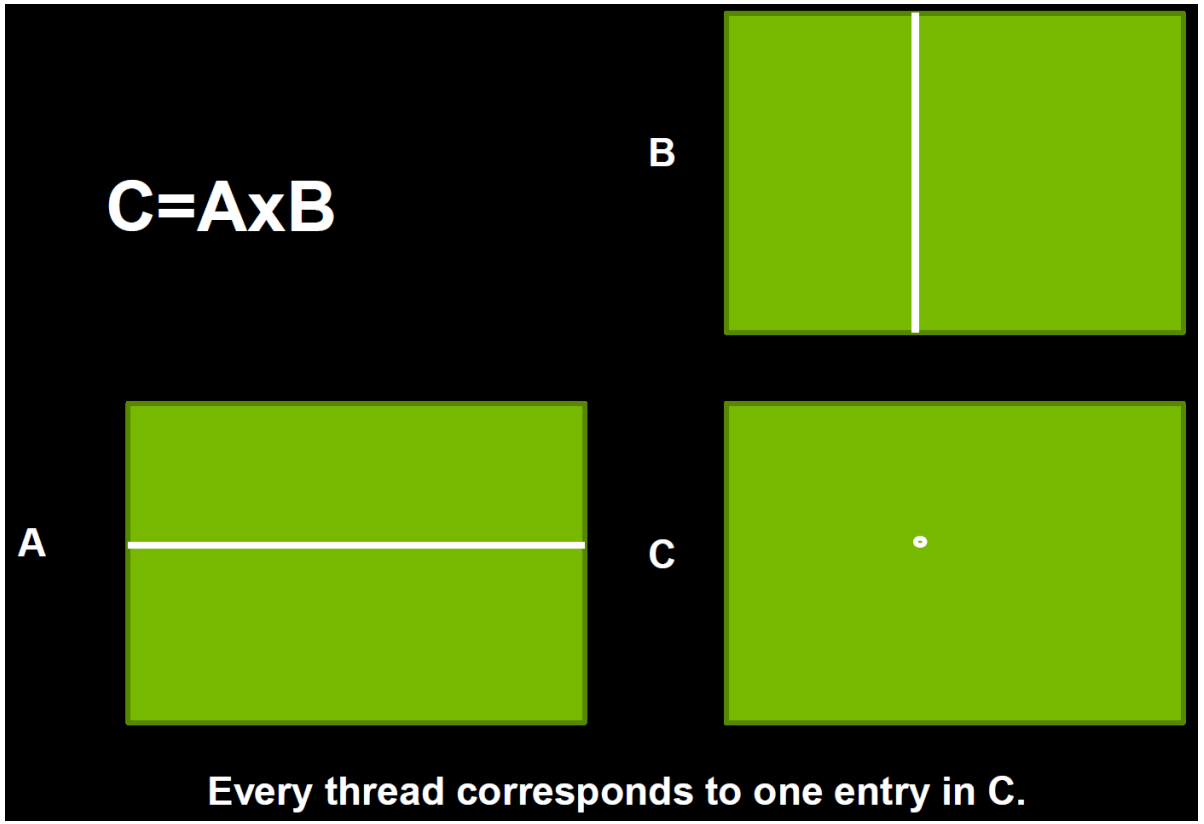
Shared Memory



- More than 1 Tbyte/sec aggregate memory bandwidth
- Use it
  - As a cache
  - To reorganize global memory accesses into coalesced pattern
  - To share data between threads
- 16 kbytes per SM

[NVIDIA]

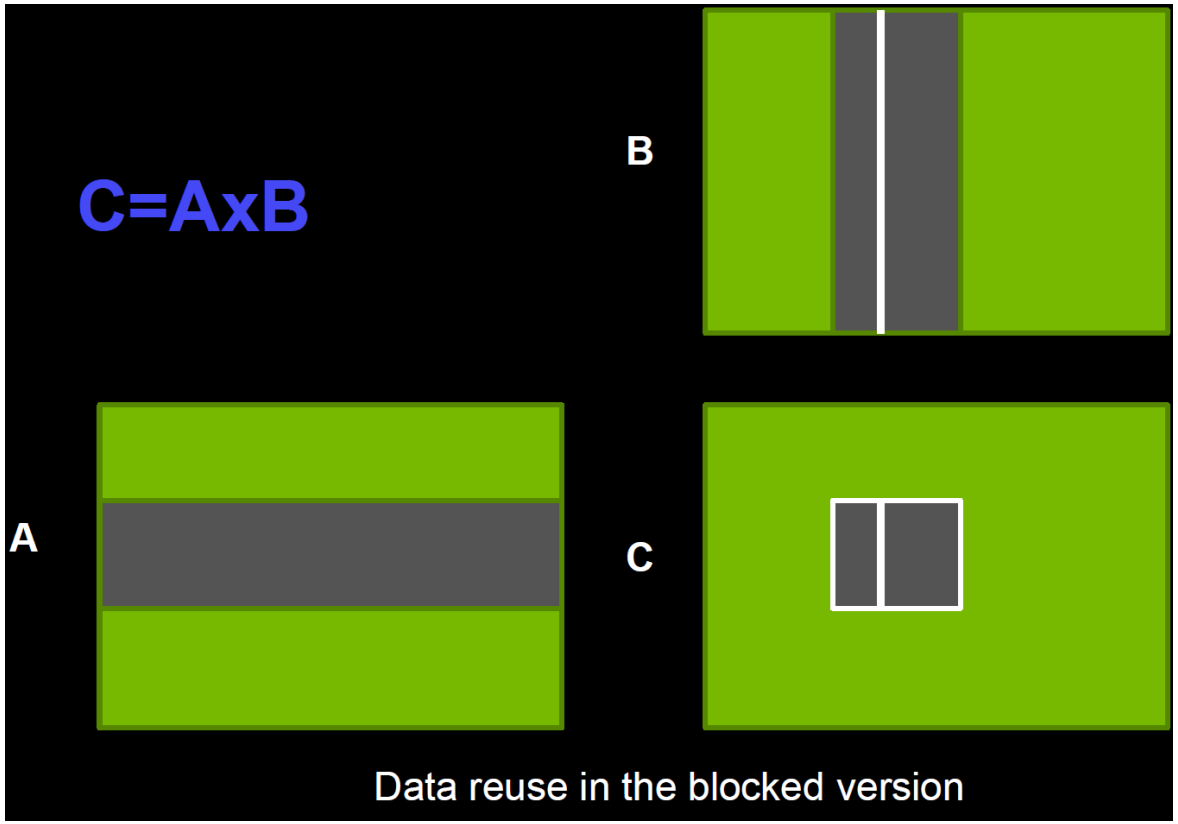
Example: Matrix Multiplication




```
__global__ void simpleMultiply(float* a,
                               float* b,
                               float* c,
                               int N)
{
  int row = threadIdx.x + blockIdx.x*blockDim.x;
  int col  = threadIdx.y + blockIdx.y*blockDim.y;
  float sum = 0.0f;
  for (int i = 0; i < N; i++) {
    sum += a[row*N+i] * b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

[NVIDIA]

Example: Matrix Multiplication




```
__global__ void coalescedMultiply(double*a,
                                  double* b,
                                  double*c,
                                  int N)
{
  __shared__ float aTile[TILE_DIM][TILE_DIM];
  __shared__ double bTile[TILE_DIM][TILE_DIM];

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for (int k = 0; k < N; k += TILE_DIM) {
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = k; i < k+TILE_DIM; i++)
      sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
  }
  c[row*N+col] = sum;
}
```

[NVIDIA]

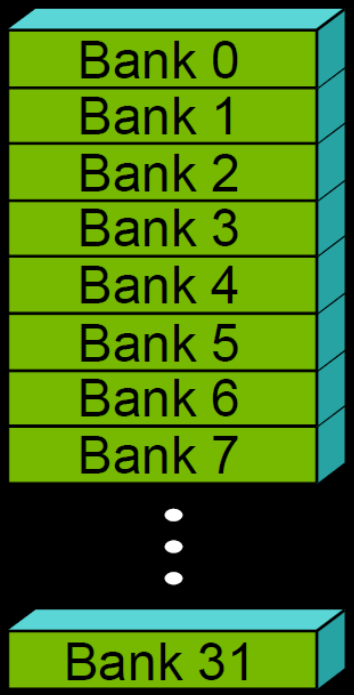
Example: Matrix Multiplication

M=N=K=512		
Optimization	C1060	C2050
A, B in global	12 Gflop/s	57 Gflop/s
A, B in shared	125 Gflop/s	181 Gflop/s

[NVIDIA]

Bank Conflicts in Shared Memory

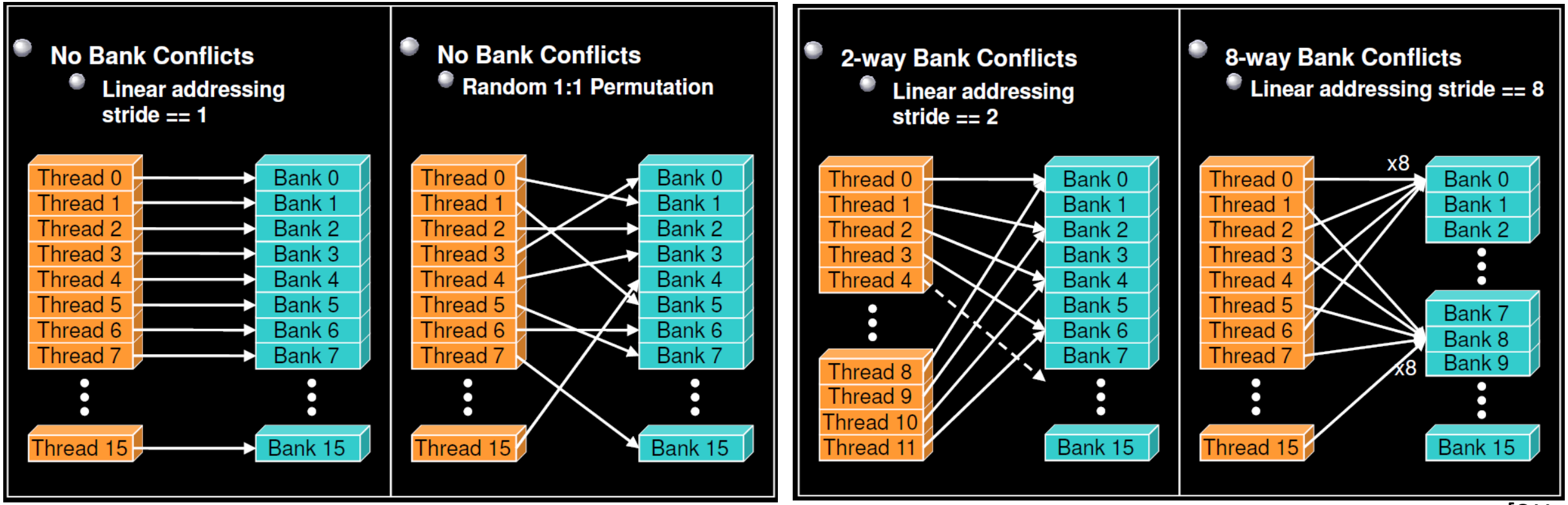
- Shared memory is divided into banks
  - Successive 32-bit words assigned to successive banks
  - Number of banks = 32 (Fermi)
- Bank conflict: two R/W fall in the same bank, the access will be serialized.
- Special cases
  - If all threads in a warp access the same word, one broadcast. Fermi can also do multi-broadcast.
  - If reading continuous byte/double, no conflict on Fermi



[NVIDIA]

Bank Conflicts

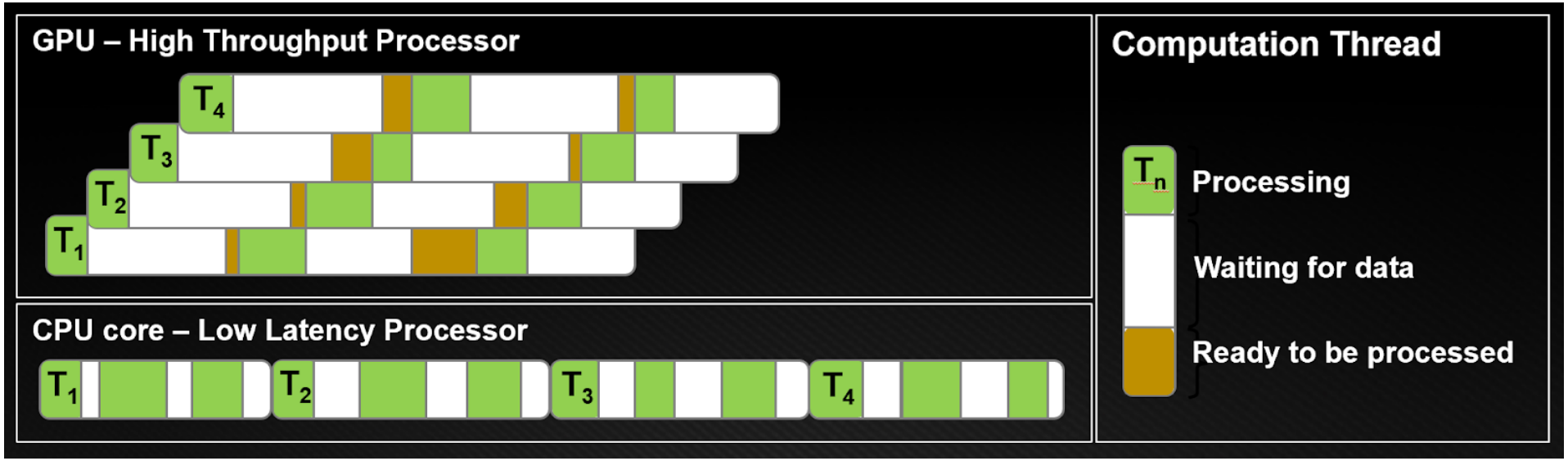
- Data layout is extremely important to minimize bank conflicts
  - Change data layout or use padding idea (array[N_BANK][N_BANK+1])



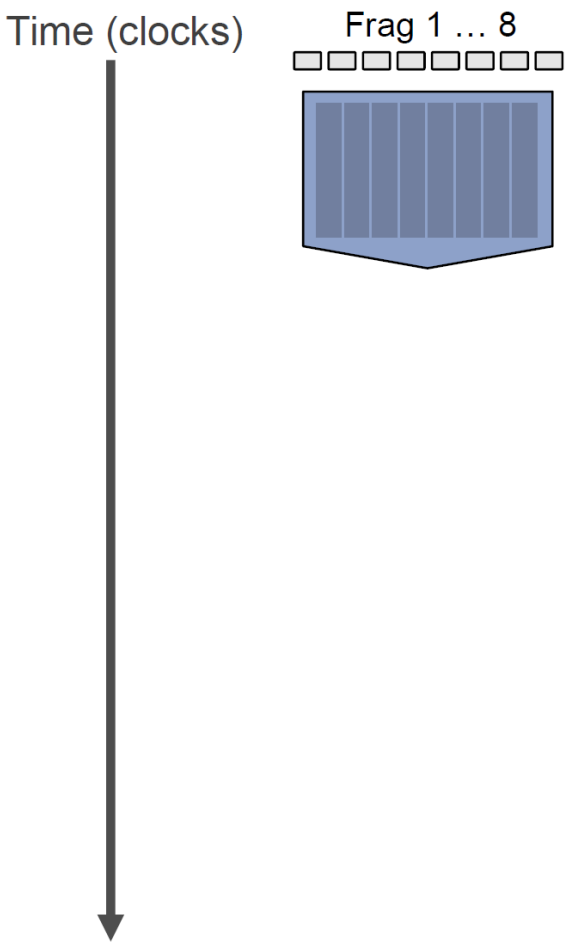
[S.Yoo]

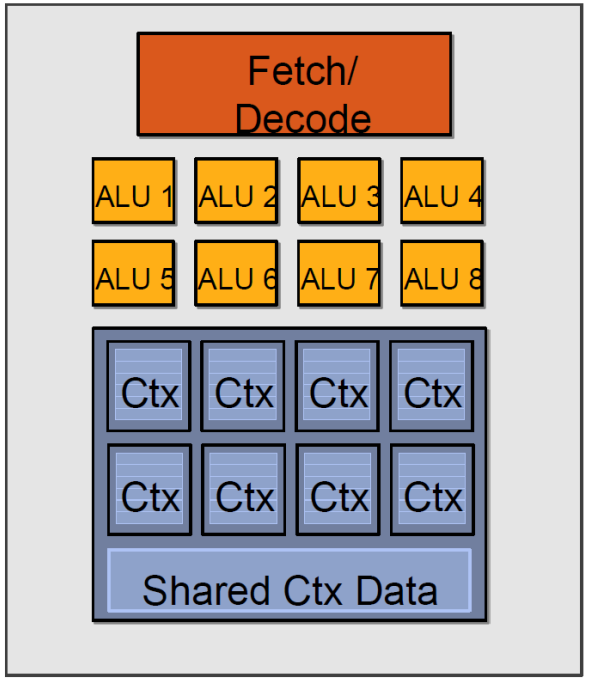
Latency Hiding

- GPU is throughput-optimized processor
  - Generate concurrent memory request to fully utilize the limited bandwidth
  - Multiple threads generate memory request concurrently
  - Thread of GPU is lightweight and each SM has multiple activated warps



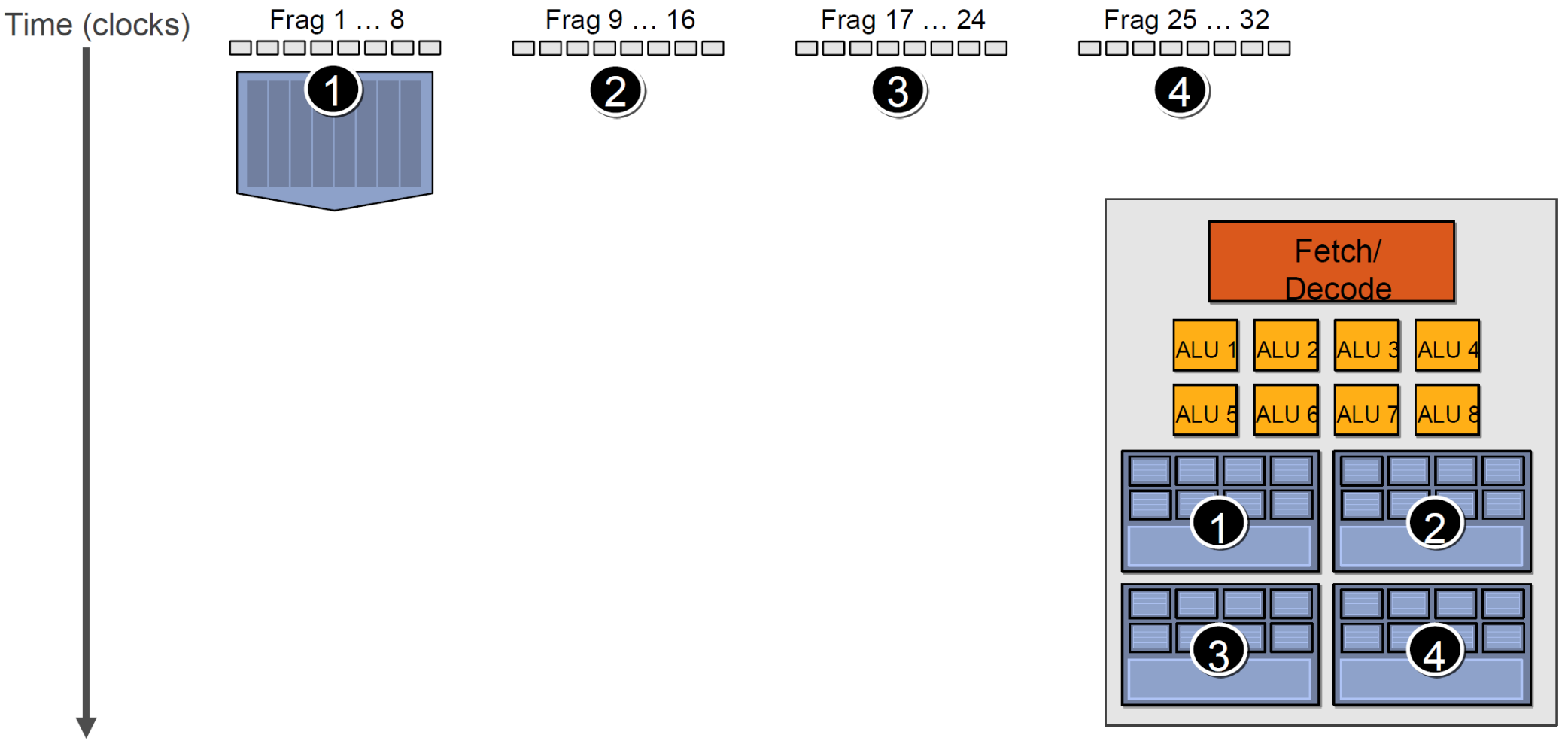
Running a Warp (=a set of threads in a block)





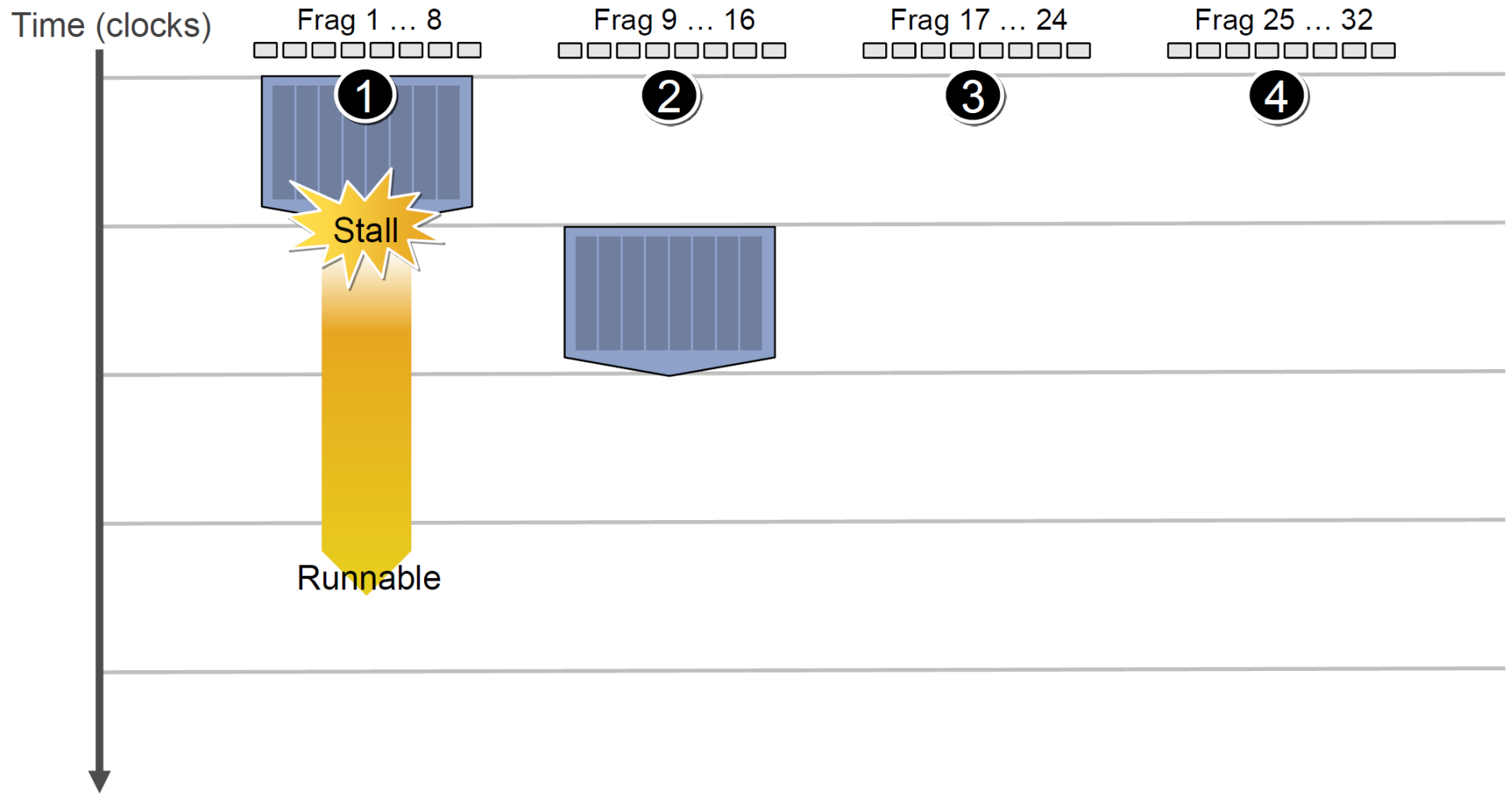
[AMD]
[S.Yoo]

Running 4 Warps



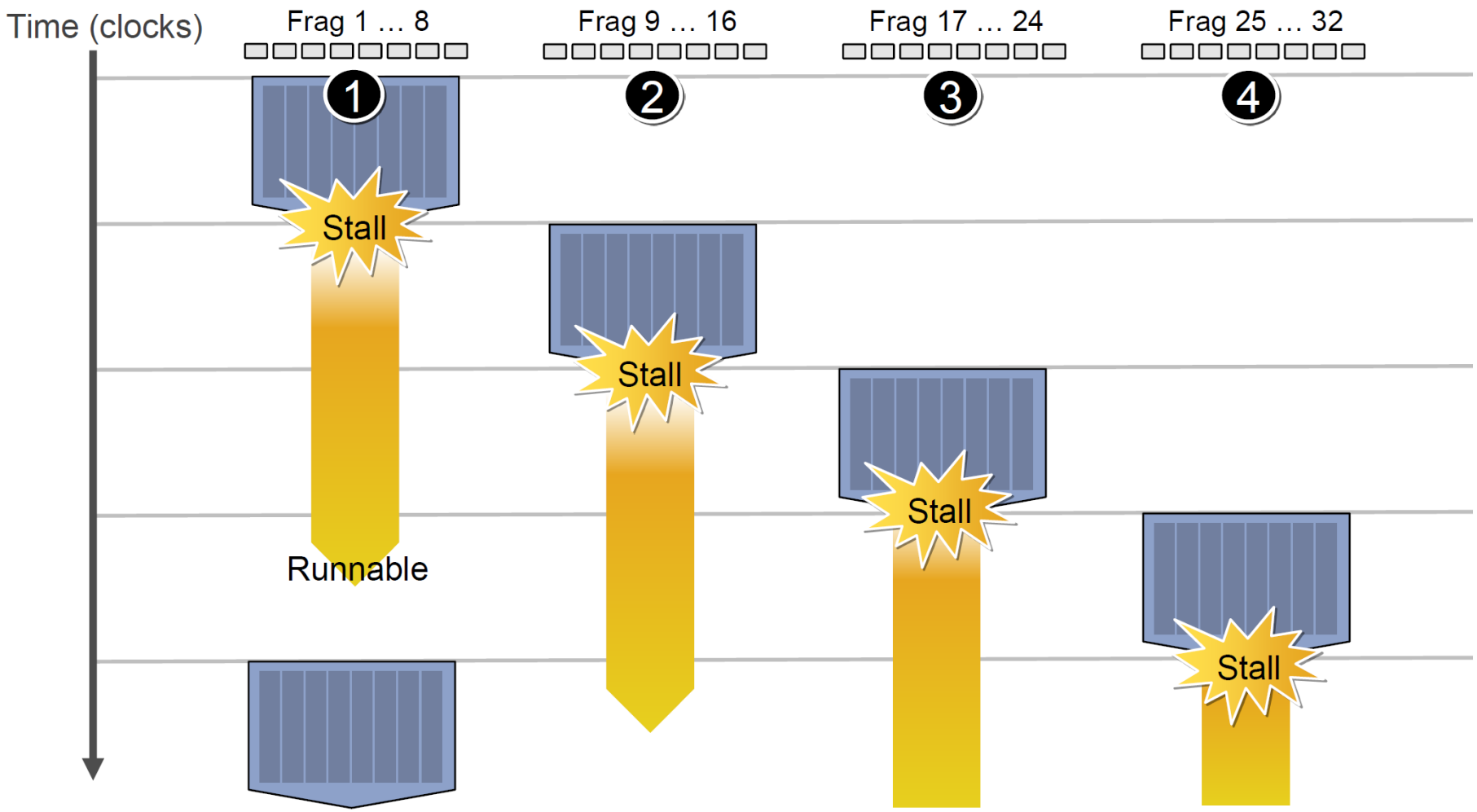
[AMD]
[S.Yoo]

Hiding Memory Stall Cycles by Running Other Runnable Warps



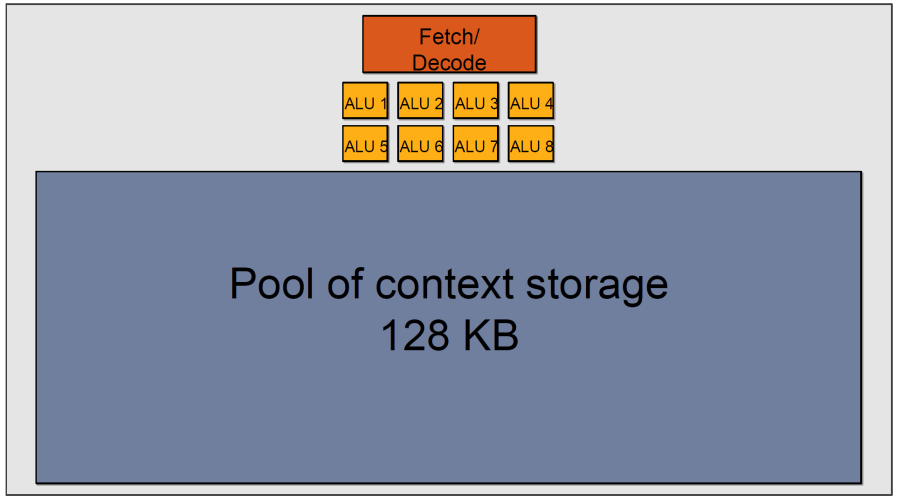
[AMD]
[S.Yoo]

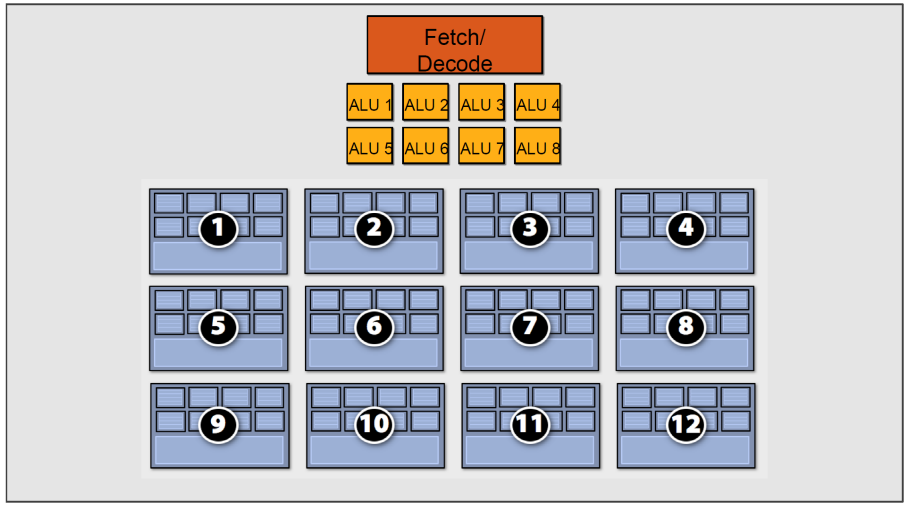
Throughput (# Computations/Sec) is Improved!

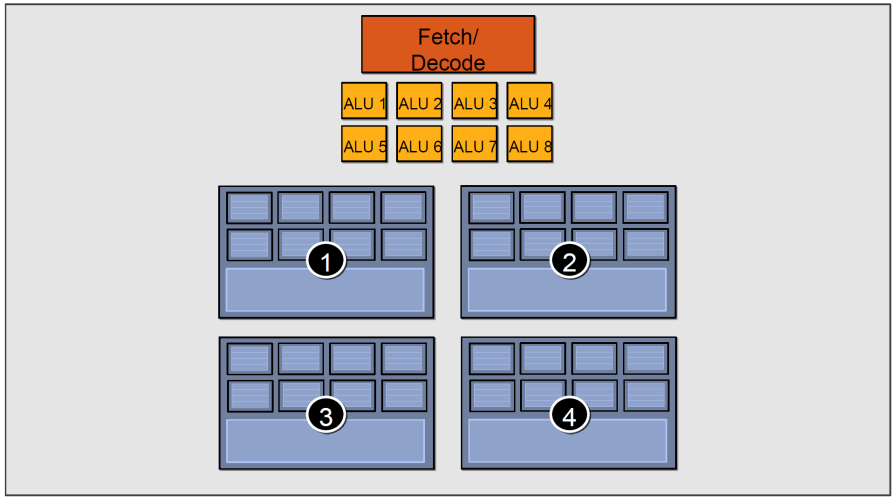


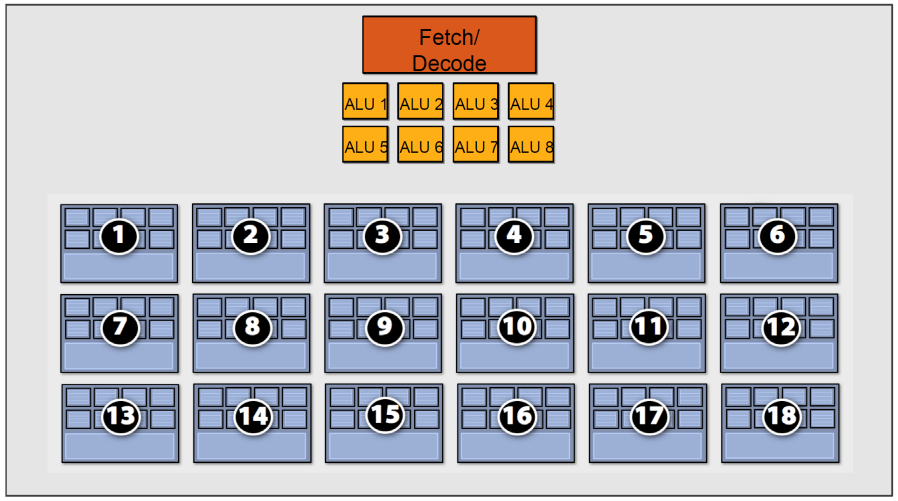
[AMD]
[S.Yoo]

Given a Fixed Amount of Register File, The More Threads → The Smaller Per-Thread Context to be Stored on Register Files



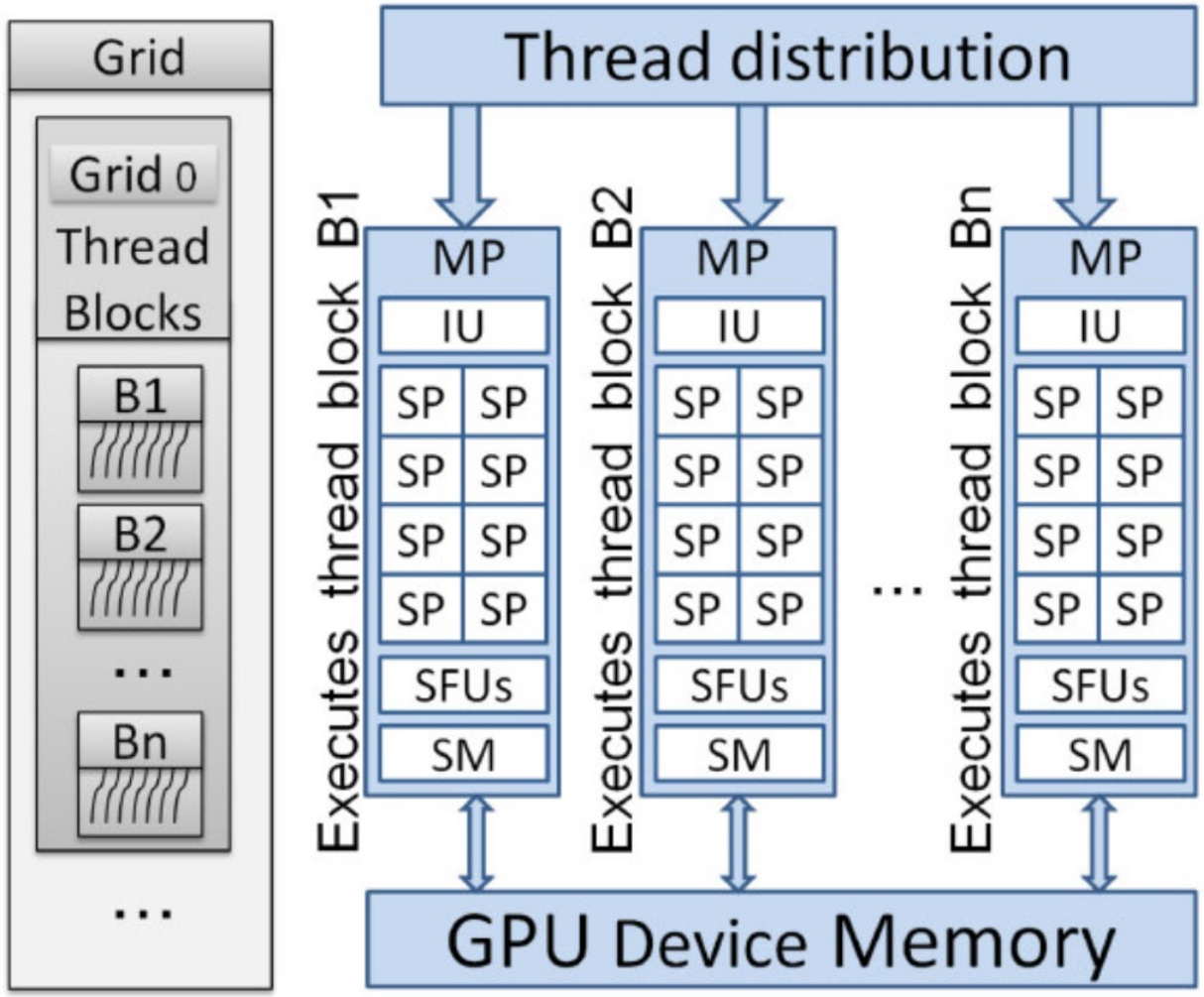






[AMD]
[S.Yoo]

Blocks are Allocated to SMs (by Hardware)



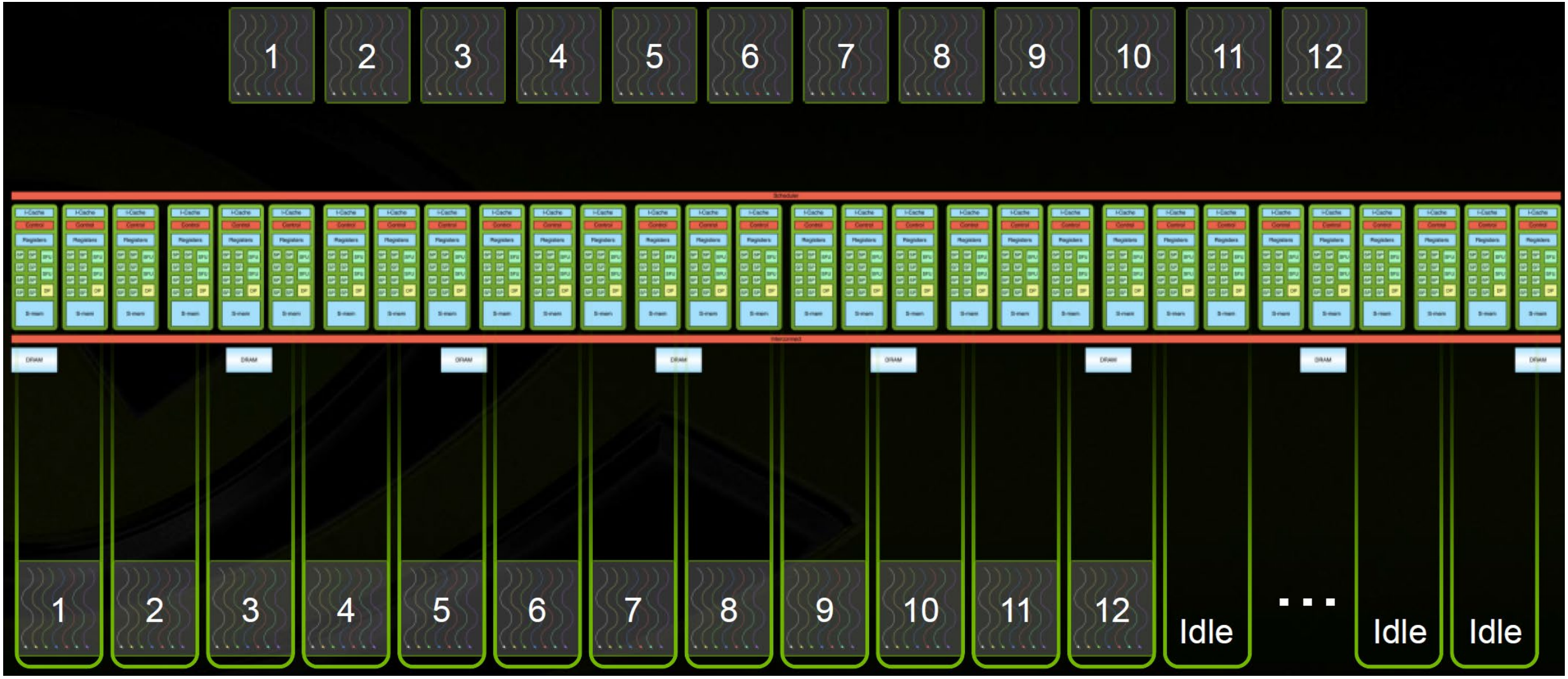
Block & Scalability – G84

[NVIDIA]

Block & Scalability – G80

[NVIDIA]

Block & Scalability – G200



[NVIDIA]

Grid/Block Size Heuristics

- # of blocks >> # of SM > 100 to scale well to future device
- Block size should be a multiple of 32 (warp size)
- Minimum: 64. I generally use 128 or 256. But use whatever is best for your app.
- Depends on the problem, do experiments!

[NVIDIA]

[NVIDIA]
[S.Yoo]

CUDA Compute Capability, https://en.wikipedia.org/wiki/CUDA

- Assume my CUDA program runs on compute capability 6.0
  - My program has 1K (1,024) threads and 64 32-bit registers/thread
- 1K threads can run in a single block
- 1K threads (with 64 registers/thread) can run in a single SM

Technical specifications	Compute capability (version)																		
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4		32			16	128	32	16	128	16	128	
Maximum dimensionality of grid of thread blocks	2				3														
Maximum x-dimension of a grid of thread blocks	65535				$2^{31}-1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																		
Maximum dimensionality of thread block	3																		
Maximum x- or y-dimension of a block	512				1024														
Maximum z-dimension of a block	64																		
Maximum number of threads per block	512				1024														
Warp size	32																		
Maximum number of resident blocks per multiprocessor	8				16					32								16	32
Maximum number of resident warps per multiprocessor	24		32		48					64								32	64
Maximum number of resident threads per multiprocessor	768		1024		1536					2048								1024	2048
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K		64 K		128 K		64 K								
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K		64 K			32 K	64 K		32 K		64 K		
Maximum number of 32-bit registers per thread	124				63					255									
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB		64 KB		96 KB	64 KB		96 KB (of 128)	64 KB (of 96)	164 KB (of 192)
Maximum amount of shared memory per thread block	48 KB															96 KB	48 KB	64 KB	160 KB

[NVIDIA]
[S.Yoo]

CUDA Compute Capability, https://en.wikipedia.org/wiki/CUDA

- My CUDA program runs on compute capability 6.0
  - My program has a block of 1K (1,024) threads and 64 32-bit registers/thread
- (Warp or thread) occupancy = 50% (=1K/2K)
  - Max # threads/SM = 2K

Technical specifications	Compute capability (version)																		
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128	16	128	
Maximum dimensionality of grid of thread blocks	2				3														
Maximum x-dimension of a grid of thread blocks	65535				$2^{31}-1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																		
Maximum dimensionality of thread block	3																		
Maximum x- or y-dimension of a block	512				1024														
Maximum z-dimension of a block	64																		
Maximum number of threads per block	512				1024														
Warp size	32																		
Maximum number of resident blocks per multiprocessor	8				16			32										16	32
Maximum number of resident warps per multiprocessor	24		32		48	64												32	64
Maximum number of resident threads per multiprocessor	768		1024		1536	2048												1024	2048
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K									
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K	64 K				32 K	64 K		32 K	64 K			
Maximum number of 32-bit registers per thread	124				63	255													
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB (of 128)		64 KB (of 96)	164 KB (of 192)
Maximum amount of shared memory per thread block	48 KB															96 KB	48 KB	64 KB	160 KB

[NVIDIA]
[S.Yoo]

CUDA Compute Capability, https://en.wikipedia.org/wiki/CUDA

- What if my program has 2K (2,048) threads and 64 registers/thread?
- My program needs 2K*64 registers > 64K registers/SM

Technical specifications	Compute capability (version)																		
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4		32			16	128	32	16	128	16	128	
Maximum dimensionality of grid of thread blocks	2				3														
Maximum x-dimension of a grid of thread blocks	65535				$2^{31}-1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																		
Maximum dimensionality of thread block	3																		
Maximum x- or y-dimension of a block	512				1024														
Maximum z-dimension of a block	64																		
Maximum number of threads per block	512				1024														
Warp size	32																		
Maximum number of resident blocks per multiprocessor	8				16					32								16	32
Maximum number of resident warps per multiprocessor	24		32		48	64												32	64
Maximum number of resident threads per multiprocessor	768		1024		1536	2048												1024	2048
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K									
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K		64 K			32 K	64 K		32 K	64 K			
Maximum number of 32-bit registers per thread	124				63	255													
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB (of 128)		64 KB (of 96)	164 KB (of 192)
Maximum amount of shared memory per thread block	48 KB															96 KB	48 KB	64 KB	160 KB

[NVIDIA]
[S.Yoo]

CUDA Compute Capability, https://en.wikipedia.org/wiki/CUDA

- What if my program has 2K (2,048) threads and requires 16B shared memory per thread?
  - 2k * 16B = 32 kB shared memory
  - Maximum 3k threads for CUDA version <= 6.2 / at max 1 active warp per SM
  - Maximum 10k threads for CUDA version == 8.0 / at max 5 active warp per SM

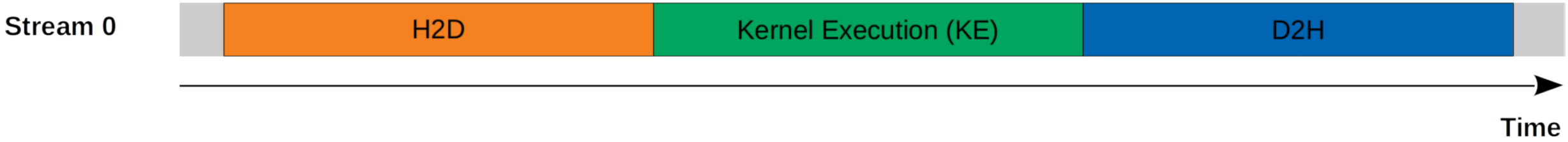
Technical specifications	Compute capability (version)																		
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128	16	128	
Maximum dimensionality of grid of thread blocks	2				3														
Maximum x-dimension of a grid of thread blocks	65535				$2^{31}-1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																		
Maximum dimensionality of thread block	3																		
Maximum x- or y-dimension of a block	512				1024														
Maximum z-dimension of a block	64																		
Maximum number of threads per block	512				1024														
Warp size	32																		
Maximum number of resident blocks per multiprocessor	8				16					32								16	32
Maximum number of resident warps per multiprocessor	24		32		48	64												32	64
Maximum number of resident threads per multiprocessor	768		1024		1536	2048												1024	2048
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K									
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K	64 K				32 K	64 K		32 K	64 K			
Maximum number of 32-bit registers per thread	124				63	255													
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			112 KB	64 KB	96 KB		64 KB		96 KB	64 KB	96 KB (of 128)		64 KB (of 96)	164 KB (of 192)
Maximum amount of shared memory per thread block	48 KB															96 KB	48 KB	64 KB	160 KB

CUDA Stream

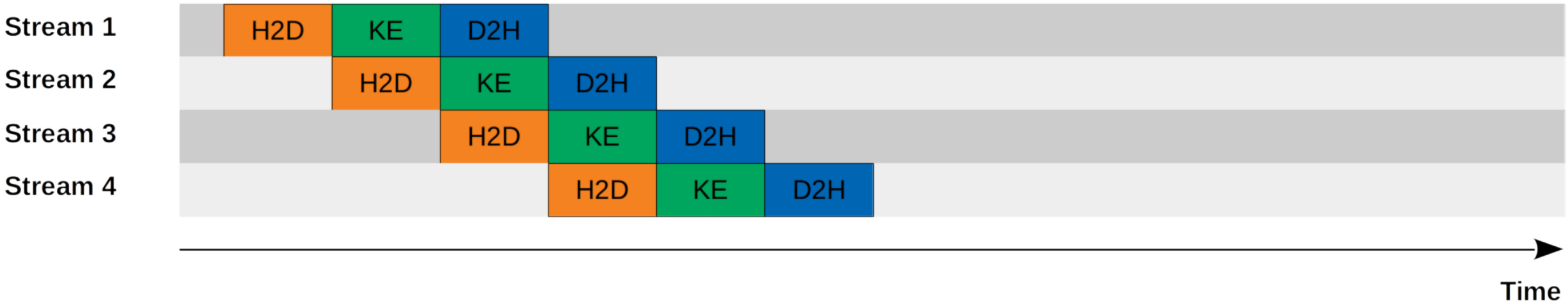
- Default API:
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- Streams and async functions provide:
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
  - Concurrent kernel in Fermi
- Stream = sequence of operations that execute in issue-order on GPU
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped

CUDA Stream

Serial Model



Concurrent Model



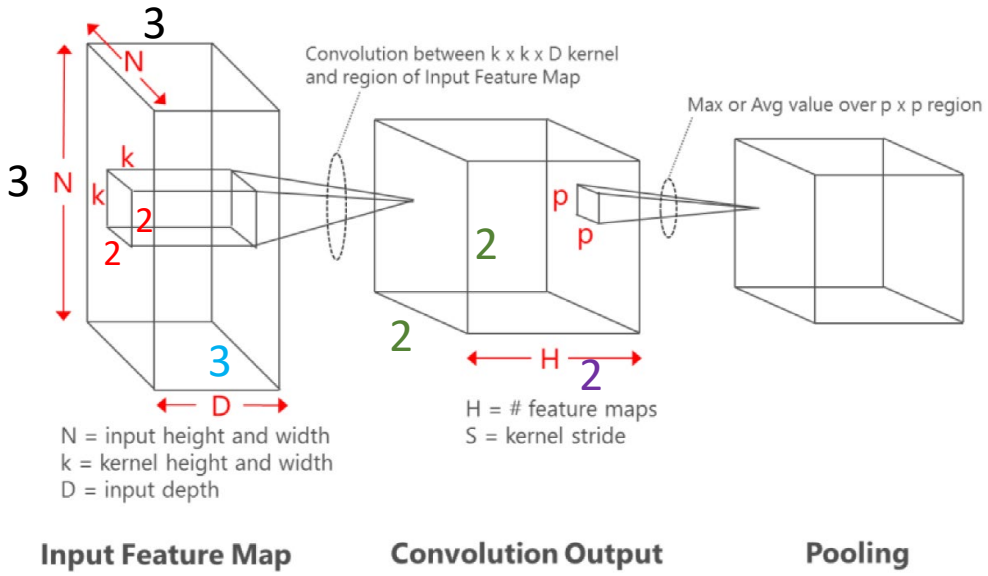
[NVIDIA]

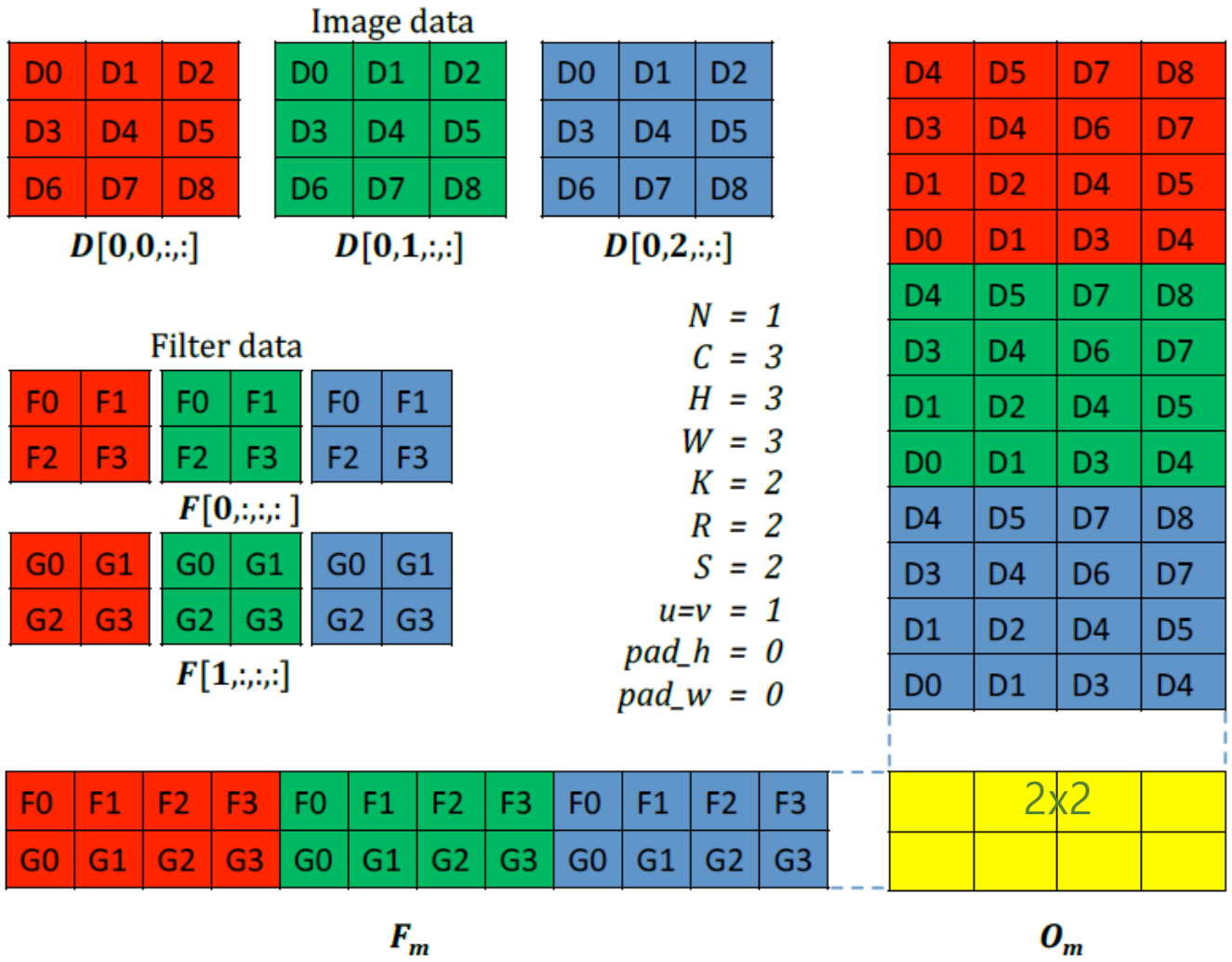
Matrix Multiplication & Convolution Computation

[Chetlur 2014]
[S.Yoo]

Convolution with Matrix Multiplication (called Convolution Lowering)

- Input: 3x3x3
- Output: 2x2x2
- Convolutional kernel: 3x2x2





[Chetlur 2014]
[S.Yoo]

Convolution with Matrix Multiplication (called Convolution Lowering)

- Input: 3x3x3
- Output: 2x2x2
- Convolutional kernel: 3x2x2

3: vertical dimension

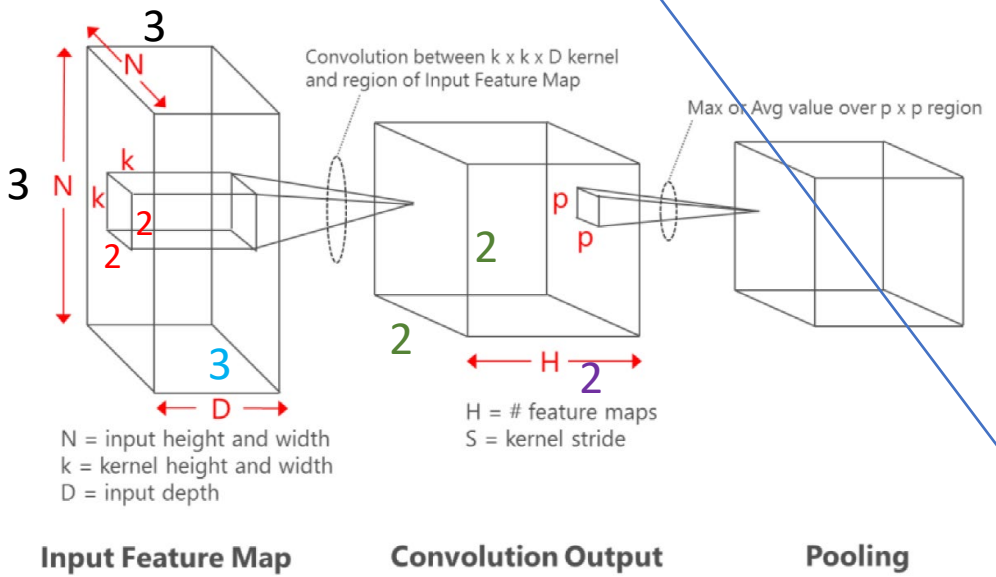


Image data

D0	D1	D2
D3	D4	D5
D6	D7	D8

$D[0,0,:,:]$

D0	D1	D2
D3	D4	D5
D6	D7	D8

$D[0,1,:,:]$

D0	D1	D2
D3	D4	D5
D6	D7	D8

$D[0,2,:,:]$

2

Filter data

F0	F1	F0	F1	F0	F1
F2	F3	F2	F3	F2	F3

$F[0,:,:,:]$

G0	G1	G0	G1	G0	G1
G2	G3	G2	G3	G2	G3

$F[1,:,:,:]$

2

$N = 1$
$C = 3$
$H = 3$
$W = 3$
$K = 2$
$R = 2$
$S = 2$
$u=v = 1$
$pad\_h = 0$
$pad\_w = 0$

F0	F1	F2	F3	F0	F1	F2	F3	F0	F1	F2	F3
G0	G1	G2	G3	G0	G1	G2	G3	G0	G1	G2	G3

F_m

2

D4	D5	D7	D8
D3	D4	D6	D7
D1	D2	D4	D5
D0	D1	D3	D4
D4	D5	D7	D8
D3	D4	D6	D7
D1	D2	D4	D5
D0	D1	D3	D4
D4	D5	D7	D8
D3	D4	D6	D7
D1	D2	D4	D5
D0	D1	D3	D4

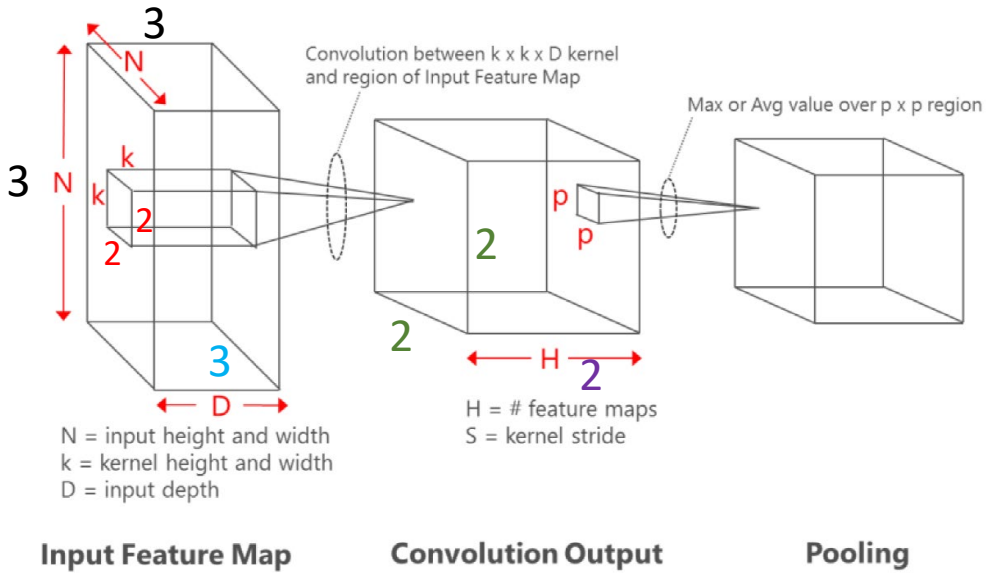
	2x2	

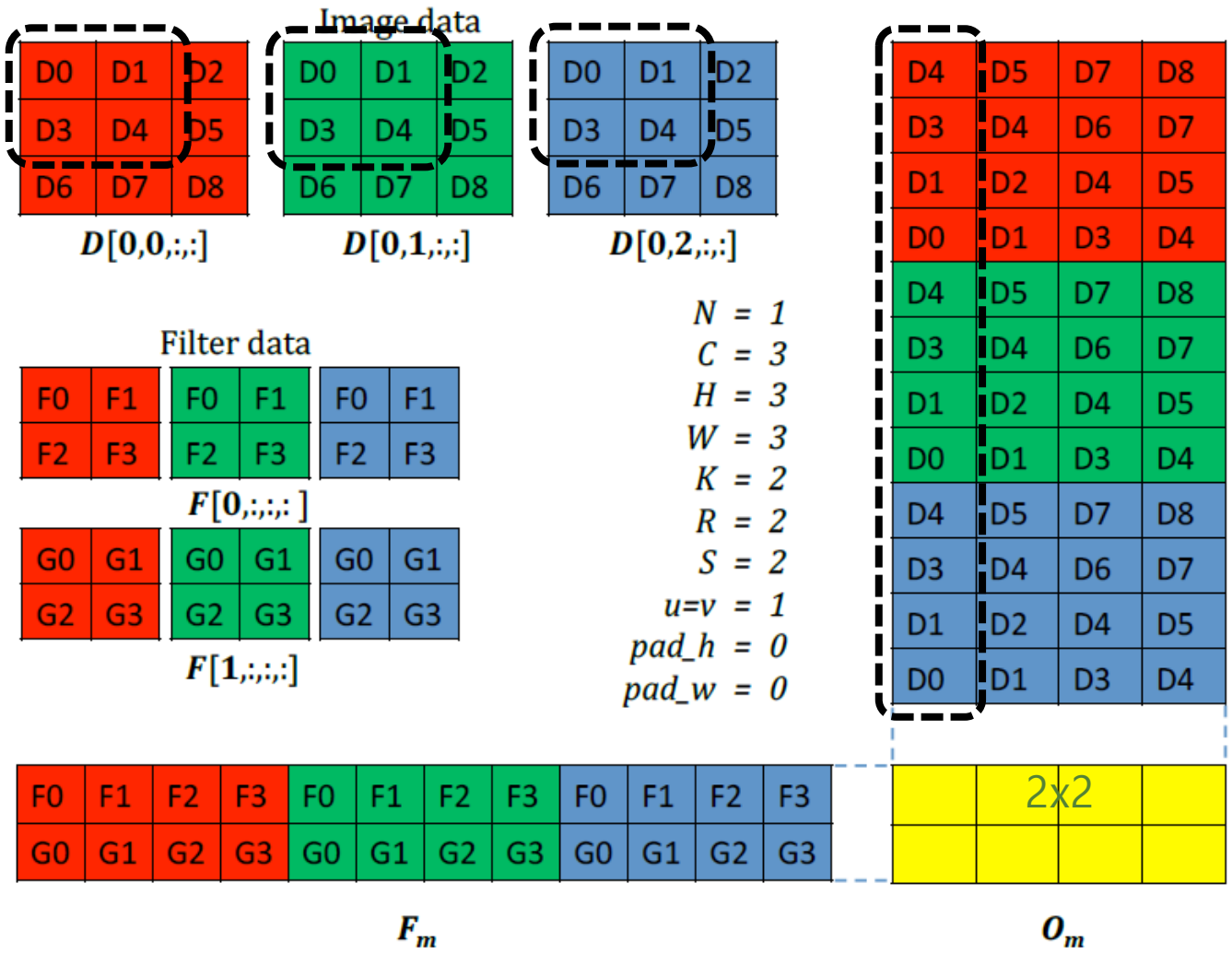
O_m

[Chetlur 2014]
[S.Yoo]

Convolution with Matrix Multiplication (called Convolution Lowering)

- Input: 3x3x3
- Output: 2x2x2
- Convolutional kernel: 3x2x2

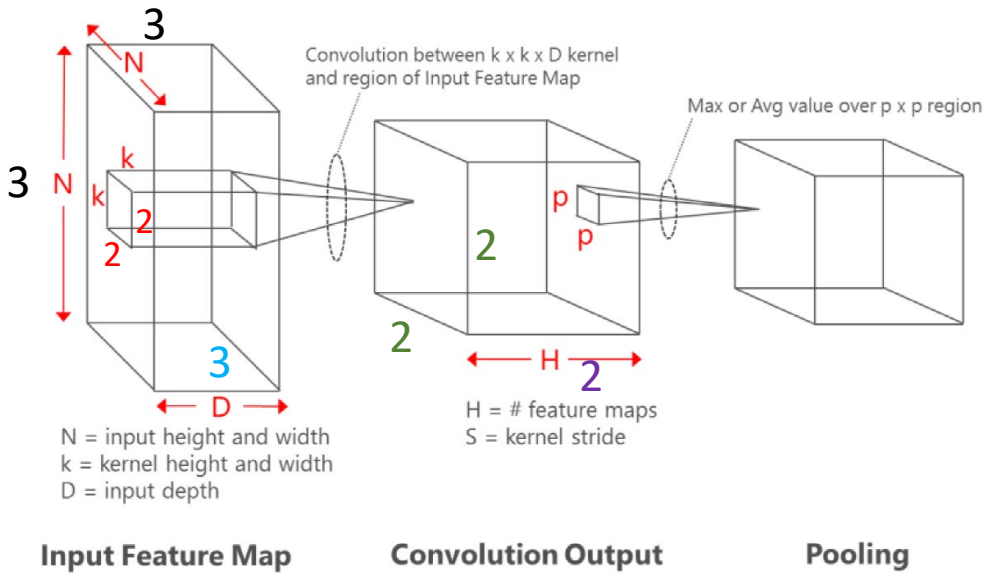


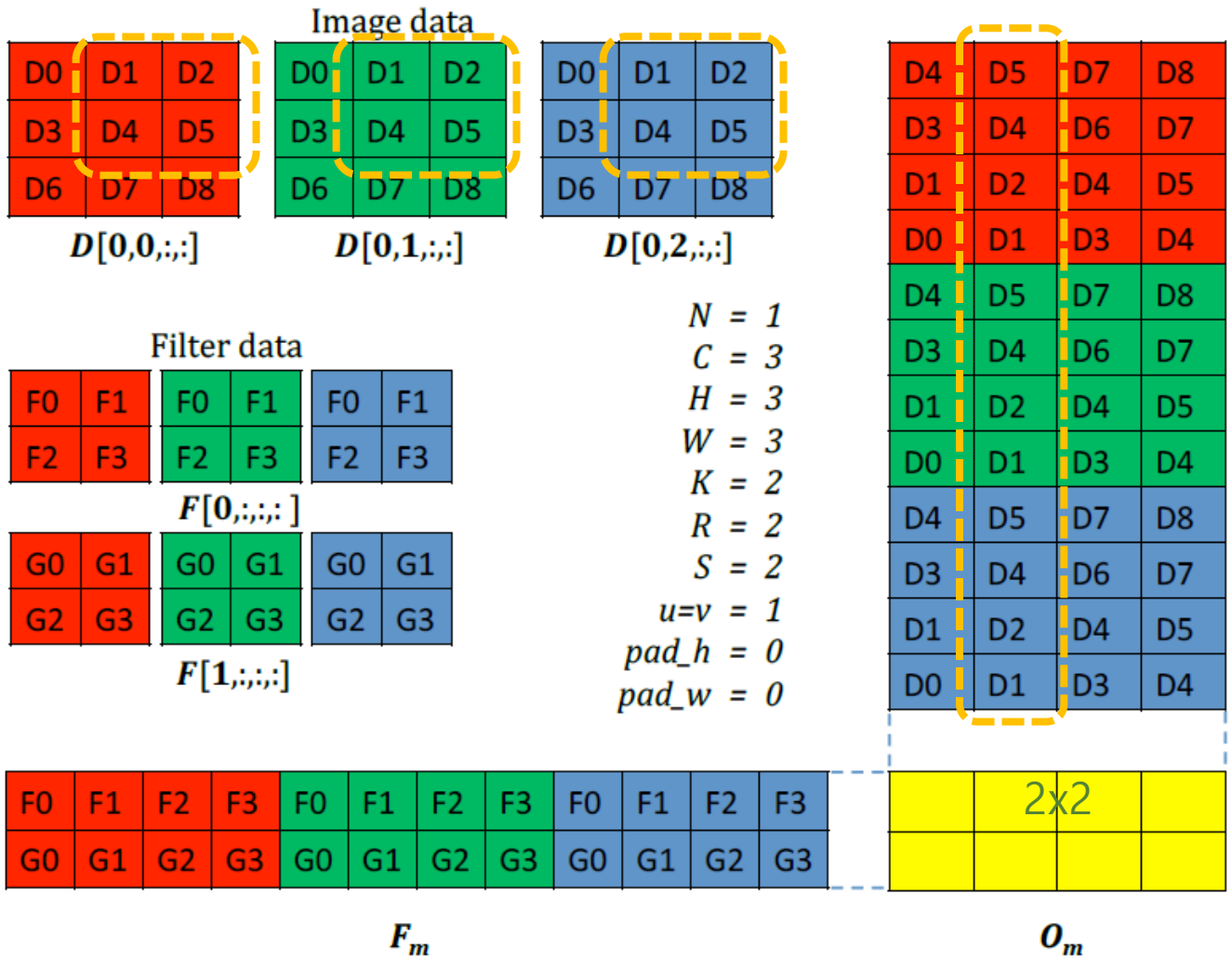


[Chetlur 2014]
[S.Yoo]

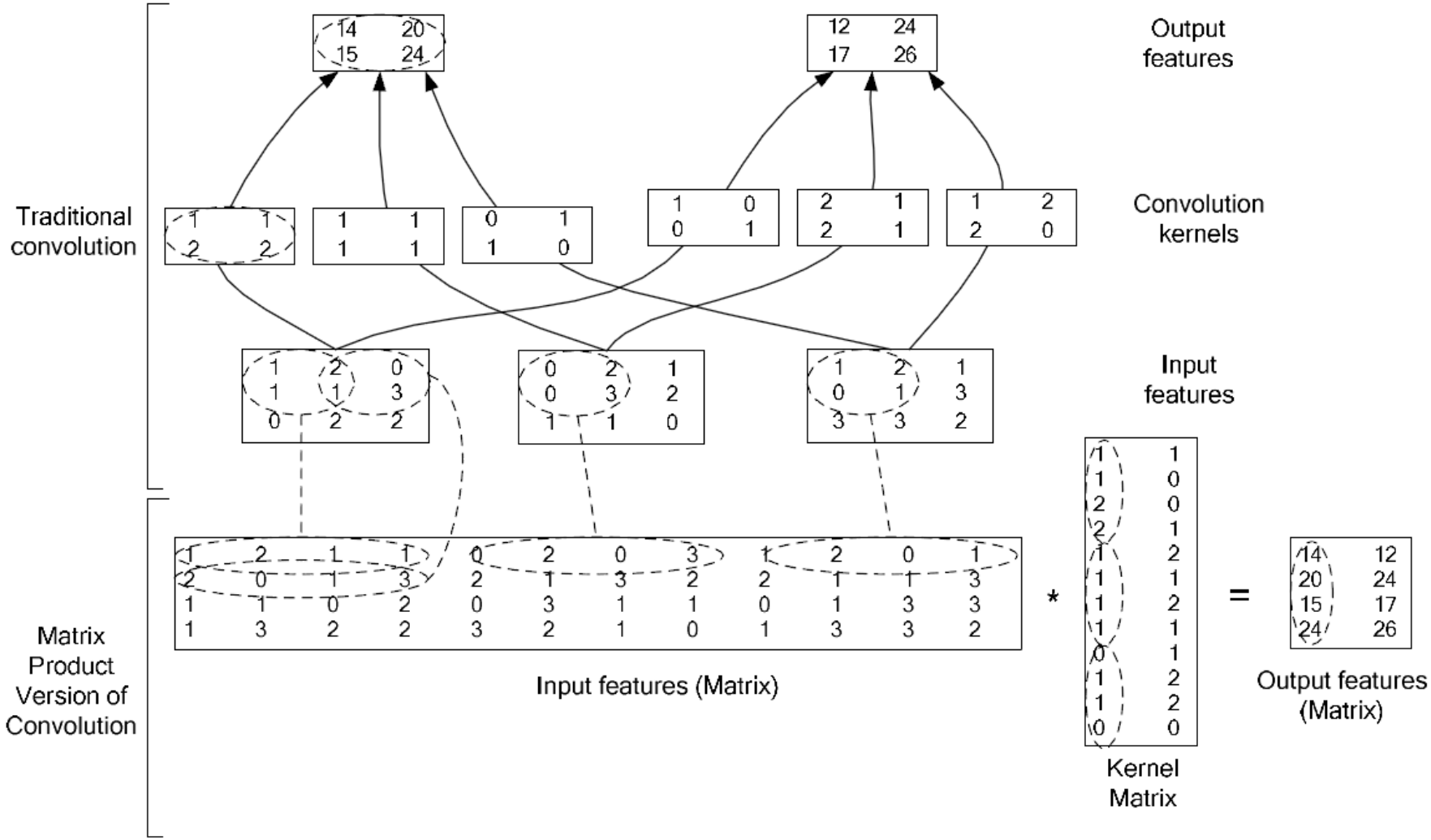
Convolution with Matrix Multiplication (called Convolution Lowering)

- Input: 3x3x3
- Output: 2x2x2
- Convolutional kernel: 3x2x2





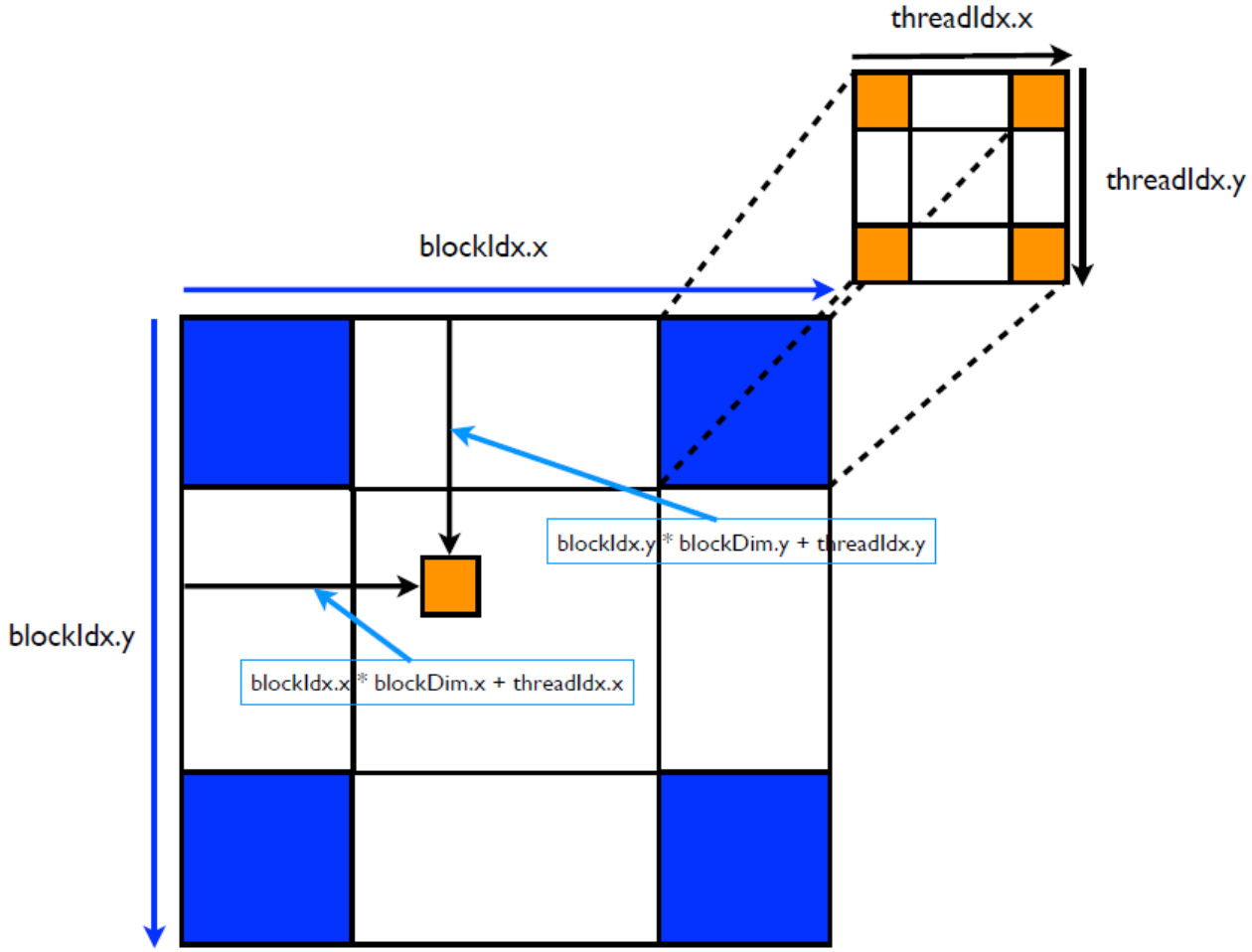
Another Example



[Chellapilar]

A Complete Example: Matrix Addition

- 2D block and grid



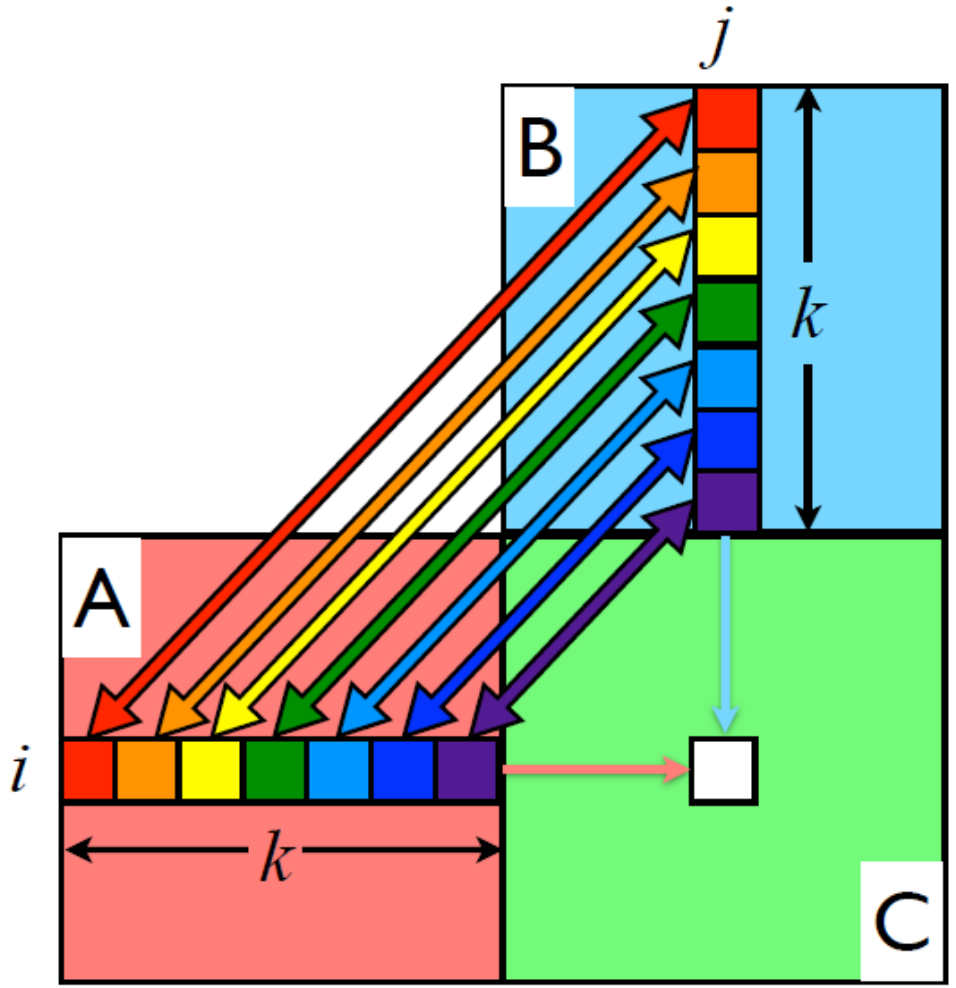

```
#define N 512
#define BLOCK_DIM 512

__global__ void matrixAdd (int *a, int *b, int *c);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 int size = N * N * sizeof(int);

 // initialize a and b with real values (NOT SHOWN)

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
 dim3 dimGrid((int)ceil(N/dimBlock.x),(int)ceil(N/dimBlock.y));

 matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixAdd (int *a, int *b, int *c) {
 int col = blockIdx.x * blockDim.x + threadIdx.x;
 int row = blockIdx.y * blockDim.y + threadIdx.y;

 int index = col + row * N;

 if (col < N && row < N) {
  c[index] = a[index] + b[index];
 }
}
```

[S.Yoo]

Matrix Multiplication

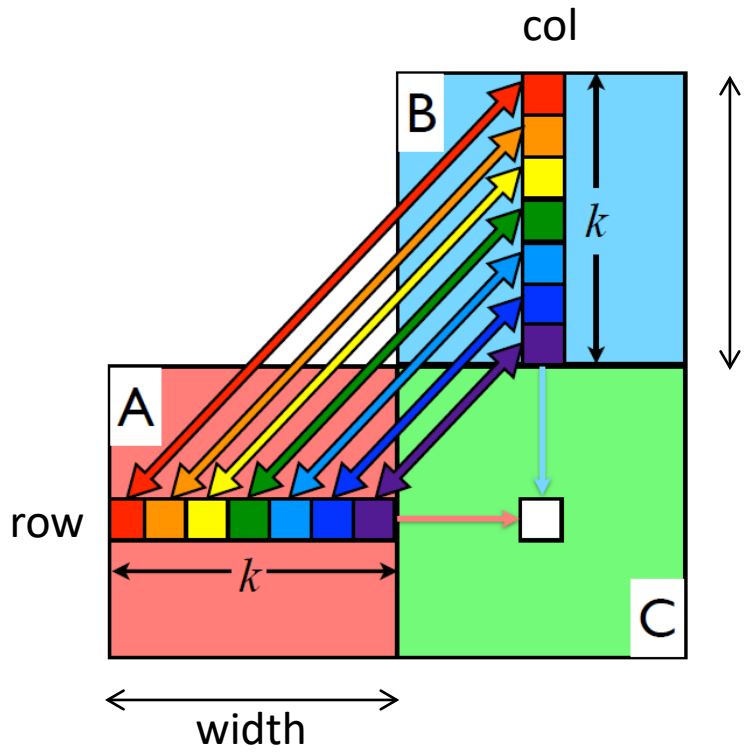
- How to parallelize this?

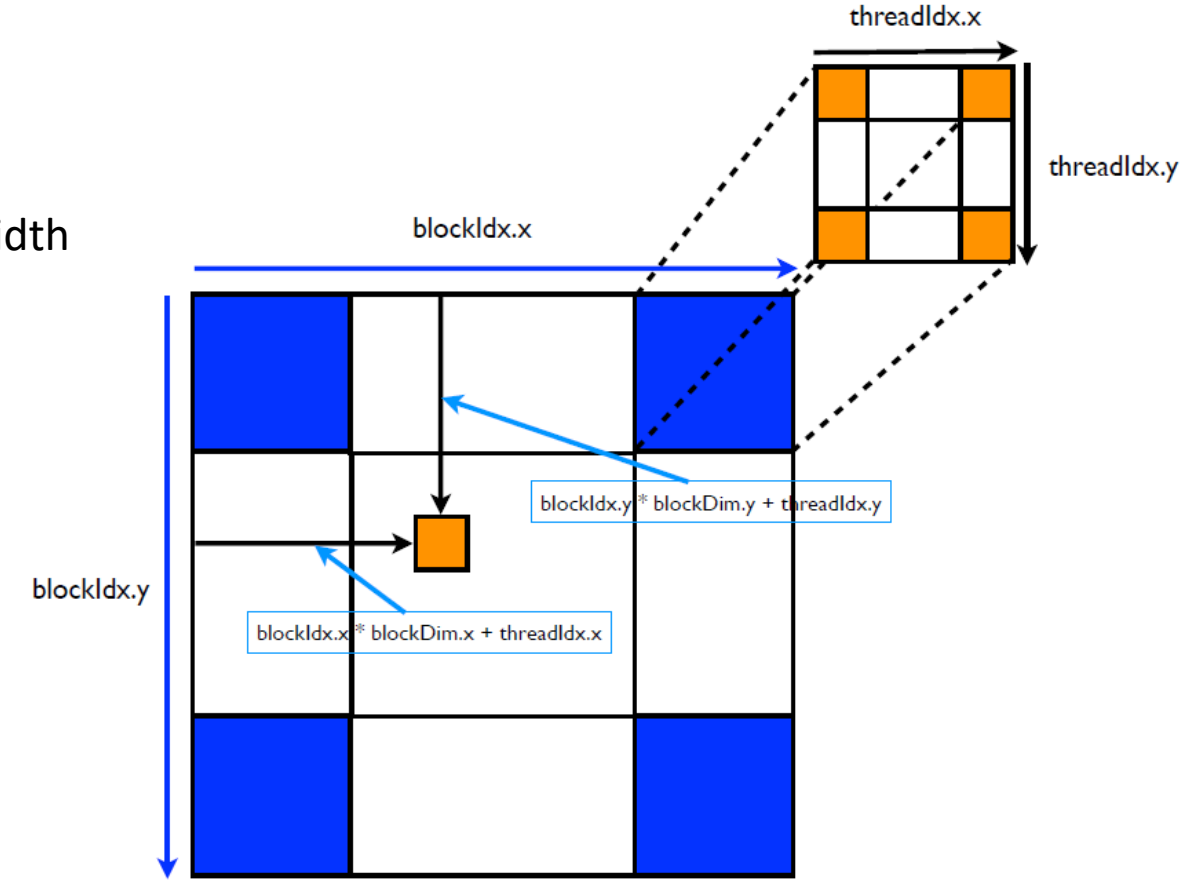
```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;
      }
      c[i][j] = sum;
    }
  }
}
```



- One thread for each output value

[S.Yoo]






```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;
      }
      c[i][j] = sum;
    }
  }
}
```

```
__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```

[S.Yoo]

Complete Example

- How many threads?
- N=16
- N*N, i.e., 256 threads run.
- Each of thread produces one element of the output matrix

```
#define N 16
#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 // initialize matrices a and b with appropriate values

 int size = N * N * sizeof(int);
 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```

[S.Yoo]

Complete Example

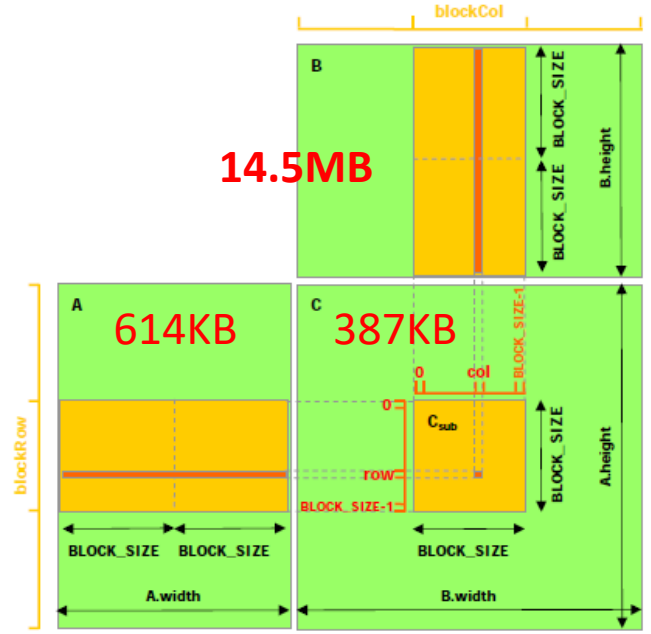
- What if N = 1024?
  - Floating point type = 4B
  - Size of matrix a, b, c = 4MB each
  - Typical L1 cache size = 32KB~64KB
  - L1 cache can contain only a portion of matrices → frequent cache misses!

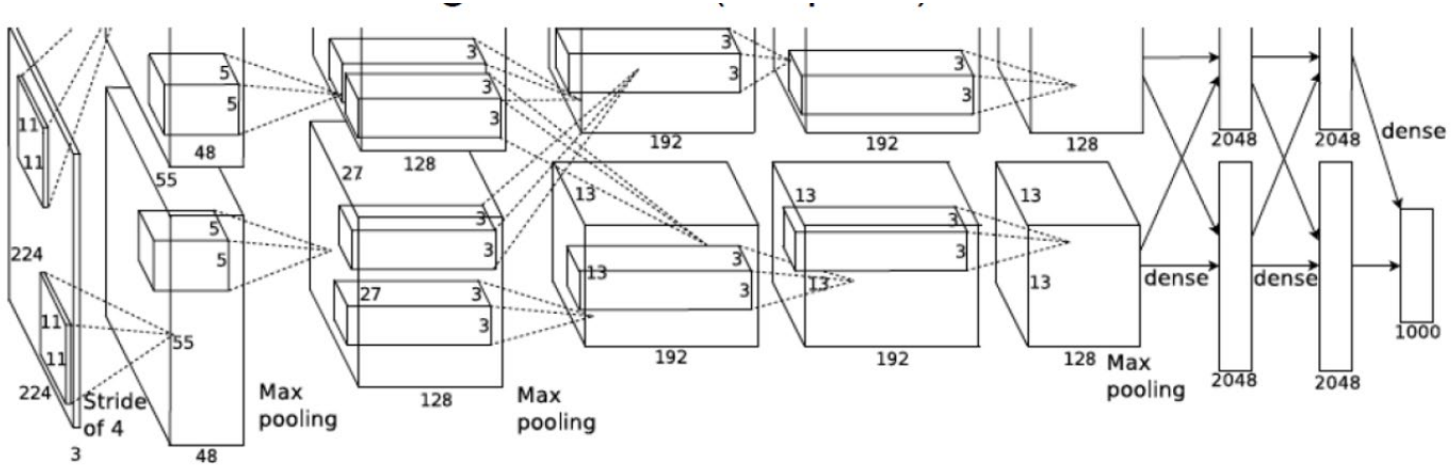
```
#define N 16
#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 // initialize matrices a and b with appropriate values

 int size = N * N * sizeof(int);
 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```

[S.Yoo]

Matrix Size vs. GPU Cache Size

- Example: 2nd convolutional layer on AlexNet
- Input size = 55x55x48x4B = 580KB
  - Input matrix size = 580KBx5x5 = 14.5MB
- Output size = 27x27x128x4B = 387KB
- Kernel size = 48x5x5x128x4B = 614KB



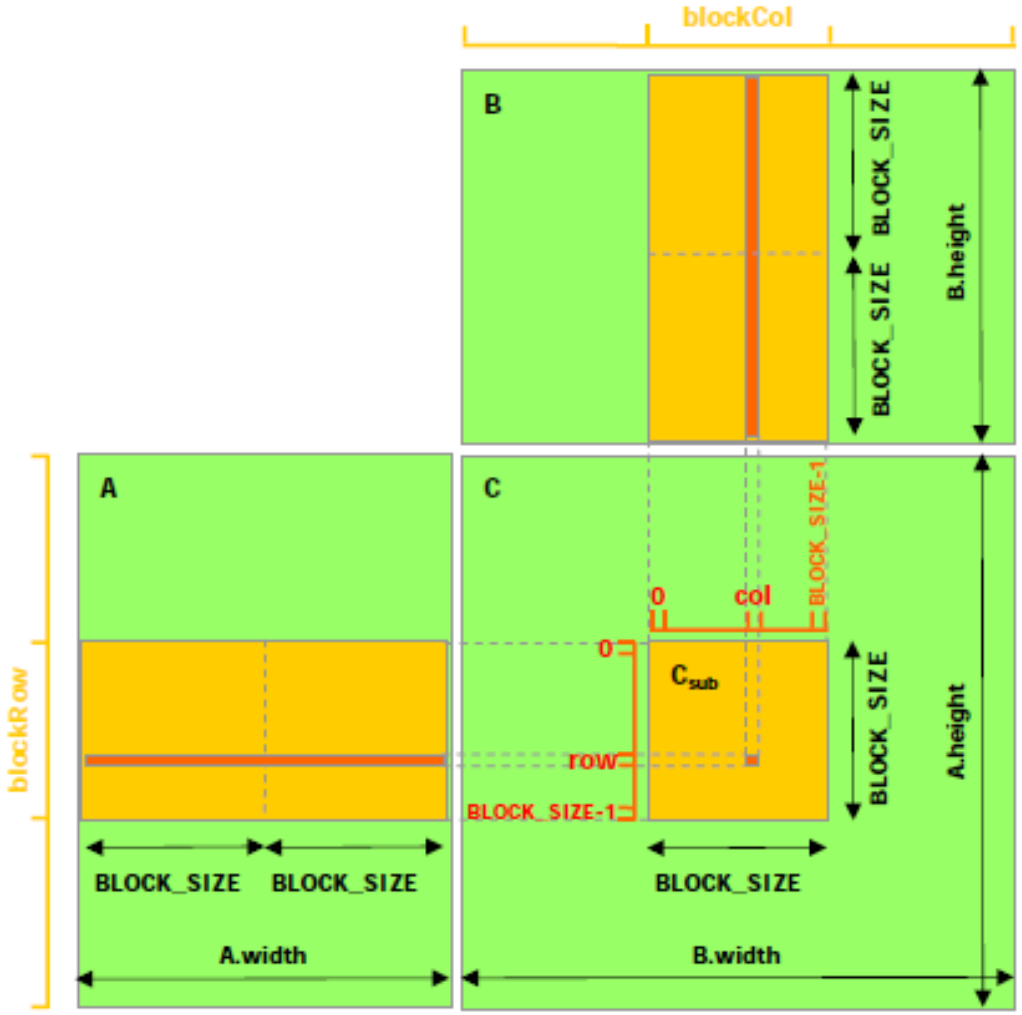


[S.Yoo]

Data Reuse with Shared Memory

- Blocking or tiling for matrix multiplication
  - Divide matrix into several tiles
  - Reuse each tile repeatedly based on cache or scratch memory

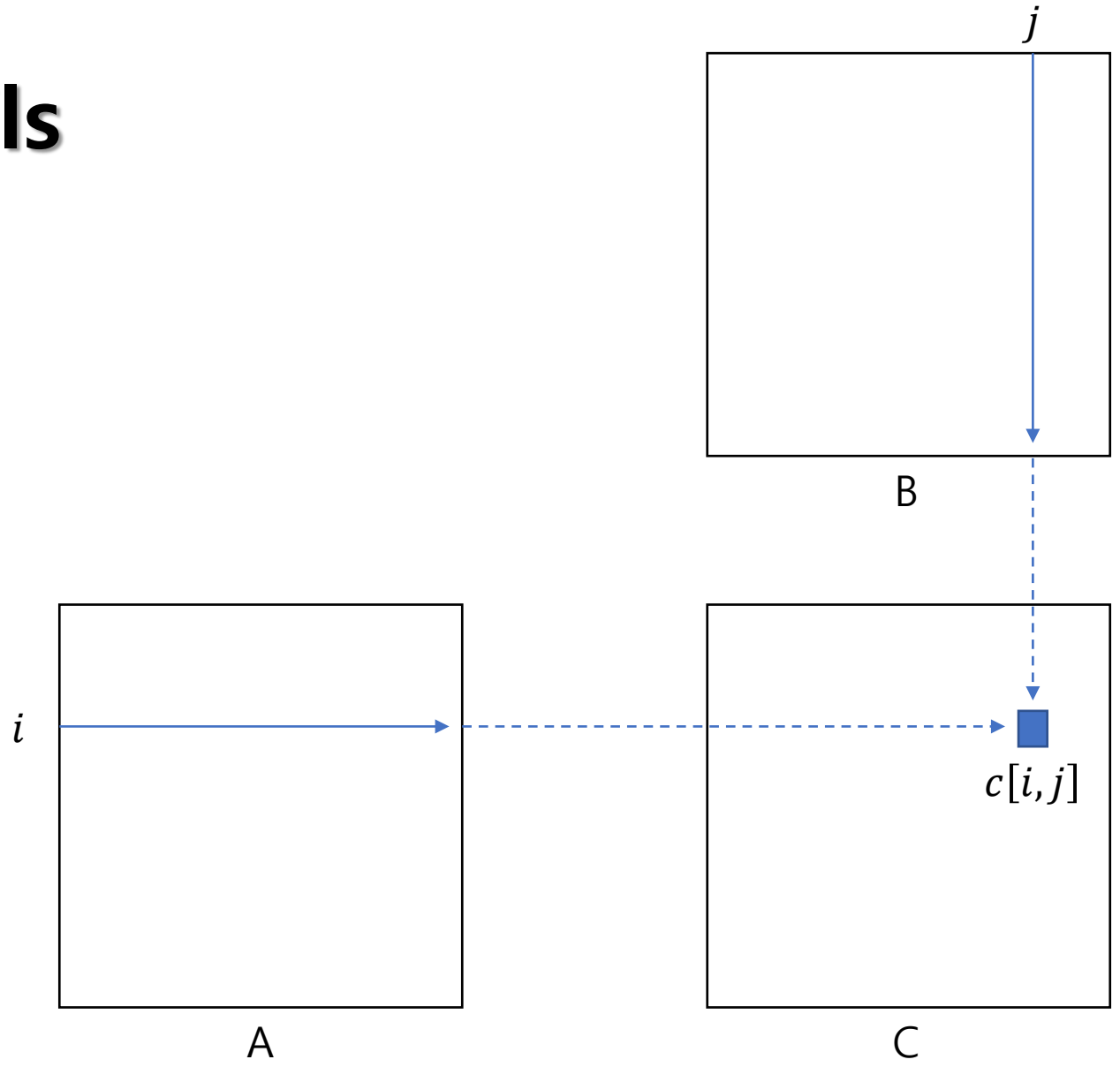
  - Shared memory in the case of NVIDIA SM



[CUDA Programming Guide]

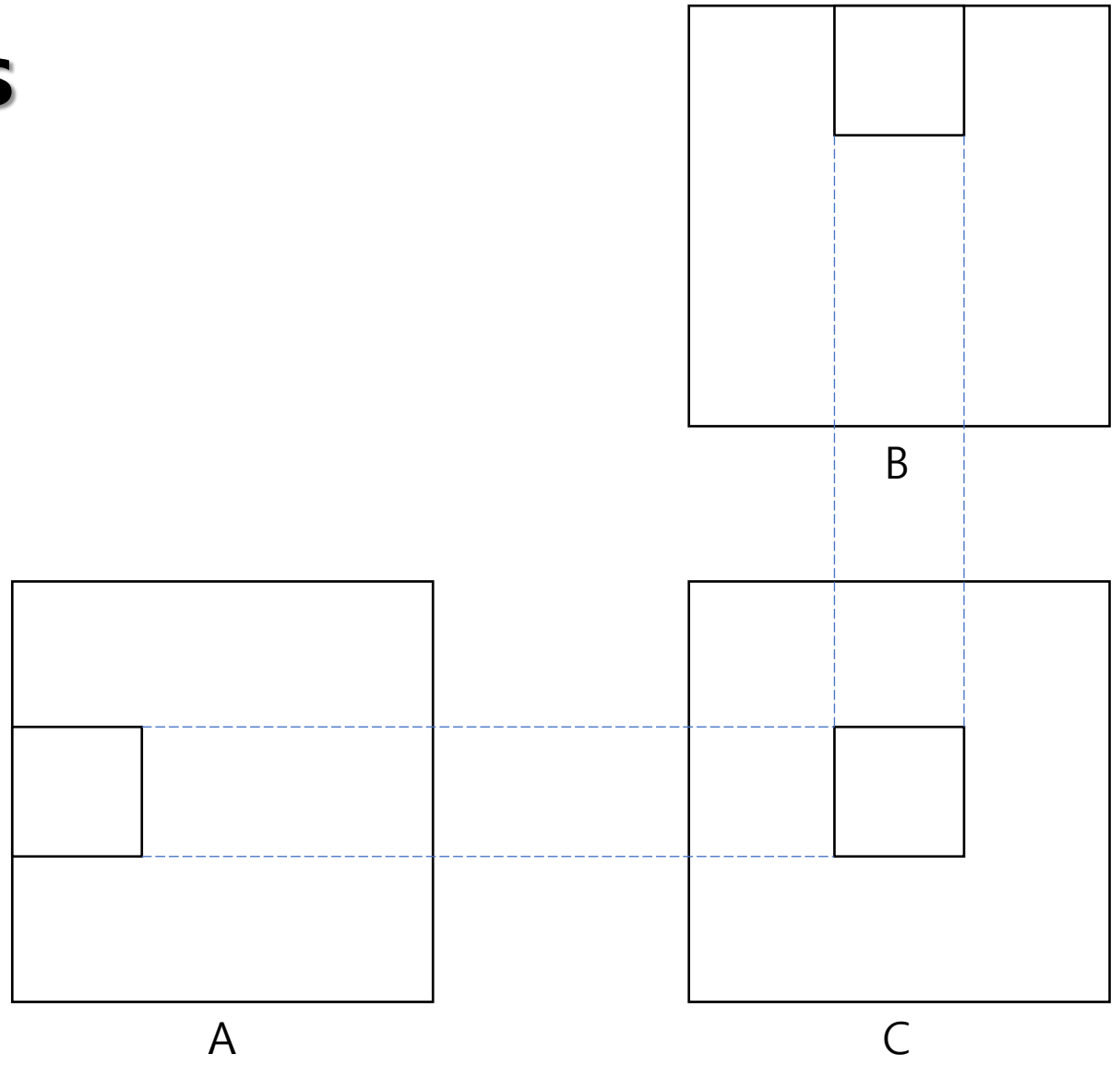
Matrix Tiling Details

- $C[i,j] = \Sigma_k A[i,k] \cdot B[k,j]$



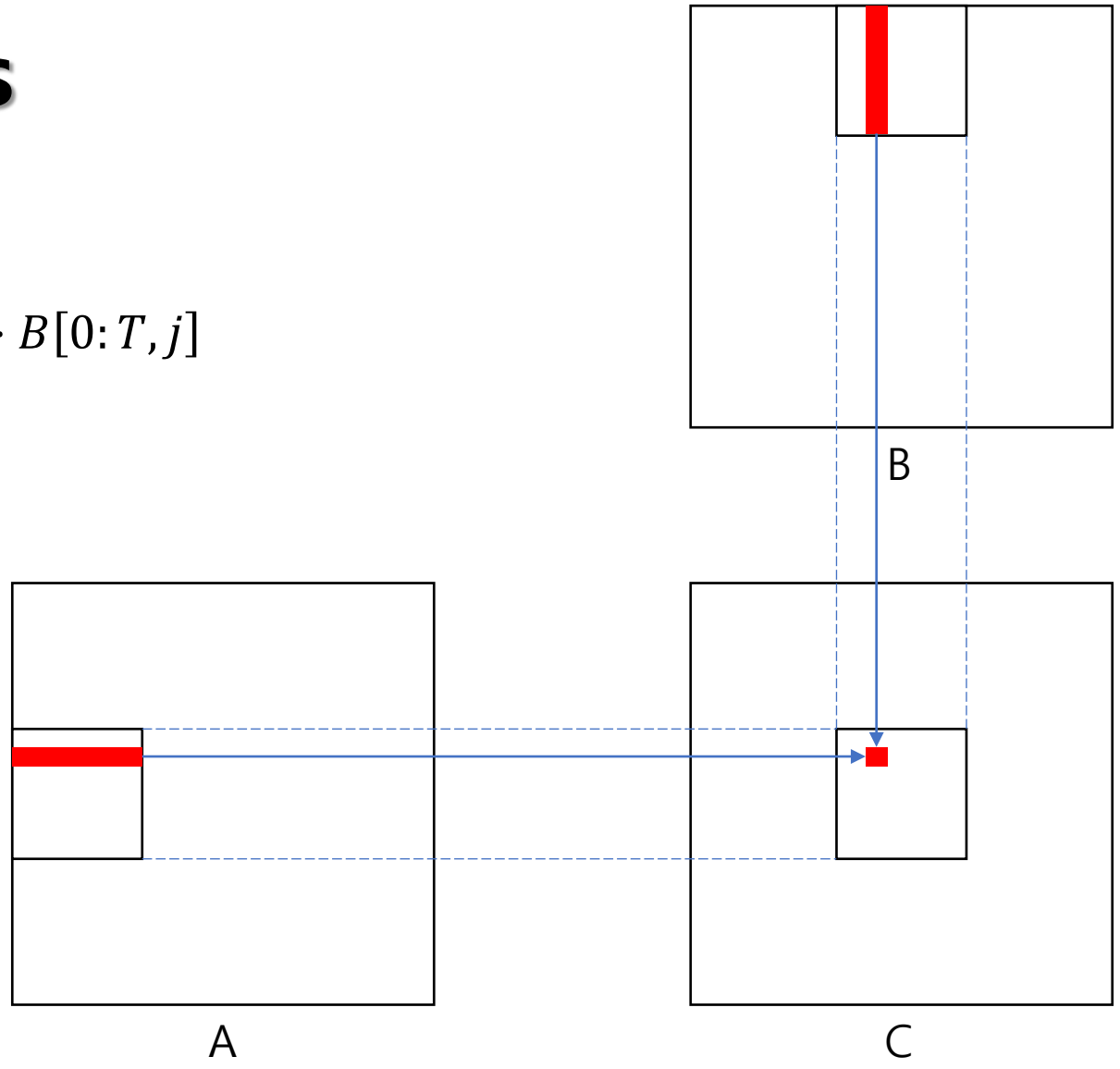
Matrix Tiling Details

- Let the tile size as $T\ by\ T$



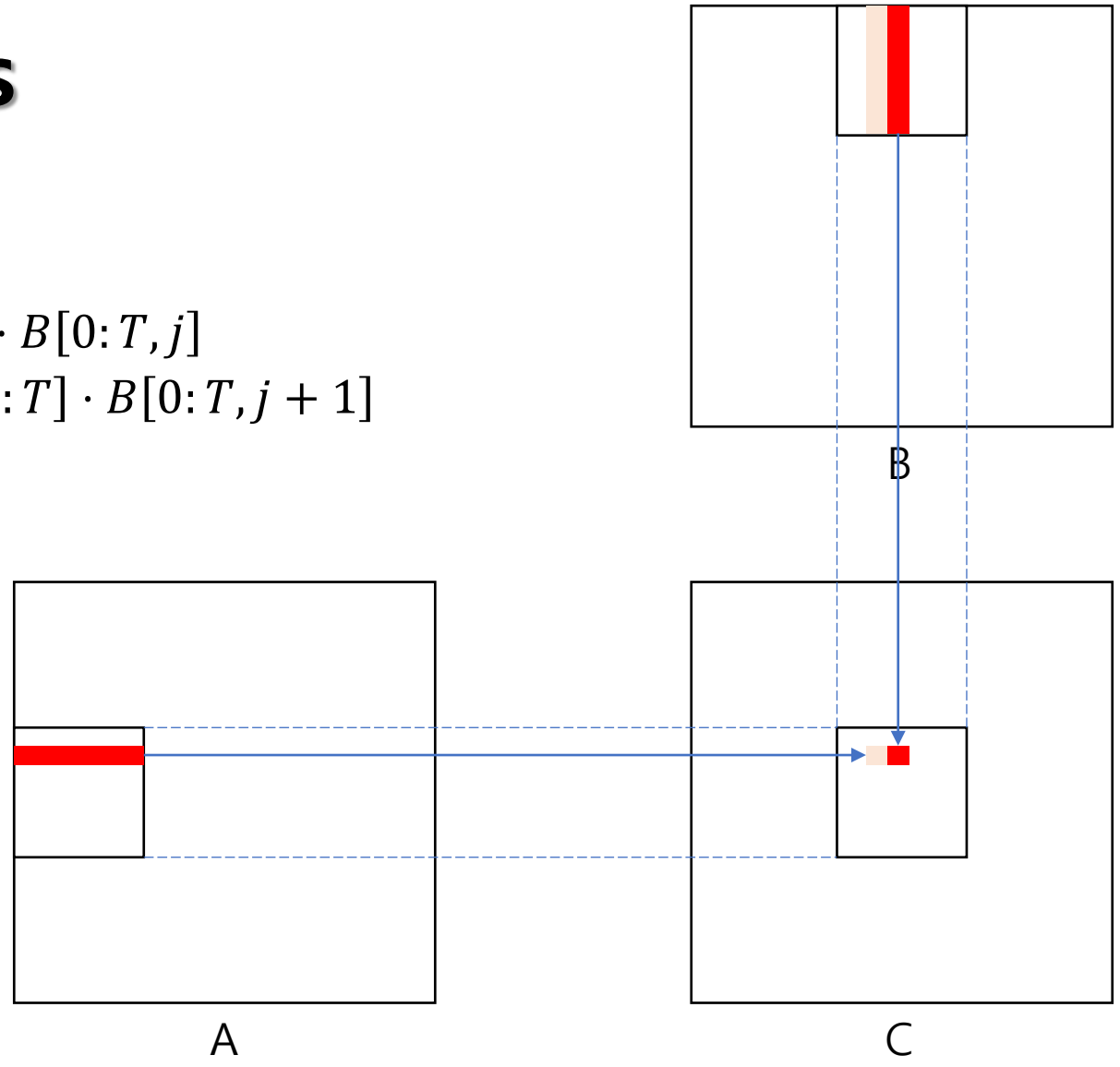
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$



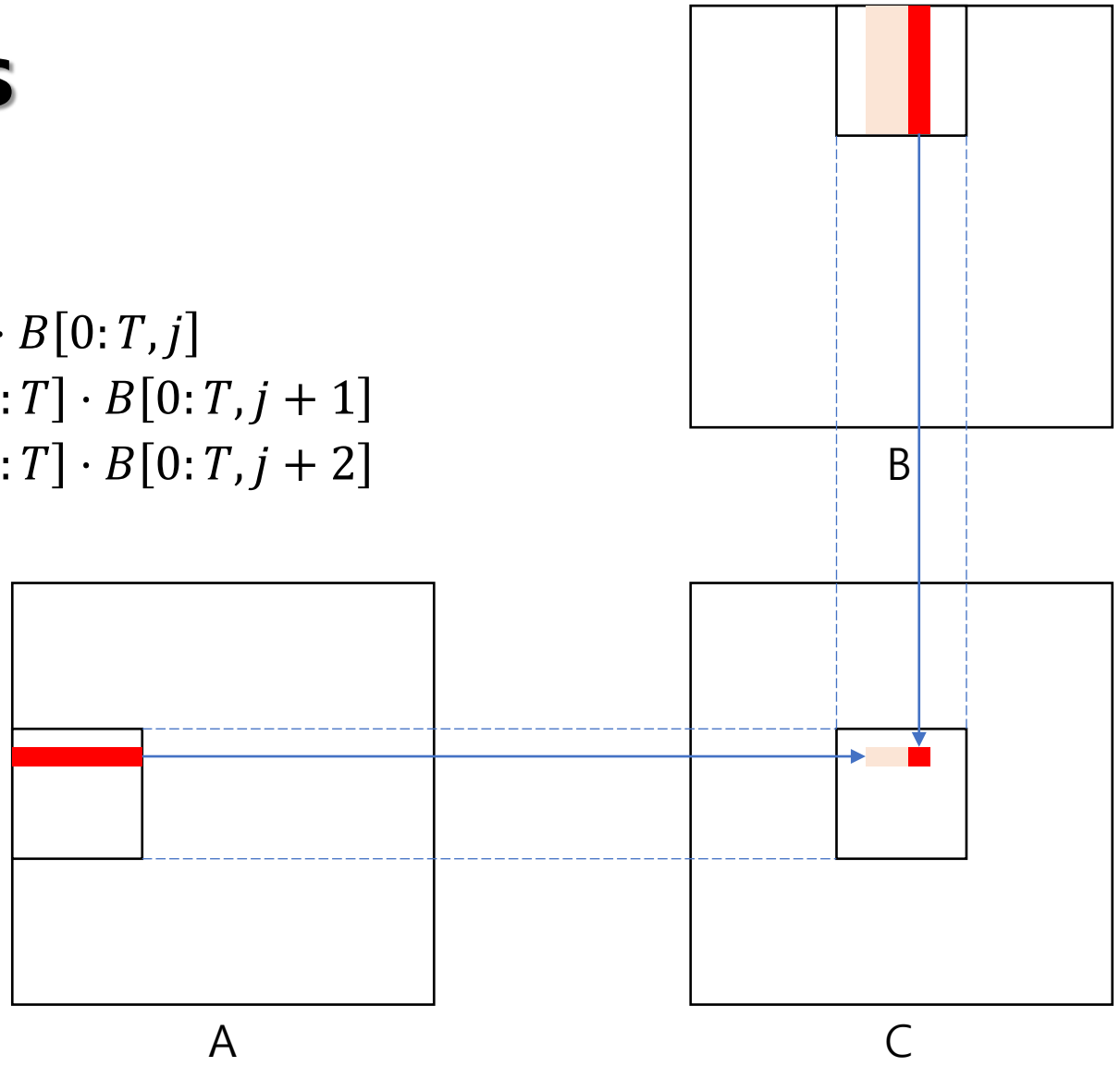
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$



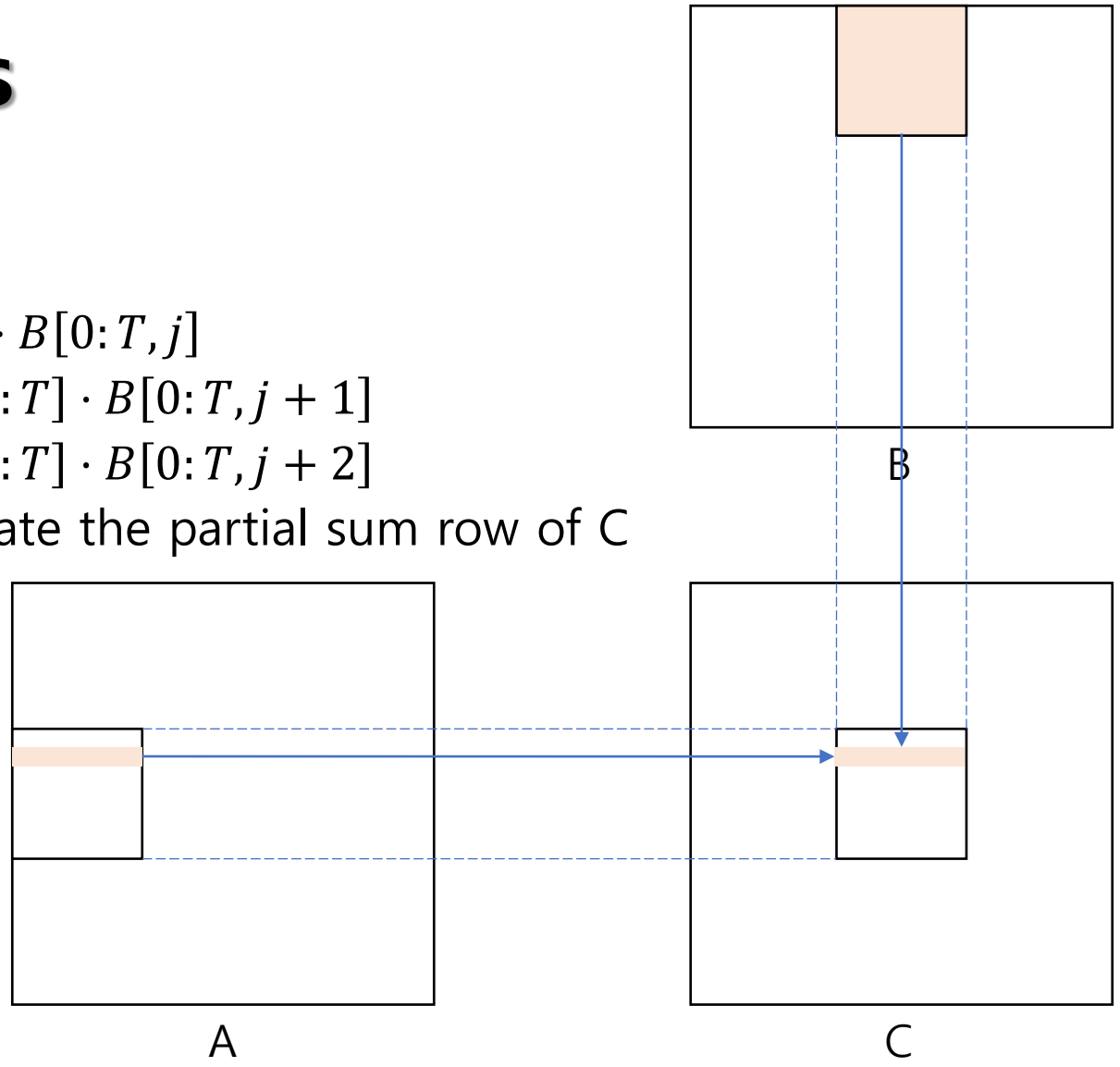
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$



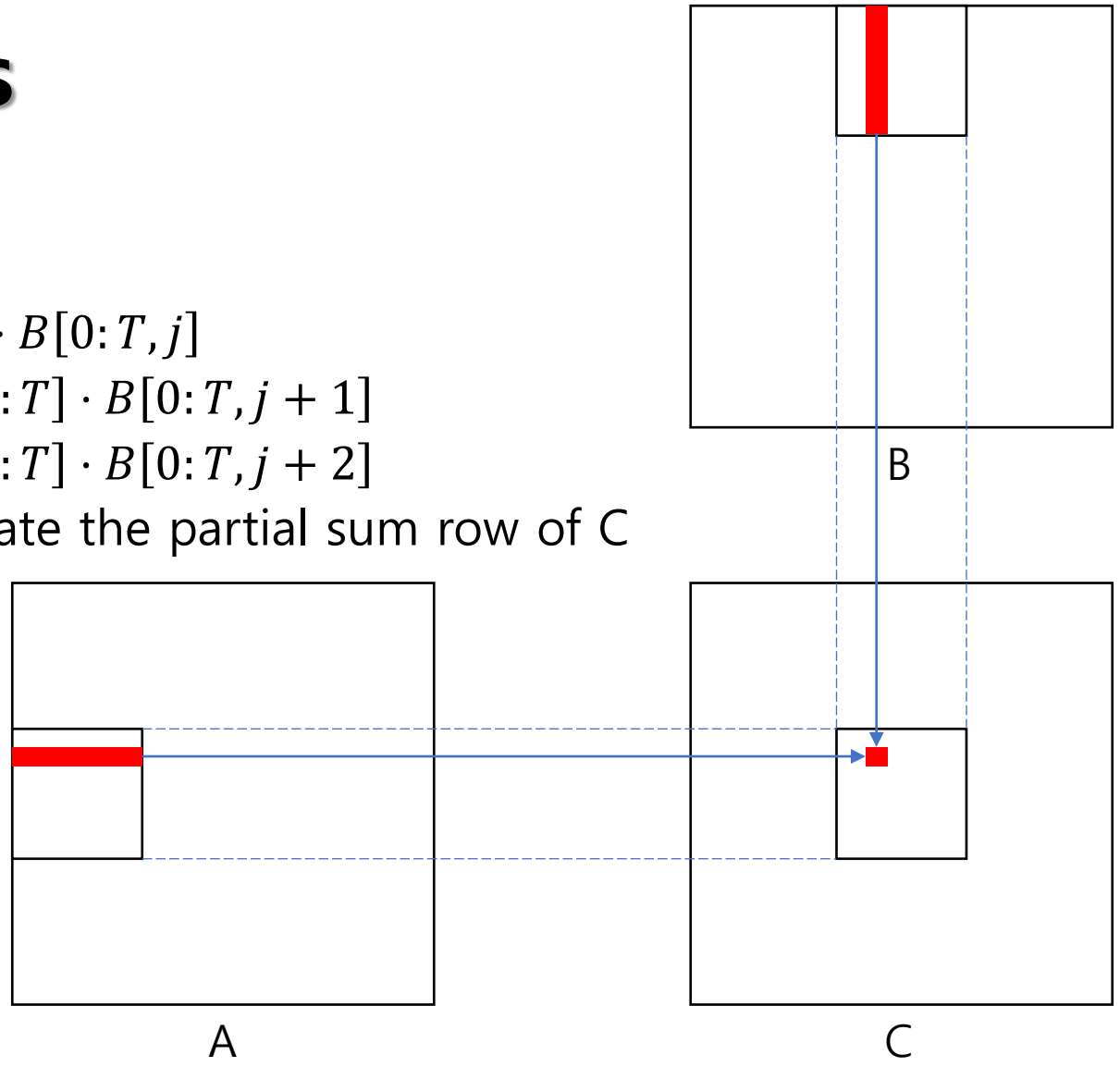
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C



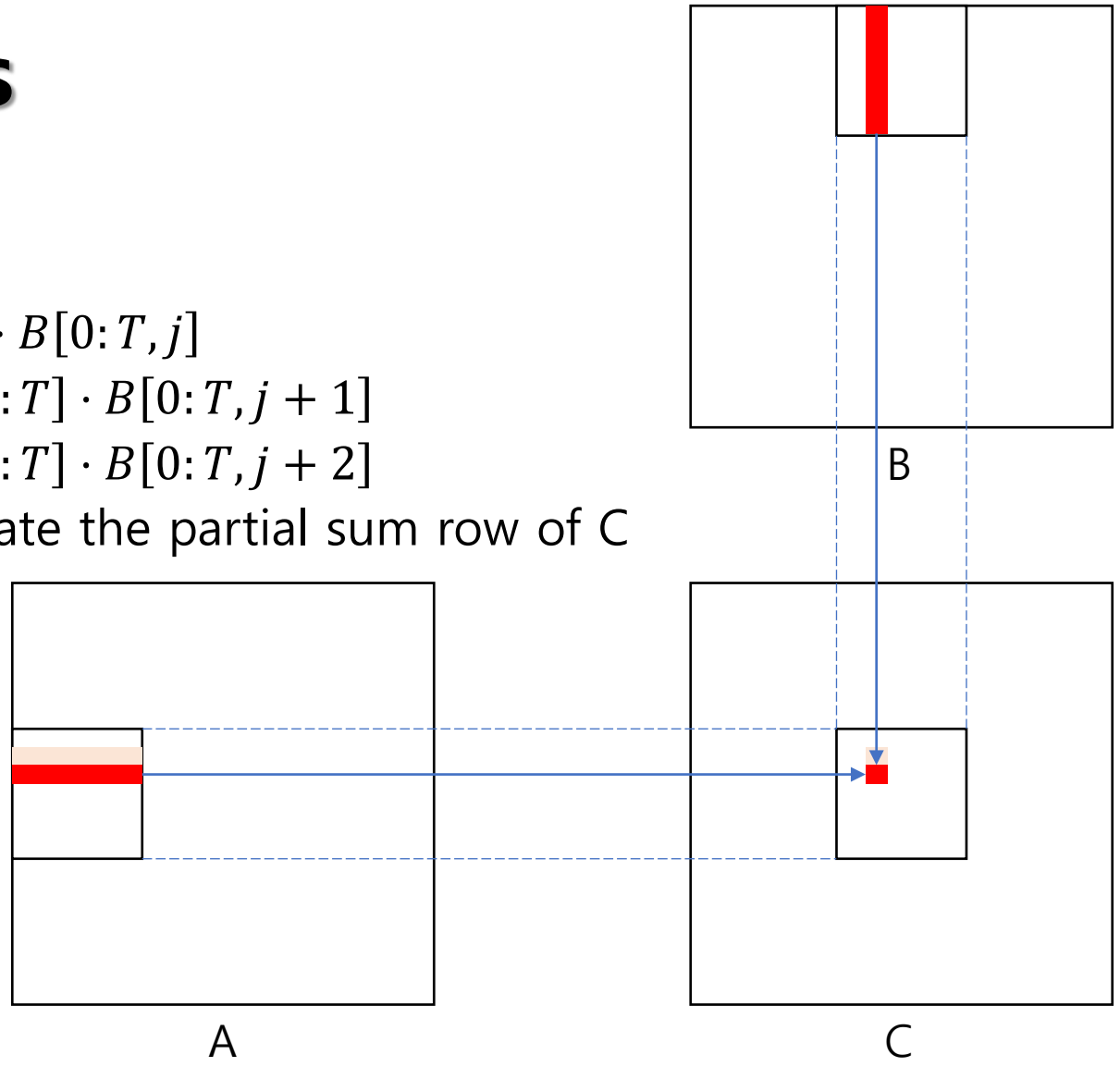
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C
  - Likewise, by sweeping A, we could generate the partial sum column of C



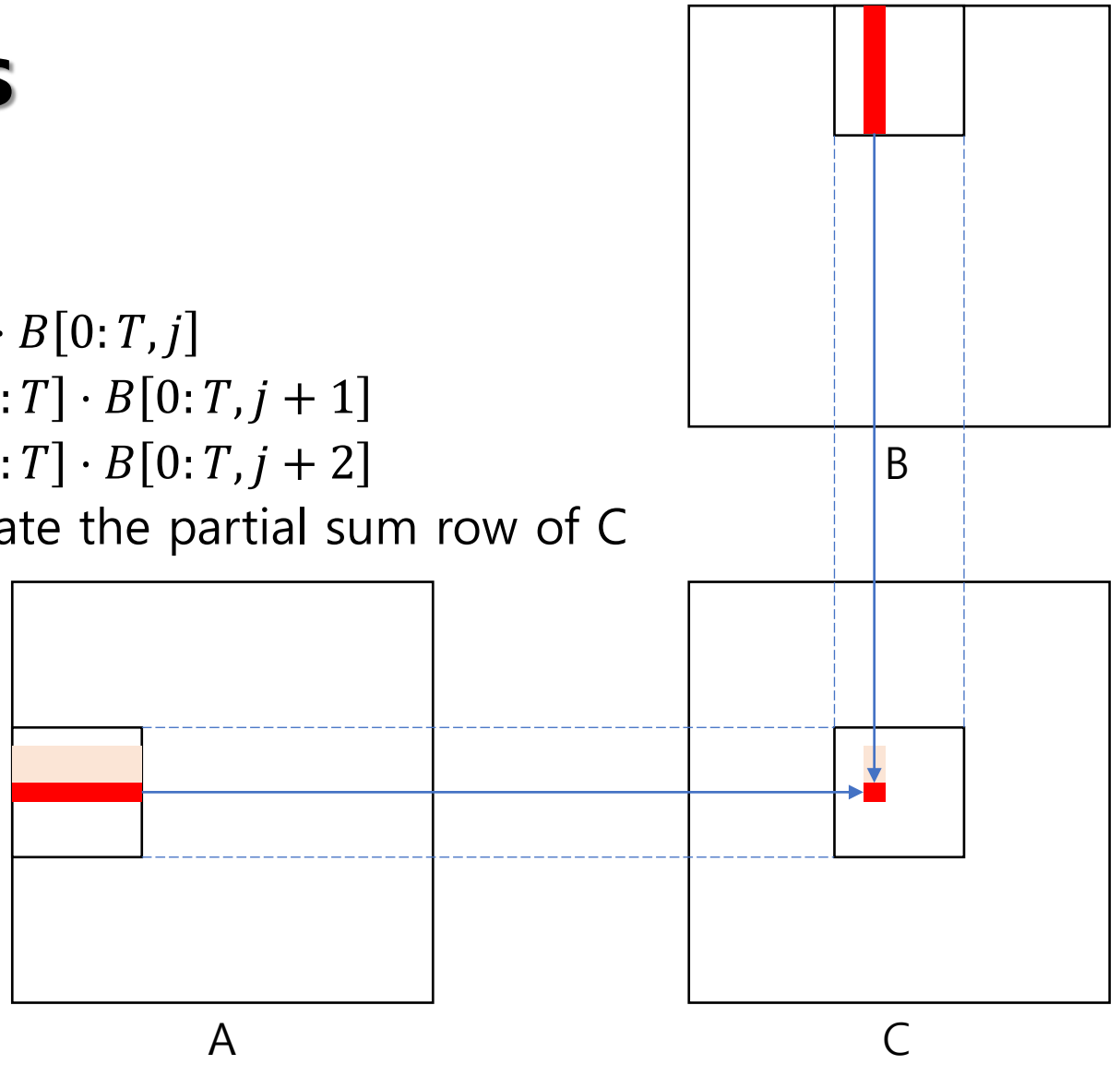
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C
  - Likewise, by sweeping A, we could generate the partial sum column of C



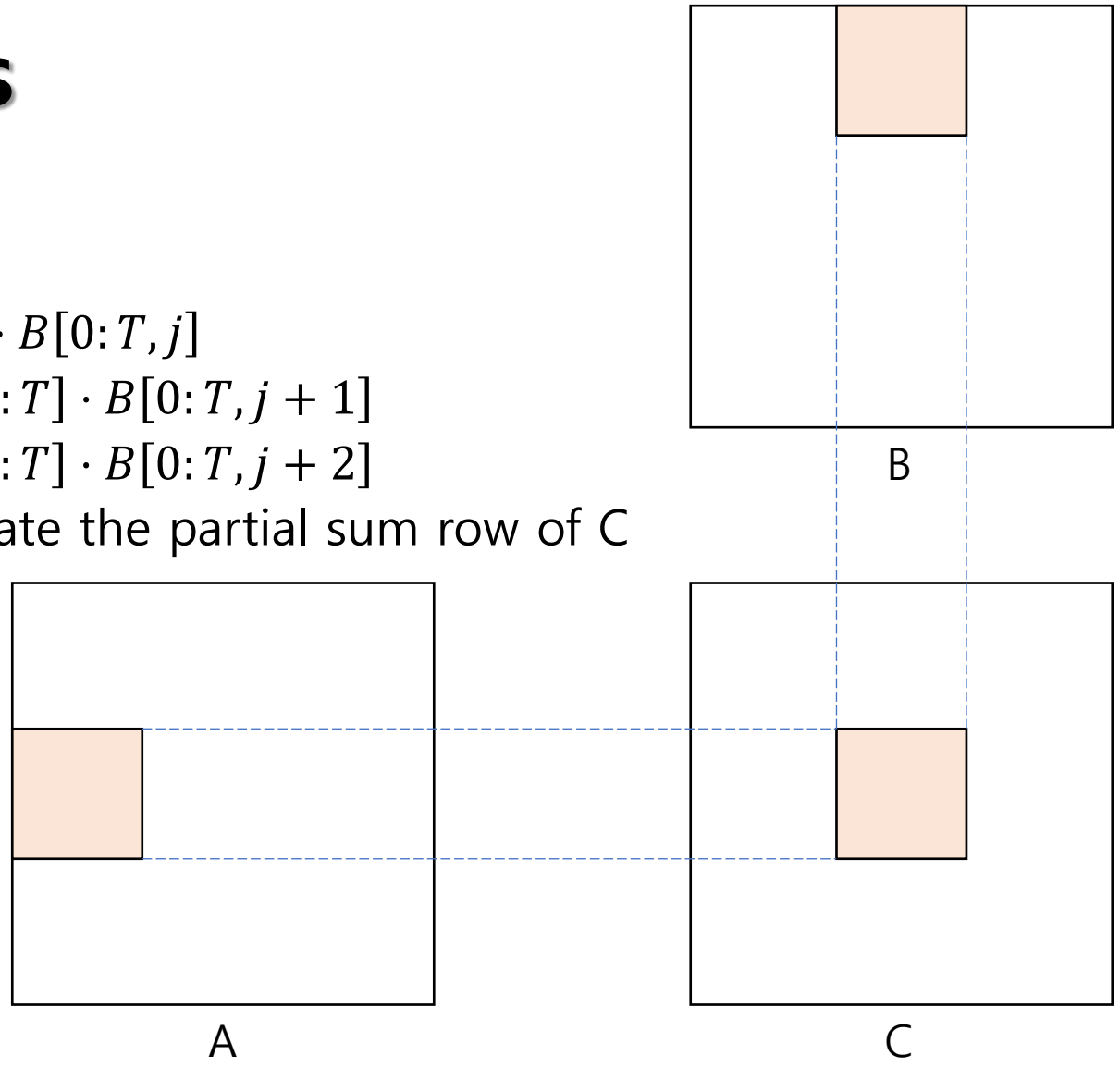
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C
  - Likewise, by sweeping A, we could generate the partial sum column of C



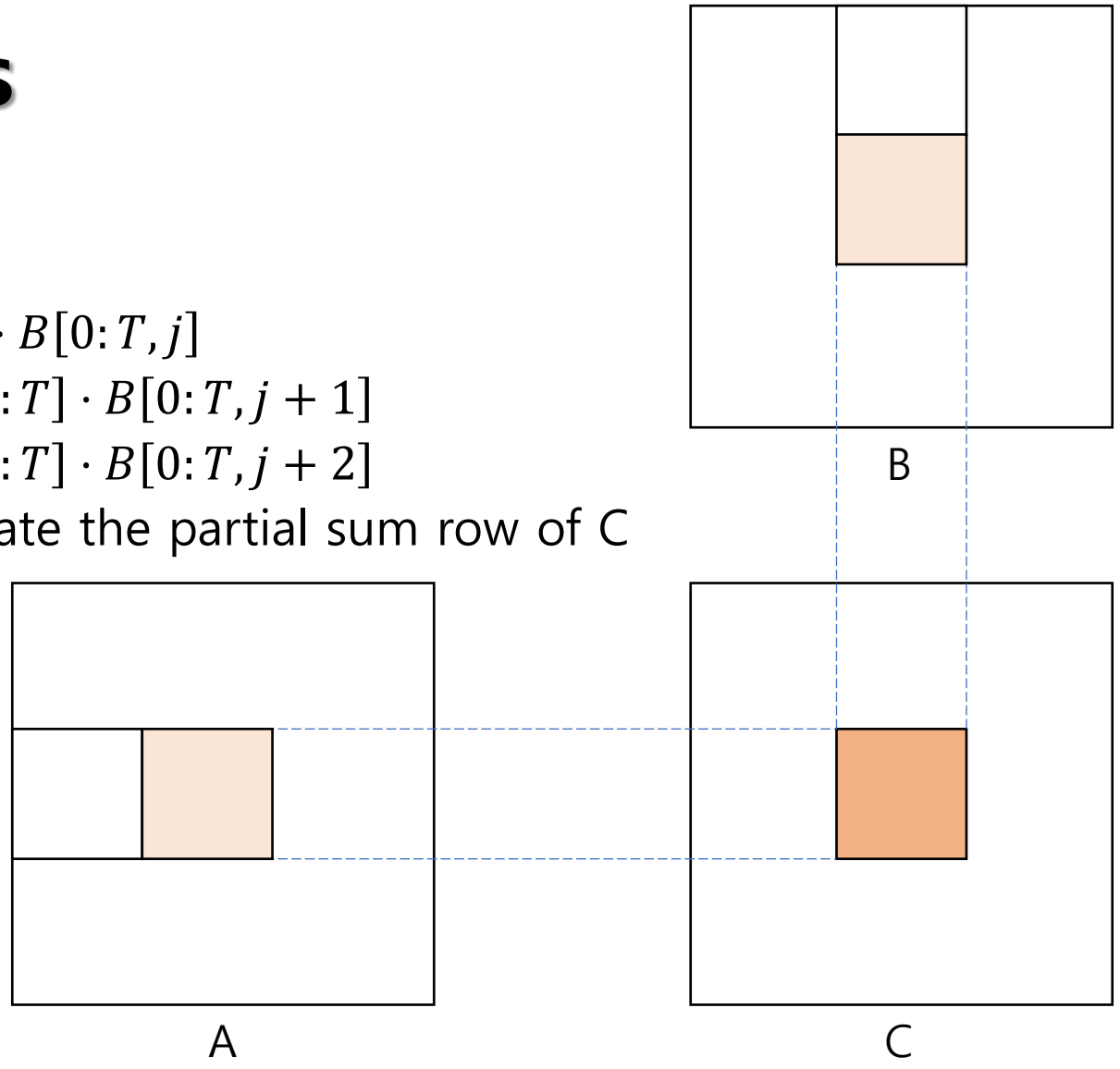
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C
  - Likewise, by sweeping A, we could generate the partial sum column of C
  - Data is reused T times



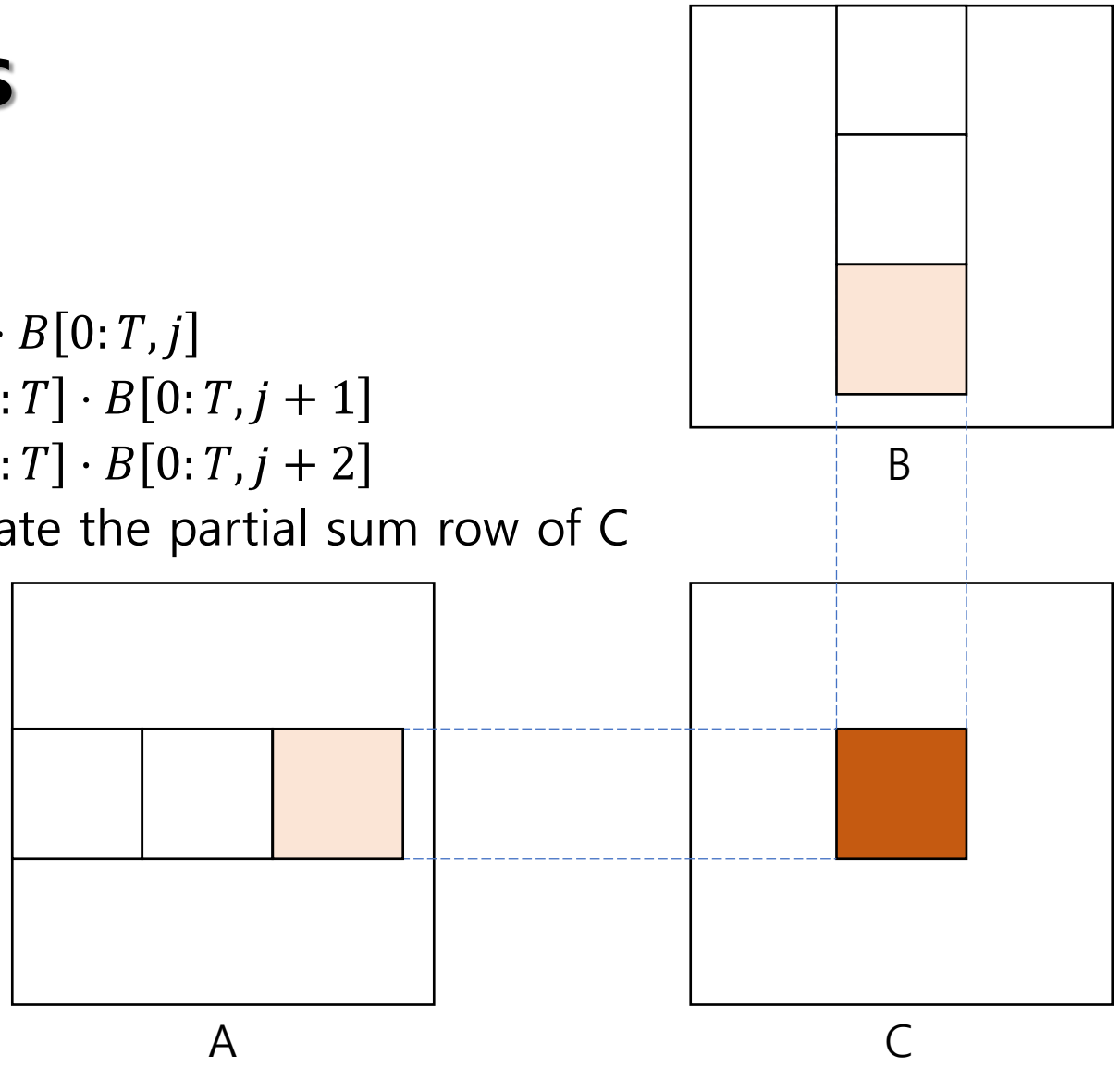
Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C
  - Likewise, by sweeping A, we could generate the partial sum column of C
  - Data is reused T times
  - By moving tile progressively, we could get total sum of C

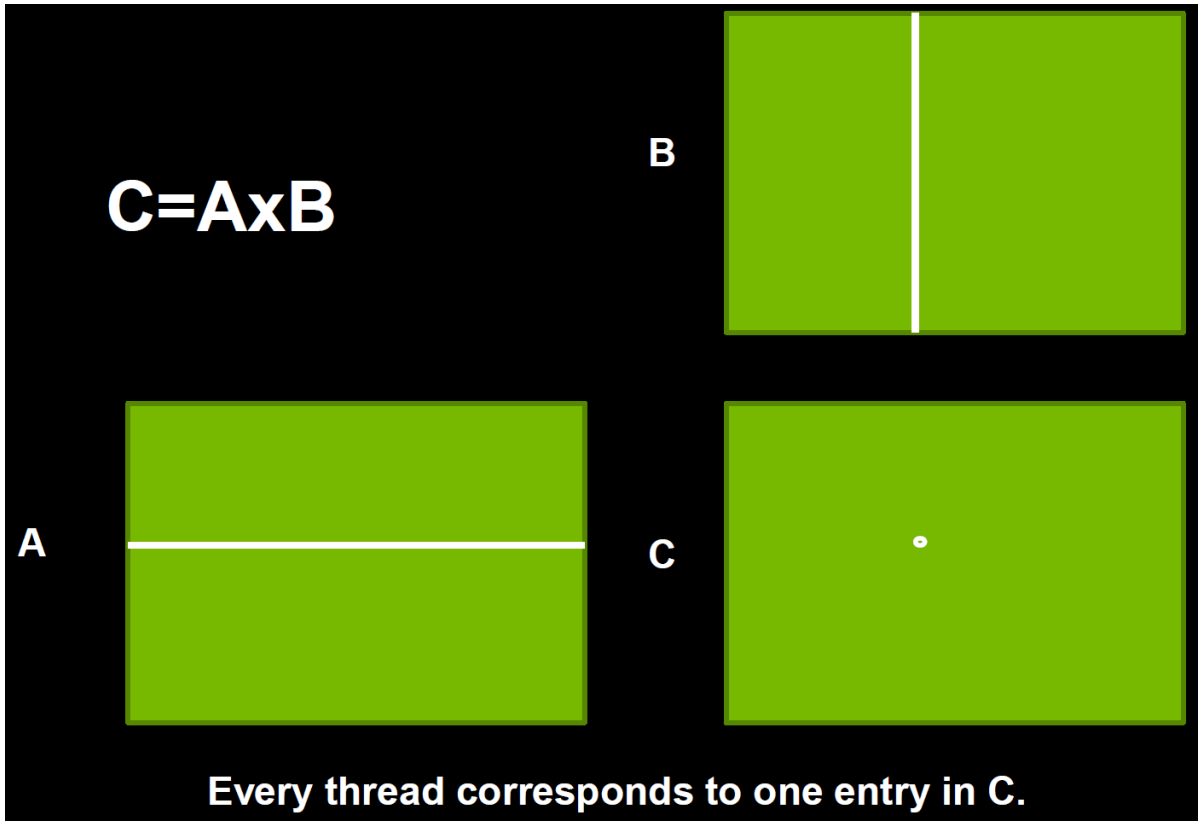


Matrix Tiling Details

- Let the tile size as $T\ by\ T$
  - Partial sum of $C[i,j] = A[i,0:T] \cdot B[0:T,j]$
  - Partial sum of $C[i,j+1] = A[i,0:T] \cdot B[0:T,j+1]$
  - Partial sum of $C[i,j+2] = A[i,0:T] \cdot B[0:T,j+2]$
  - By sweeping B, we could generate the partial sum row of C
  - Likewise, by sweeping A, we could generate the partial sum column of C
  - Data is reused T times
  - By moving tile progressively, we could get total sum of C



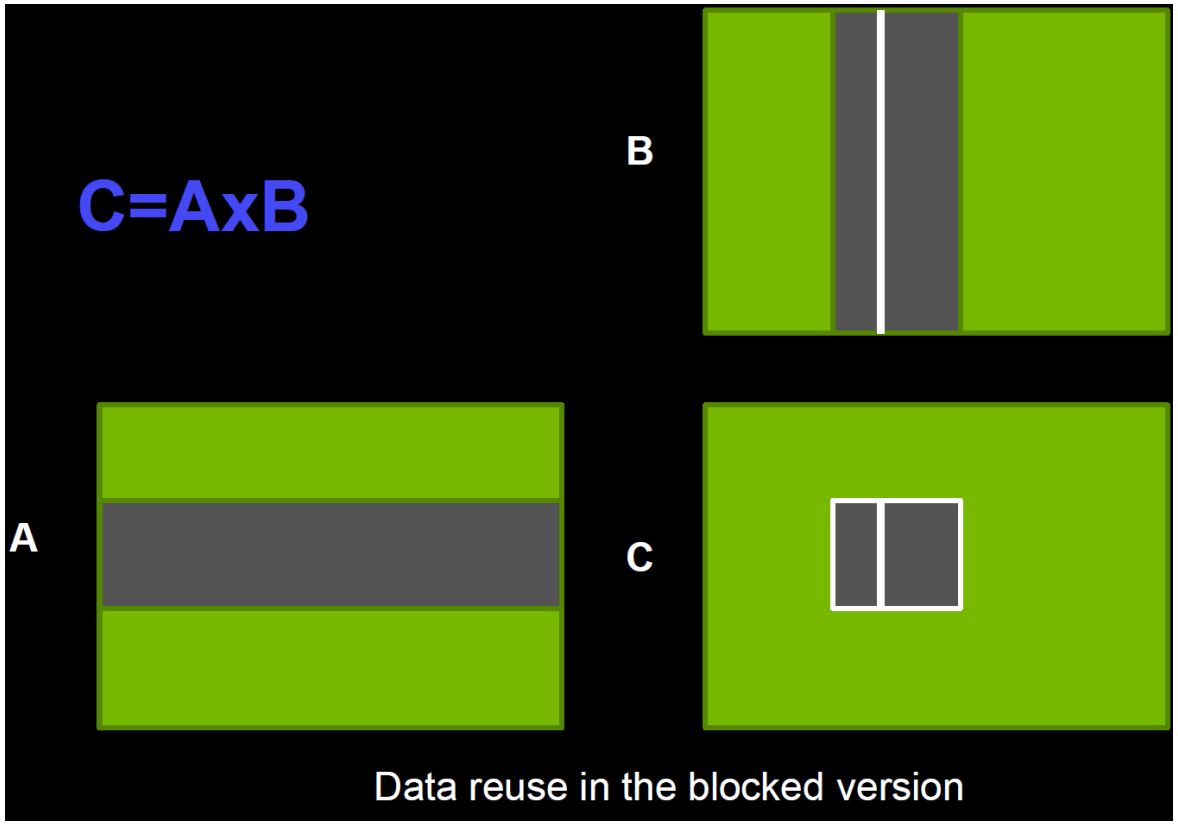
Example: Matrix Multiplication




```
__global__ void simpleMultiply(float* a,
                               float* b,
                               float* c,
                               int N)
{
  int row = threadIdx.x + blockIdx.x*blockDim.x;
  int col  = threadIdx.y + blockIdx.y*blockDim.y;
  float sum = 0.0f;
  for (int i = 0; i < N; i++) {
    sum += a[row*N+i] * b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

[NVIDIA]

Example: Matrix Multiplication




```
__global__ void coalescedMultiply(double*a,
                                  double* b,
                                  double*c,
                                  int N)
{
  __shared__ float aTile[TILE_DIM][TILE_DIM];
  __shared__ double bTile[TILE_DIM][TILE_DIM];

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for (int k = 0; k < N; k += TILE_DIM) {
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = k; i < k+TILE_DIM; i++)
      sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
  }
  c[row*N+col] = sum;
}
```

[NVIDIA]

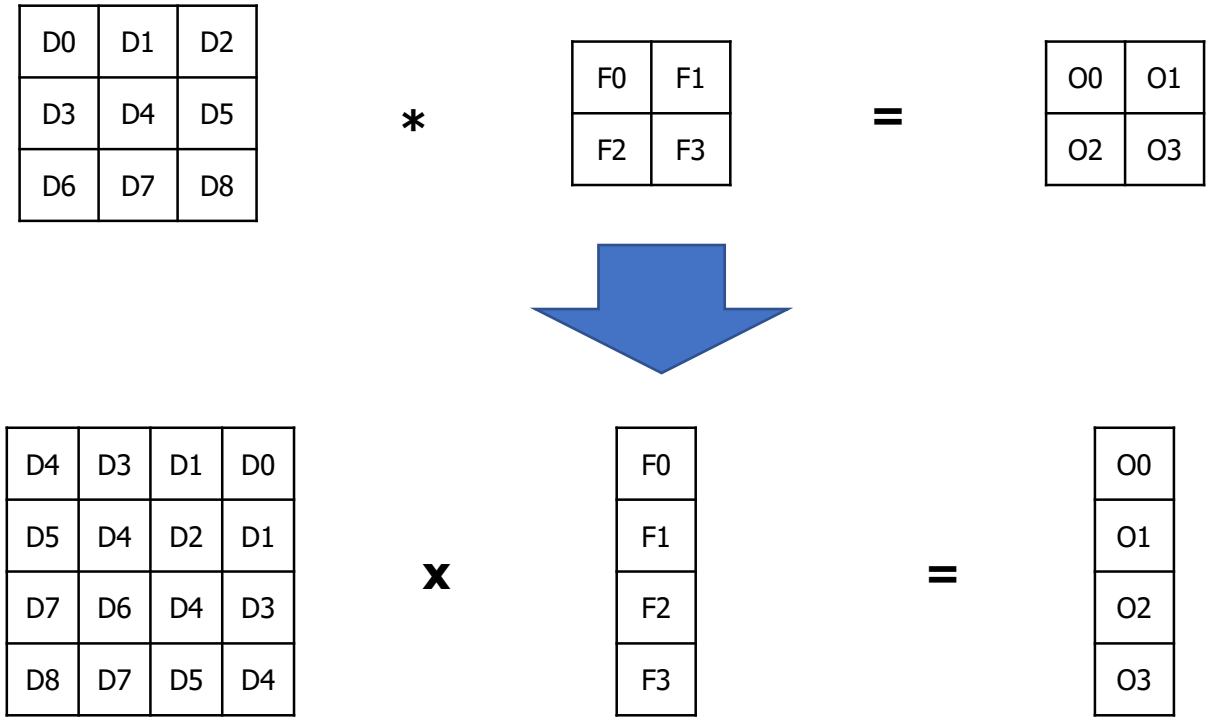
Example: Matrix Multiplication

M=N=K=512		
Optimization	C1060	C2050
A, B in global	12 Gflop/s	57 Gflop/s
A, B in shared	125 Gflop/s	181 Gflop/s

[NVIDIA]

Convolution in GPU

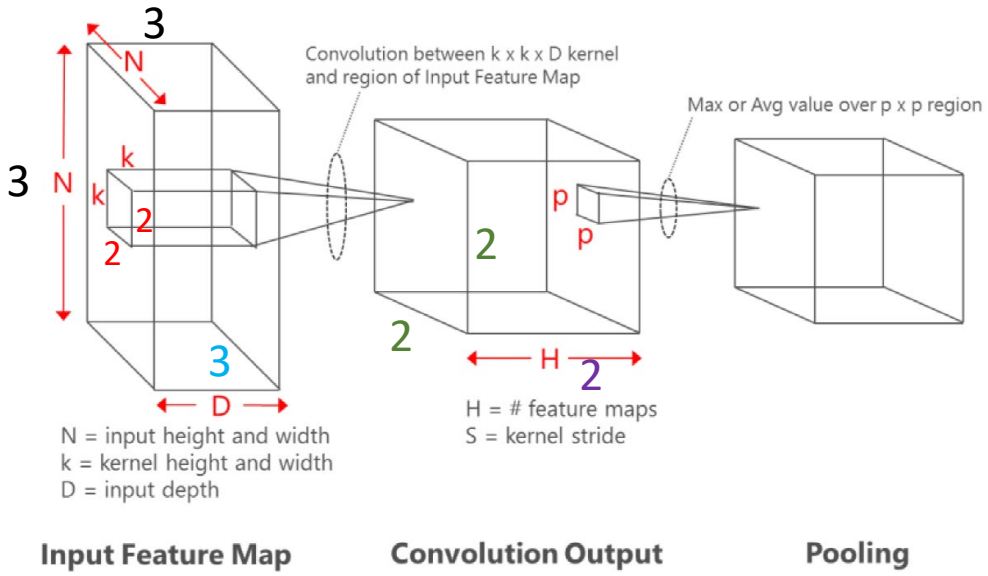
- Matrix multiplication with convolution lowering

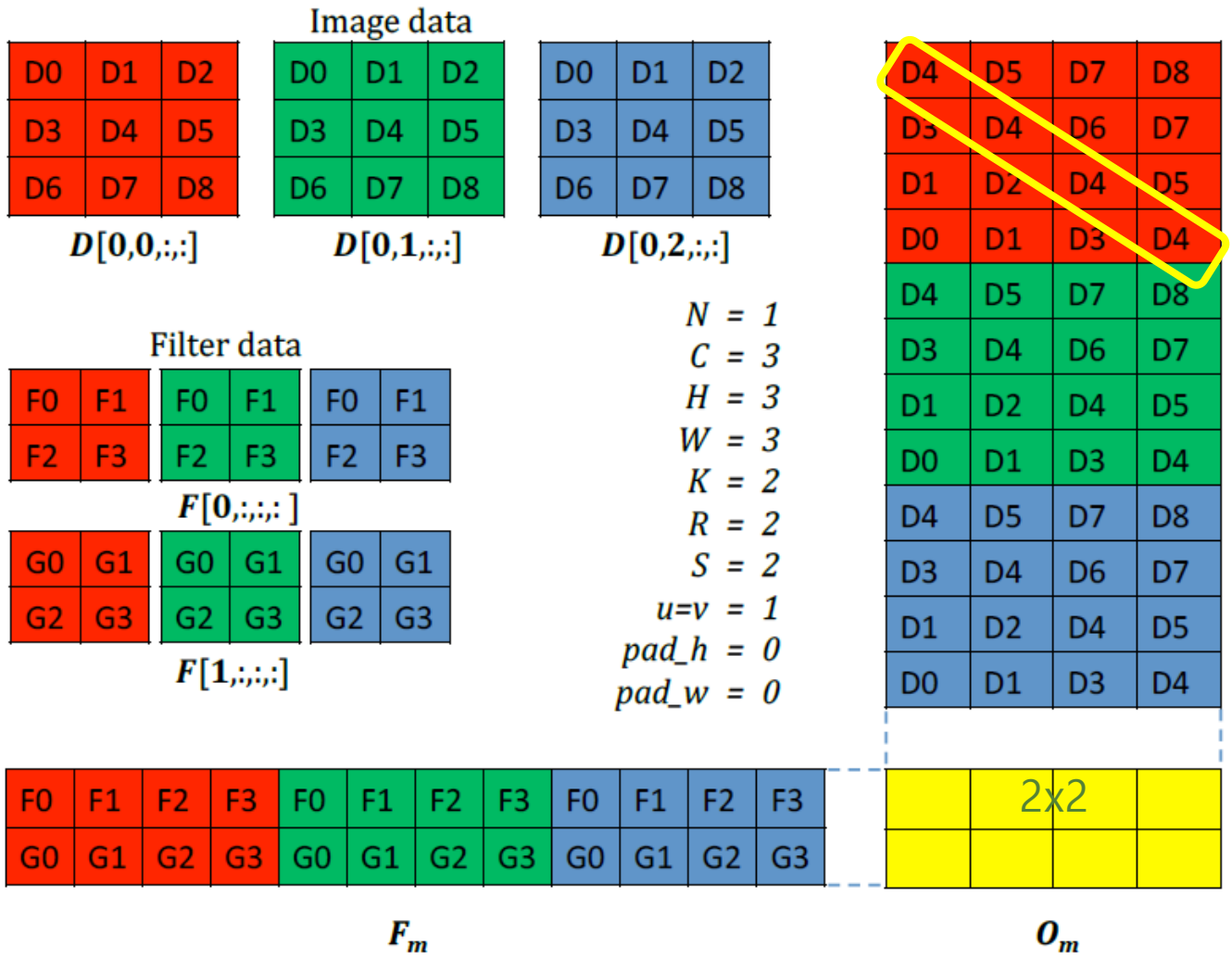


[Chetlur, 2014]
[S.Yoo]

Data Duplication Problem in Convolution Lowering

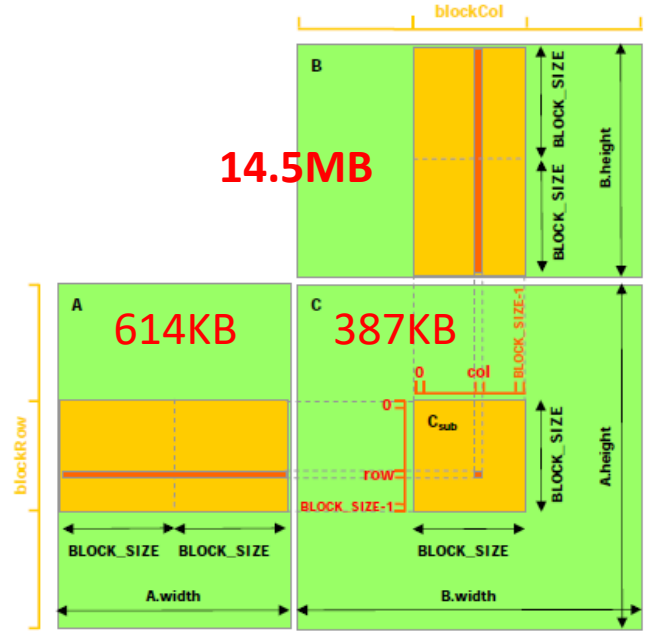
- Input: 3x3x3
- Output: 2x2x2
- Convolutional kernel: 3x2x2

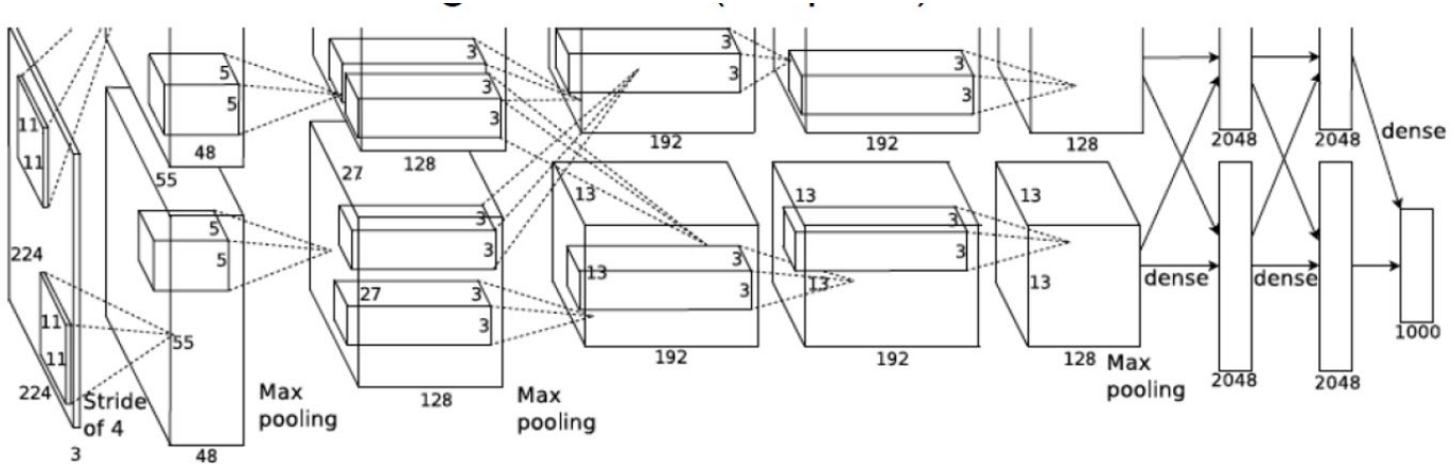




Matrix Size vs. GPU Cache Size

- Example: 2nd convolutional layer on AlexNet
- Input size = 55x55x48x4B = 580KB
  - Input matrix size = 580KBx5x5 = 14.5MB
- Output size = 27x27x128x4B = 387KB
- Kernel size = 48x5x5x128x4B = 614KB

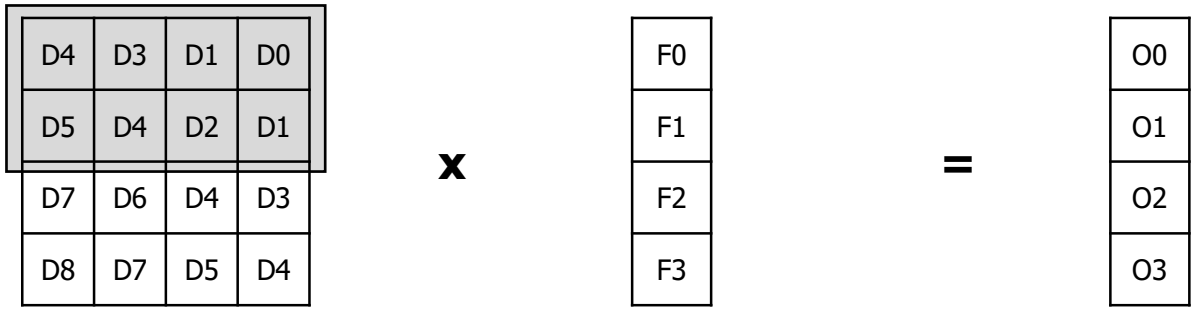


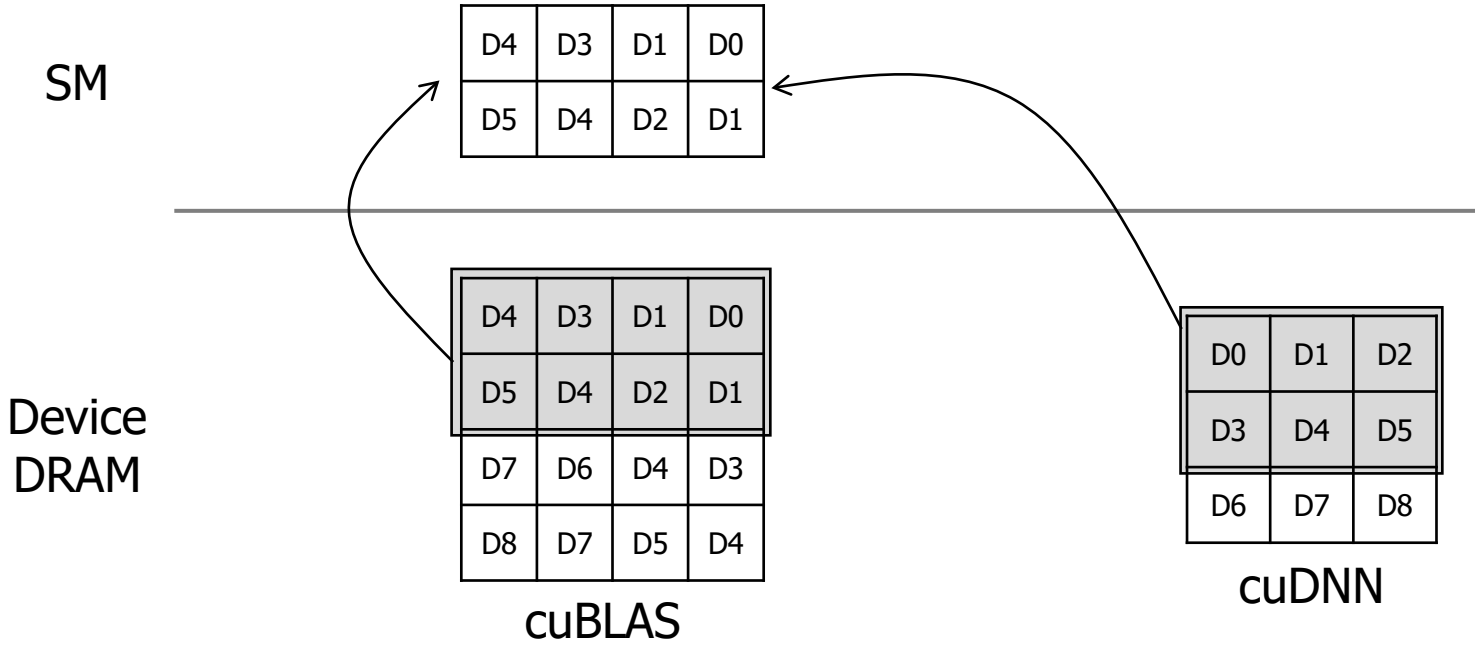


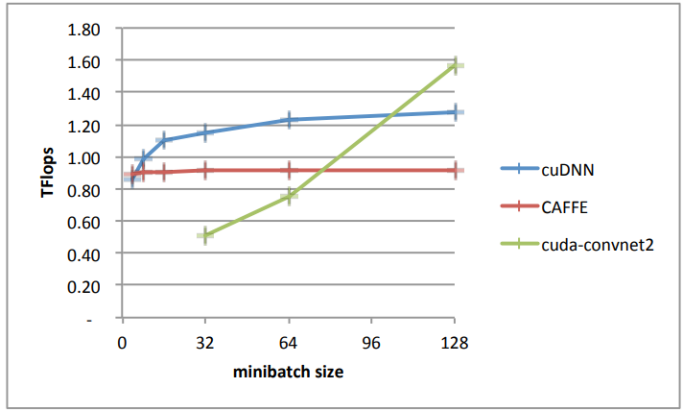
[S.Yoo]

cuBLAS vs. cuDNN

- In the case of cuDNN, it duplicates the data internally with indexing
  - Trade off between index computation and memory resource(capacity & bandwidth)







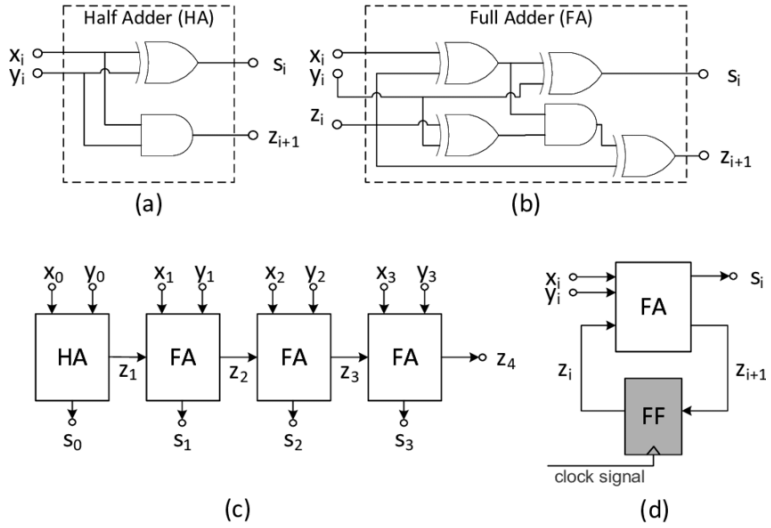
[cuDNN: Efficient Primitives for Deep Learning]

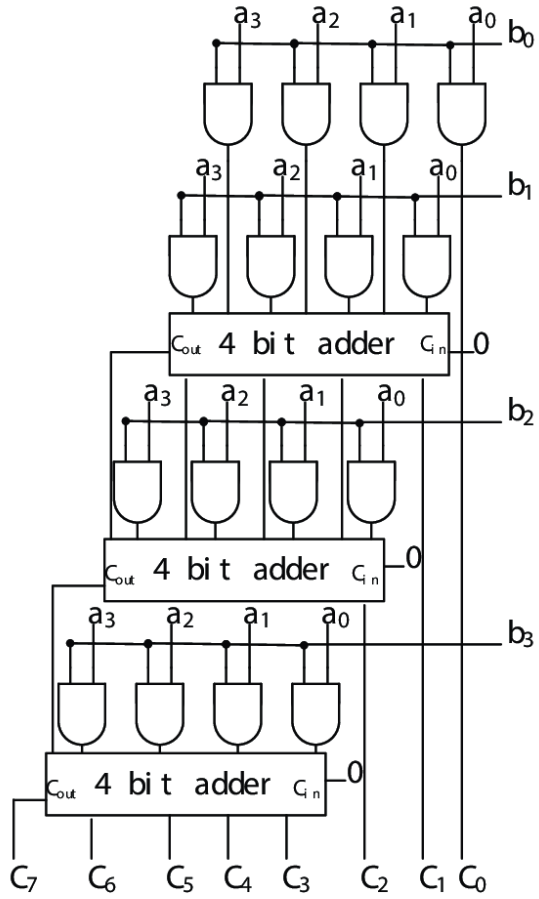
Winograd Convolution

Multiplication is Expensive

- Multiplication is more expensive than addition
  - Could we get the output in favor of additions instead of multiplications?

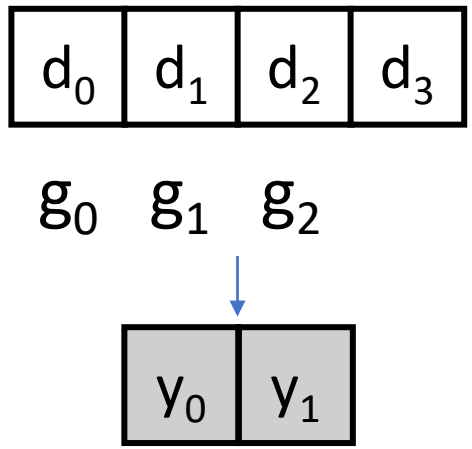
Operation	MUL	ADD
8bit Integer	0.2pJ	0.03pJ
32bit Integer	3.1pJ	0.1pJ
16bit Floating Point	1.1pJ	0.4pJ
32tbit Floating Point	3.7pJ	0.9pJ





Concept of Winograd Convolution - 1

- Reduce # multiplications at the cost of additional additions
- Example: F(2,3) ← 2 output, 3 kernel (4 input)



Concept of Winograd Convolution - 2

- Reduce # multiplications at the cost of additional additions
  - 2.26X faster than FFT for F(2x2, 3x3) [Lavin, 2015]

- Example: F(2,3)

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}$$

  - $y_0 = d_0 \cdot g_0 + d_1 \cdot g_1 + d_2 \cdot g_2$
  - $y_1 = d_1 \cdot g_0 + d_2 \cdot g_1 + d_3 \cdot g_2$
  - 6 multiplications + 4 additions

Concept of Winograd Convolution - 3

- Reduce # multiplications at the cost of additional additions
  - 2.26X faster than FFT for F(2x2, 3x3) [Lavin, 2015]

- Example: F(2,3)

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_0 + m_1 + m_2 \\ m_1 - m_2 - m_3 \end{bmatrix}$$

  - $m_0 = (d_0 - d_2) \cdot g_0$
  - $m_1 = (d_1 + d_2) \cdot \frac{g_0+g_1+g_2}{2}$
  - $m_2 = (d_2 - d_1) \cdot \frac{g_0-g_1+g_2}{2}$
  - $m_3 = (d_1 - d_3) \cdot g_2$

  - 4 multiplications + 8 additions (+ offline costs)

Concept of Winograd Convolution - 4

- Reduce # multiplications at the cost of additional additions
  - 2.26X faster than FFT for F(2x2, 3x3) [Lavin, 2015]

- Example: F(2,3)

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_0 + m_1 + m_2 \\ m_1 - m_2 - m_3 \end{bmatrix}$$

  - Matrix form

$$Y = A^{\mathrm{T}}[U \odot V] = A^{\mathrm{T}}[(Gg) \odot (B^{\mathrm{T}}d)]$$

$$A^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$B^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}$$

Winograd Convolution @ 2-D

- We can nest 1-D algorithm to form 2-D algorithm
  - and F(2x2, 3x3) ← 2x2 output, 3x2 kernel (4x4 input)

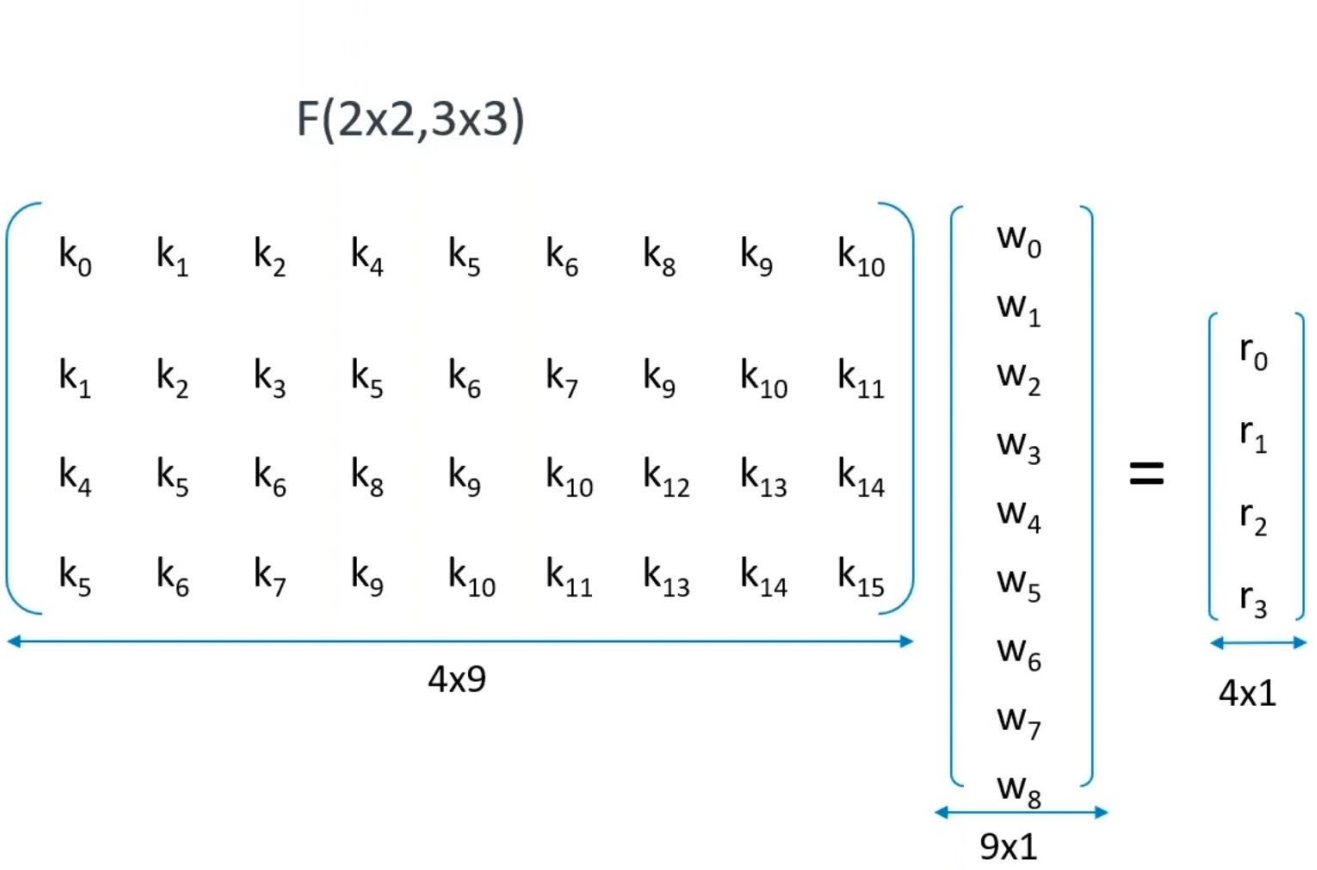
1D $$Y = A^{\mathrm{T}}[U \odot V] = A^{\mathrm{T}}[(Gg) \odot (B^{\mathrm{T}}d)]$$

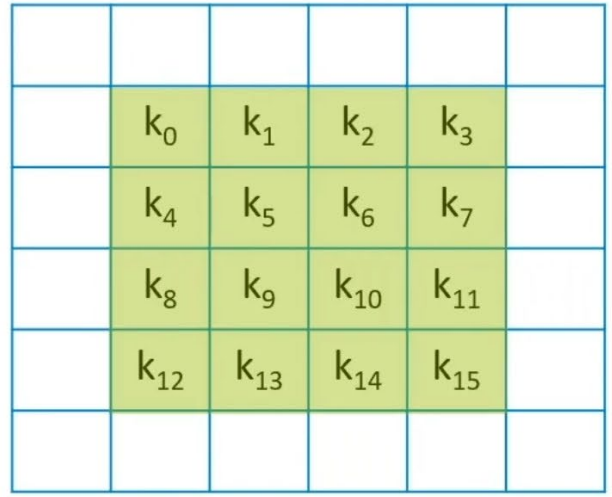
2D $$Y = A^{\mathrm{T}}[(GgG^{\mathrm{T}}) \odot (B^{\mathrm{T}}dB)]A$$

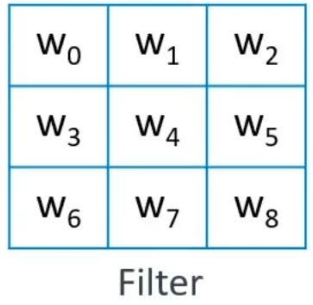
$$A^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$B^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}$$

Winograd Convolution @ 2-D

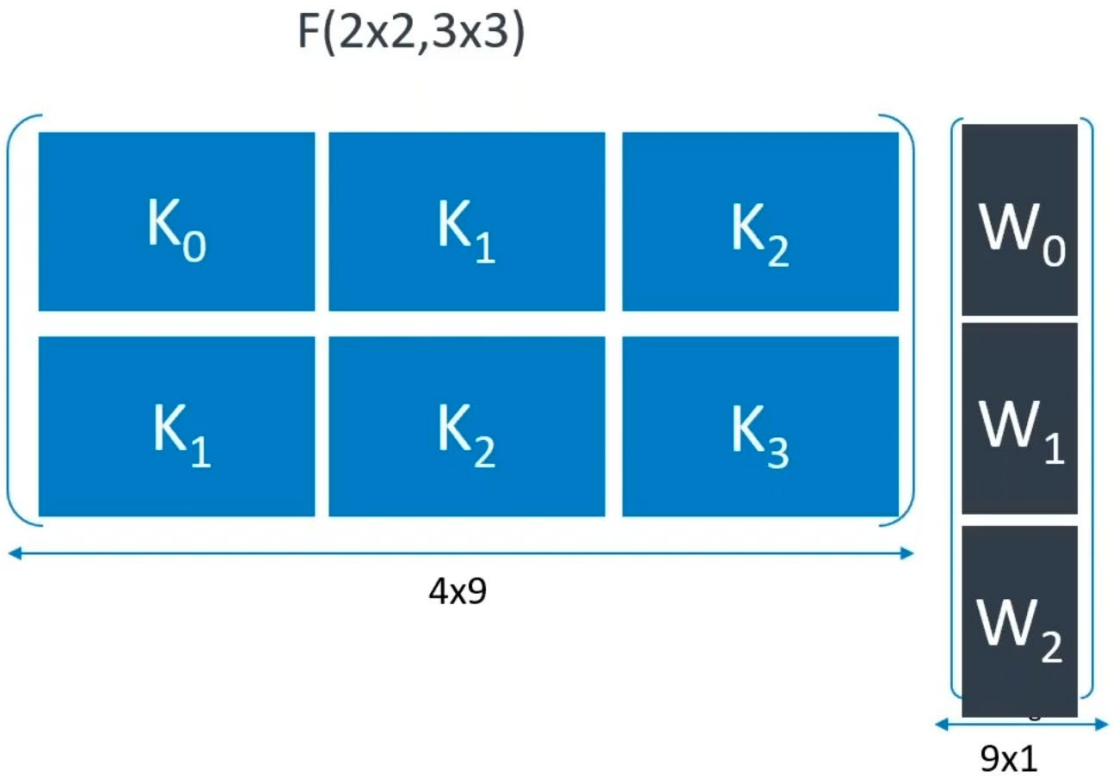






[Gian, 2018, AMD]

Winograd Convolution @ 2-D



$$\begin{bmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix}$$

[Gian, 2018, AMD]

Winograd Convolution @ 2-D

$$\begin{pmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{pmatrix} \begin{pmatrix} W_0 \\ W_1 \\ W_2 \end{pmatrix} = \begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = \begin{pmatrix} M_0 + M_1 + M_2 \\ M_1 - M_2 - M_3 \end{pmatrix}$$

Matrix multiply F(2,3)! **4** multiplications

$$M_0 = (K_0 - K_2) \cdot W_0 \qquad M_1 = (K_1 + K_2) \cdot \frac{W_0 + W_1 + W_2}{2}$$

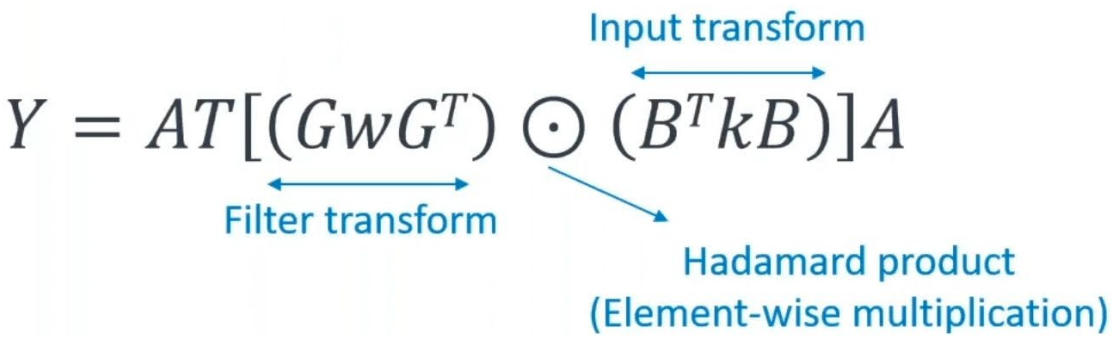
$$M_3 = (K_1 - K_3) \cdot W_2 \qquad M_2 = (K_2 - K_1) \cdot \frac{W_0 - W_1 + W_2}{2}$$

16 multiplications instead of **36** of direct convolution

2.25 multiplication complexity reduction!

[Gian, 2018, AMD]

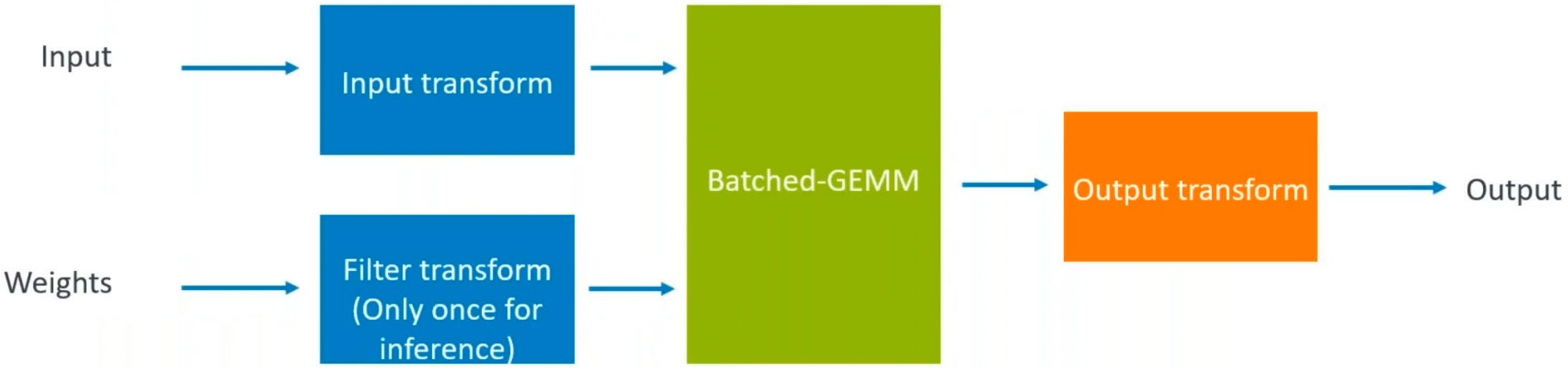
Winograd Convolution @ 2-D



- Input transform
- Filter transform (if the weights are constants, only once)
- Hadamard product (Element-wise multiplication)
- Output transform

[Gian, 2018, AMD]

Winograd Convolution @ 2-D

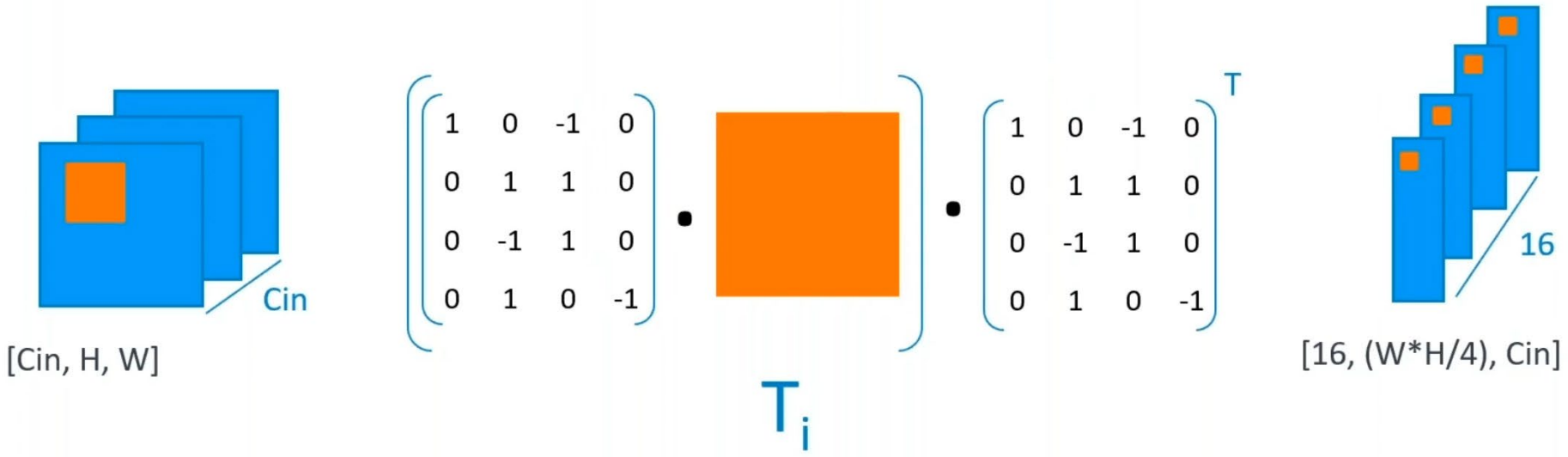


[Gian, 2018, AMD]

Winograd Convolution @ 2-D

For each 4x4 input tile (50 % overlapped)

1. Compute T_i
2. Store the transformed values across the 16 channels (4x4 - > 1x1x16)
3. Memory footprint increases of 4x

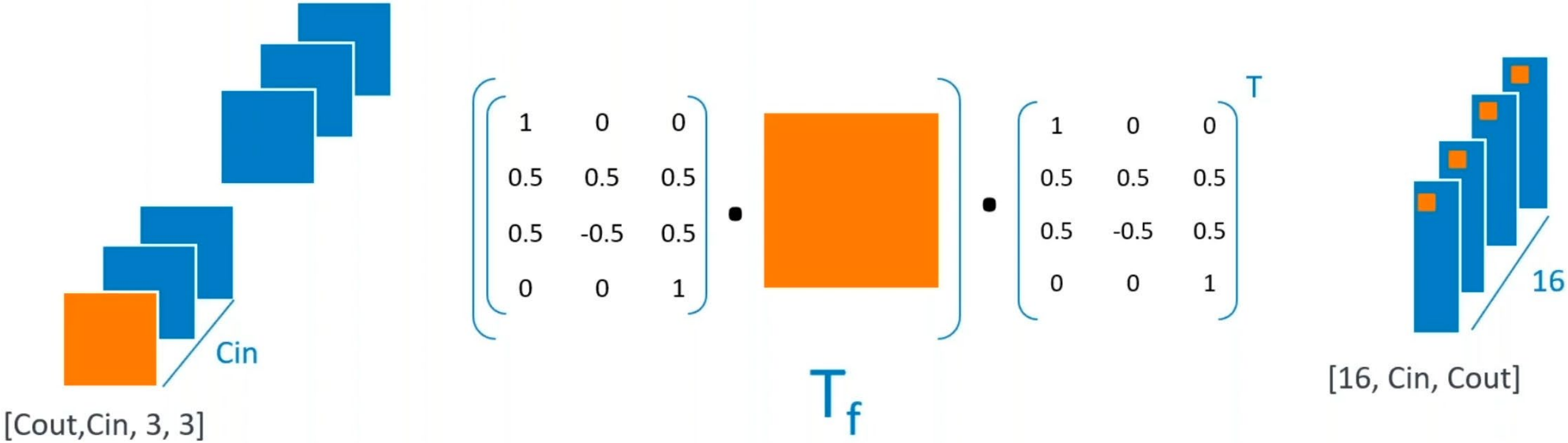


[Gian, 2018, AMD]

Winograd Convolution @ 2-D

Similar to the input transform. Extract each 3x3 filter plane

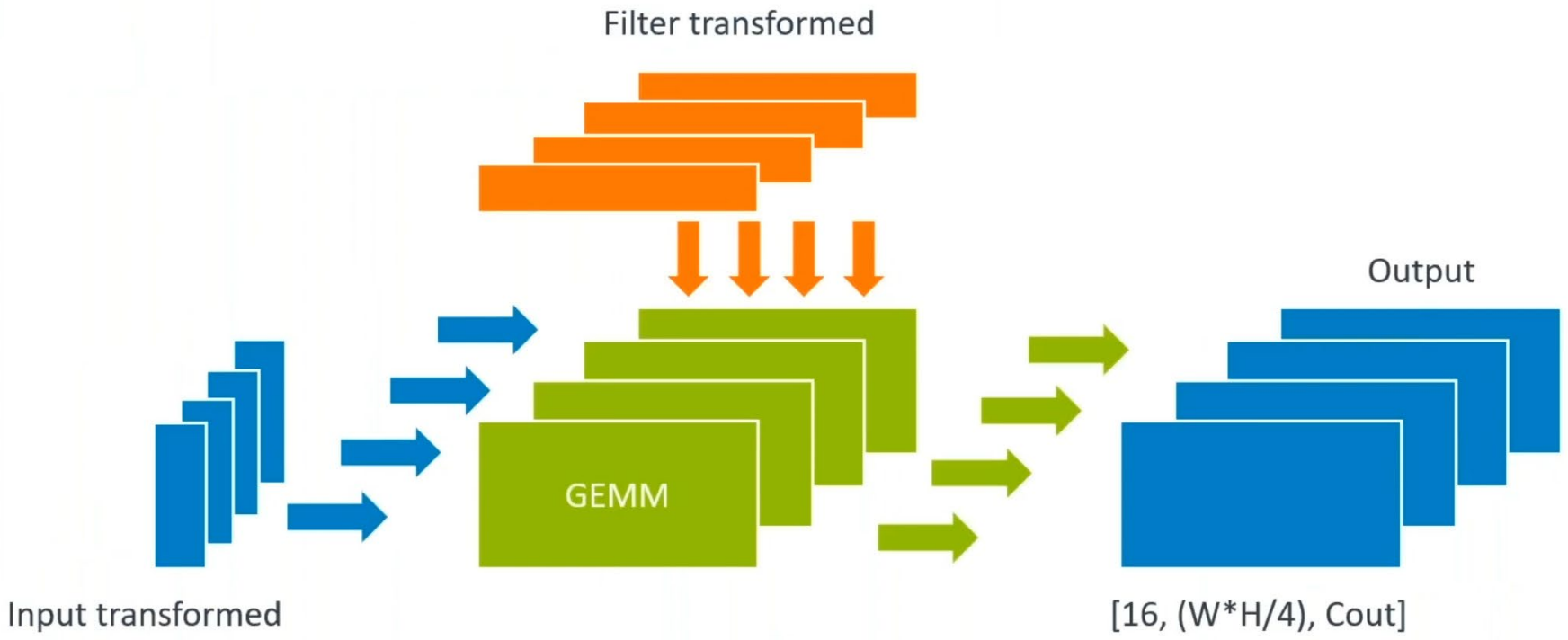
1. Compute T_f
2. Store the transformed values across the 16 channels (4x4 - > 1x1x16)
3. Memory footprint increases of 1.7x



[Gian, 2018, AMD]

Winograd Convolution @ 2-D

Element-wise multiplication can be reduced to batched-GEMM (16 GEMMs)

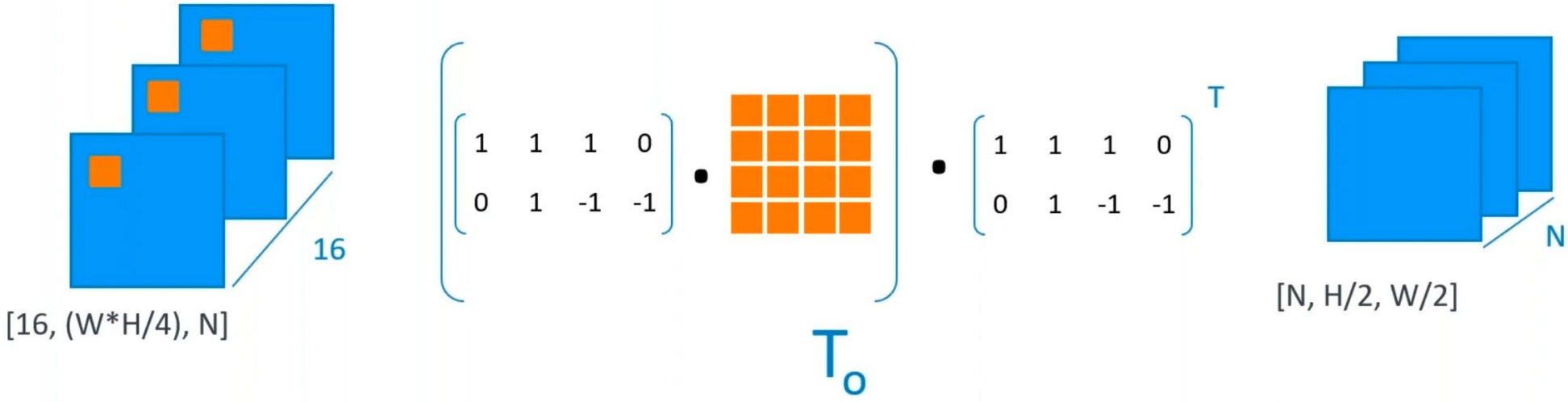


[Gian, 2018, AMD]

Winograd Convolution @ 2-D

The output transform:

1. Combine results across channels to form a 4x4 tile
2. Compute T_o
3. Store the 2x2 output tile in the space domain



[Gian, 2018, AMD]

Winograd Convolution @ 2-D

Assuming 3x3 kernels, unit pads and unit strides

- W: Width image
- H: Height image
- Cin: Input channels
- Cout: Output channels

	Im2col	Input Transform	Input transform / im2col
Size (elements)	W * H * 3 * 3 * Cin	16 * (W/2) * (H/2) * Cin	0.44

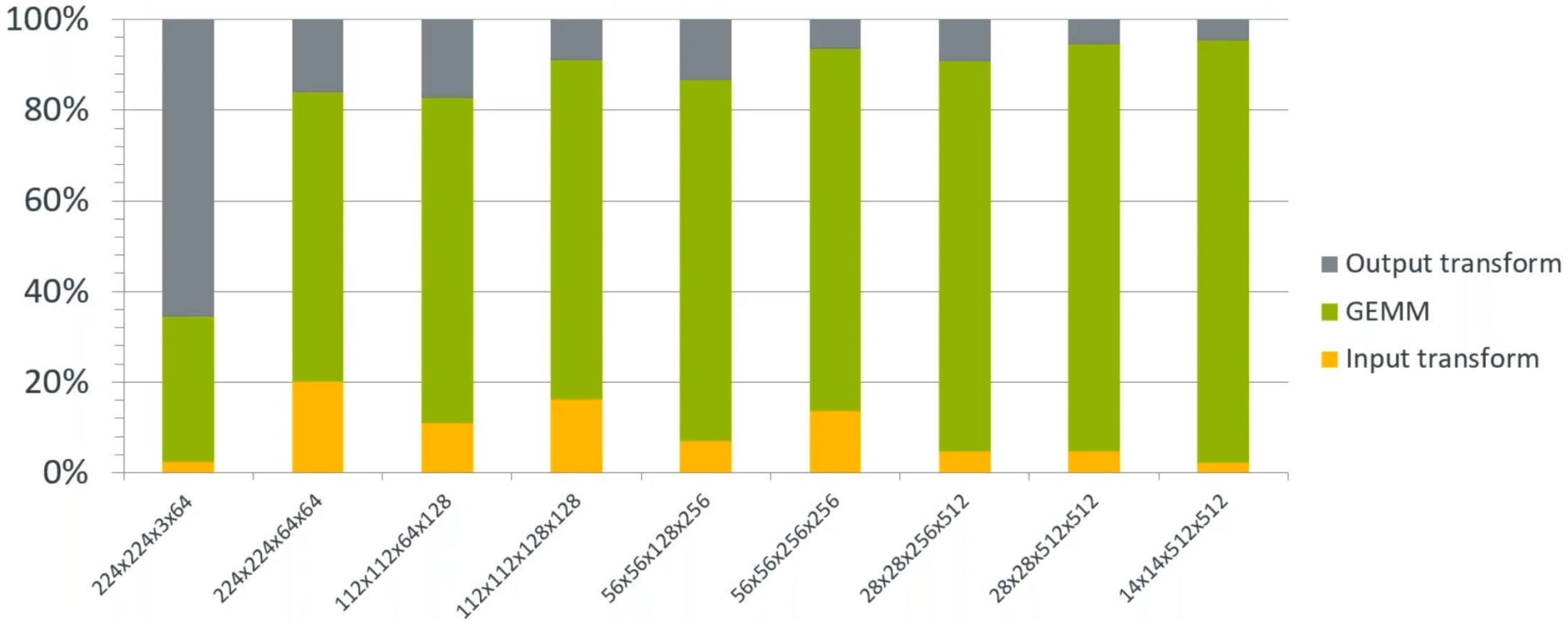
	n.a.	Filter transform	
Size (elements)		16 * Cin * Cout	

	Col2im (only if NCHW)	Output transform	Output transform / col2im
Size (elements)	W * H * Cout	16 * (W/2) * (H/2) * Cout	4

Winograd memory requirement = 16 * Cin * Cout + 4 * Col2im + 0.44 * Im2Col

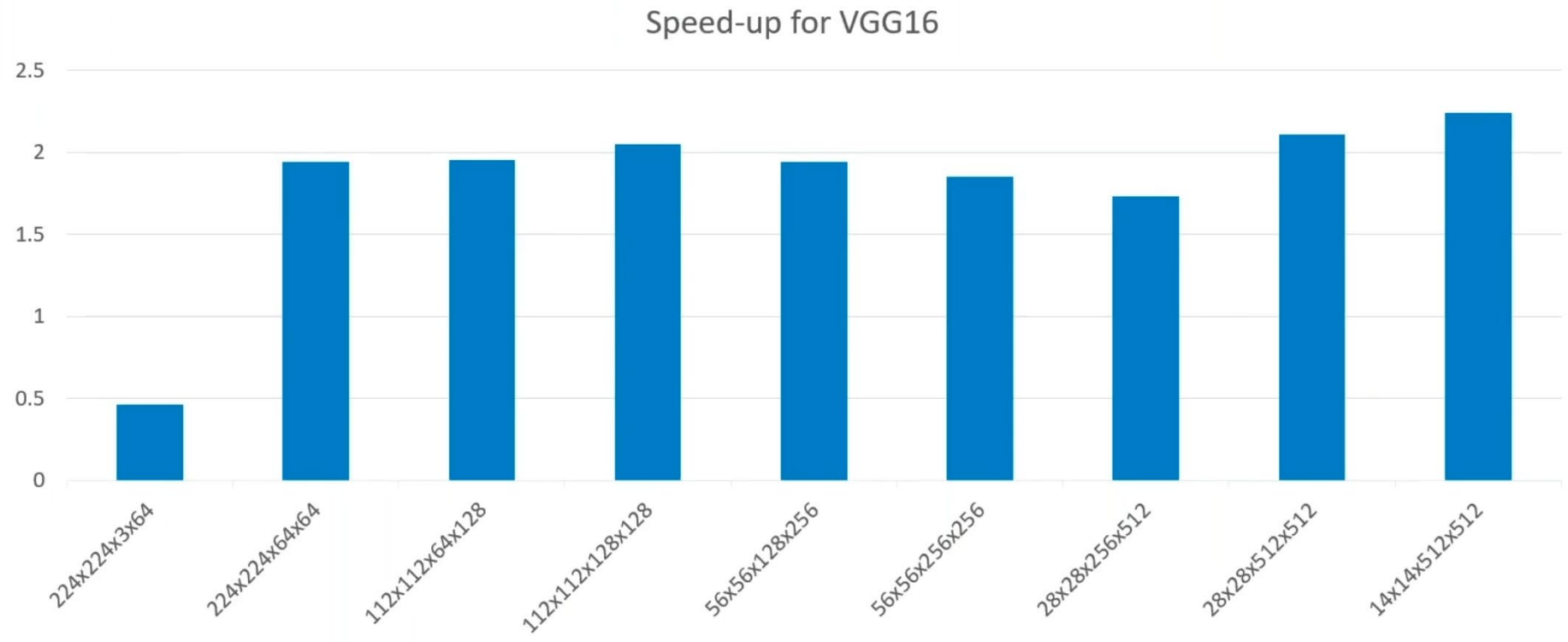
[Gian, 2018, AMD]

Winograd Convolution @ 2-D



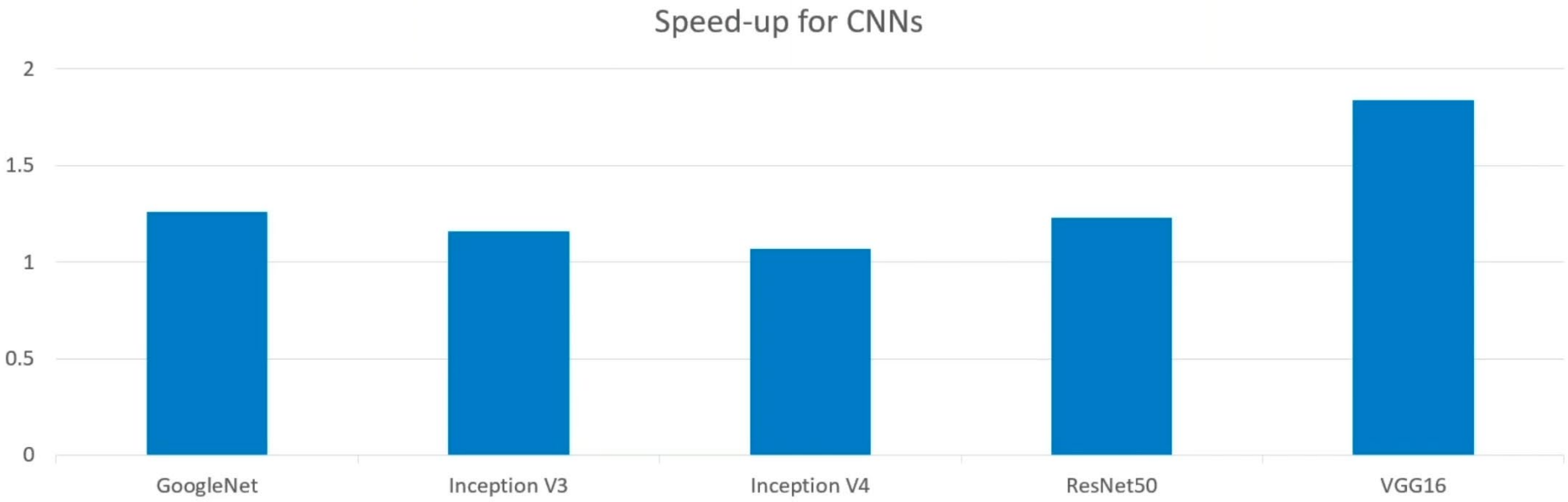
[Gian, 2018, AMD]

Winograd Convolution @ 2-D



[Gian, 2018, AMD]

Winograd Convolution @ 2-D



[Gian, 2018, AMD]

For Larger Output Tiles, e.g., F(4x4, 3x3) and F(6x6, 3x3)

2D $$Y = A^{\mathrm{T}}[(GgG^{\mathrm{T}}) \odot (B^{\mathrm{T}}dB)]A$$

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \quad G = \begin{bmatrix} 1/4 & 0 & 0 \\ -1/6 & -1/6 & -1/6 \\ -1/6 & 1/6 & -1/6 \\ 1/24 & 1/12 & 1/6 \\ 1/24 & -1/12 & 1/6 \\ 0 & 0 & 1 \end{bmatrix}$$

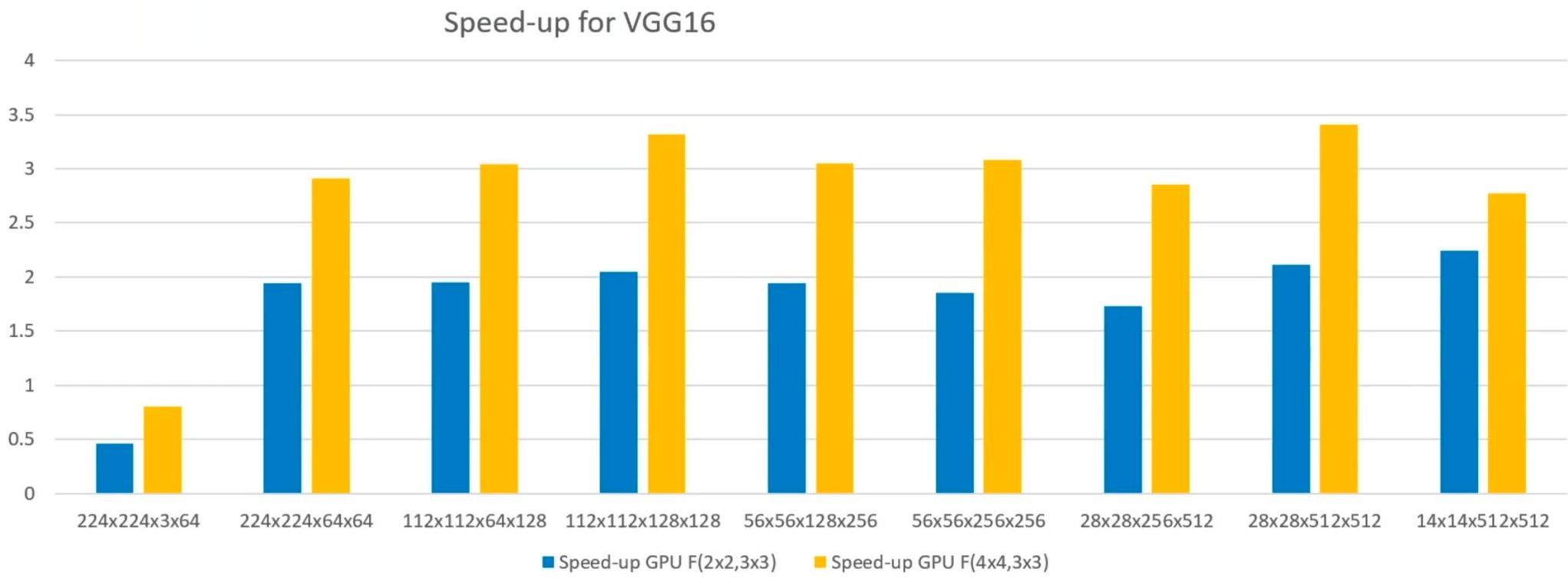
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 36 & 0 & -49 & 0 & 14 & 0 & -1 & 0 \\ 0 & 36 & 36 & -13 & -13 & 1 & 1 & 0 \\ 0 & -36 & 36 & 13 & -13 & -1 & 1 & 0 \\ 0 & -18 & -9 & 20 & 10 & -2 & -1 & 0 \\ 0 & 18 & -9 & -20 & 10 & 2 & -1 & 0 \\ 0 & 12 & 4 & -15 & -5 & 3 & 1 & 0 \\ 0 & -12 & 4 & 15 & -5 & -3 & 1 & 0 \\ 0 & -36 & 0 & 49 & 0 & -14 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1/36 & 0 & 0 \\ 1/48 & 1/48 & 1/48 \\ -1/48 & 1/48 & 1/48 \\ 1/120 & 1/60 & 1/30 \\ 1/120 & -1/60 & 1/30 \\ 1/720 & 1/90 & 1/30 \\ 1/720 & -1/90 & 1/30 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 3 & -3 & 0 \\ 0 & 1 & 1 & 4 & 4 & 9 & 9 & 0 \\ 0 & 1 & -1 & 8 & -8 & 27 & -27 & 0 \\ 0 & 1 & 1 & 16 & 16 & 81 & 81 & 0 \\ 0 & 1 & -1 & 32 & -32 & 243 & 243 & 1 \end{bmatrix}$$

Winograd Convolution @ 2-D



[Gian, 2018, AMD]

[Park, 2016]

Analyzing Data Access Patterns

F(2, 3) case

1D $Y = A^{\mathrm{T}}[U \odot V] = A^{\mathrm{T}}[(Gg) \odot (B^{\mathrm{T}}d)]$

$$V = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} d0 \\ d1 \\ d2 \\ d3 \end{bmatrix}$$

$$m_0 = (d_0 - d_2)g_0 \qquad m_1 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_2 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2} \qquad m_3 = (d_1 - d_3)g_2$$

F(2x2, 3x3) case

2D $Y = A^{\mathrm{T}}[(GgG^{\mathrm{T}}) \odot (B^{\mathrm{T}}dB)]A$

$$V = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} d00 & d01 & d02 & d03 \\ d10 & d11 & d12 & d13 \\ d20 & d21 & d22 & d23 \\ d30 & d31 & d32 & d33 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

V[0][0] = (d00-d20) – (d02-d22)
V[0][1] = (d01-d21) + (d02-d22)
…

[Park, 2016]

Code Optimization: Reordering Additions to Reduce L1 Data Cache Accesses

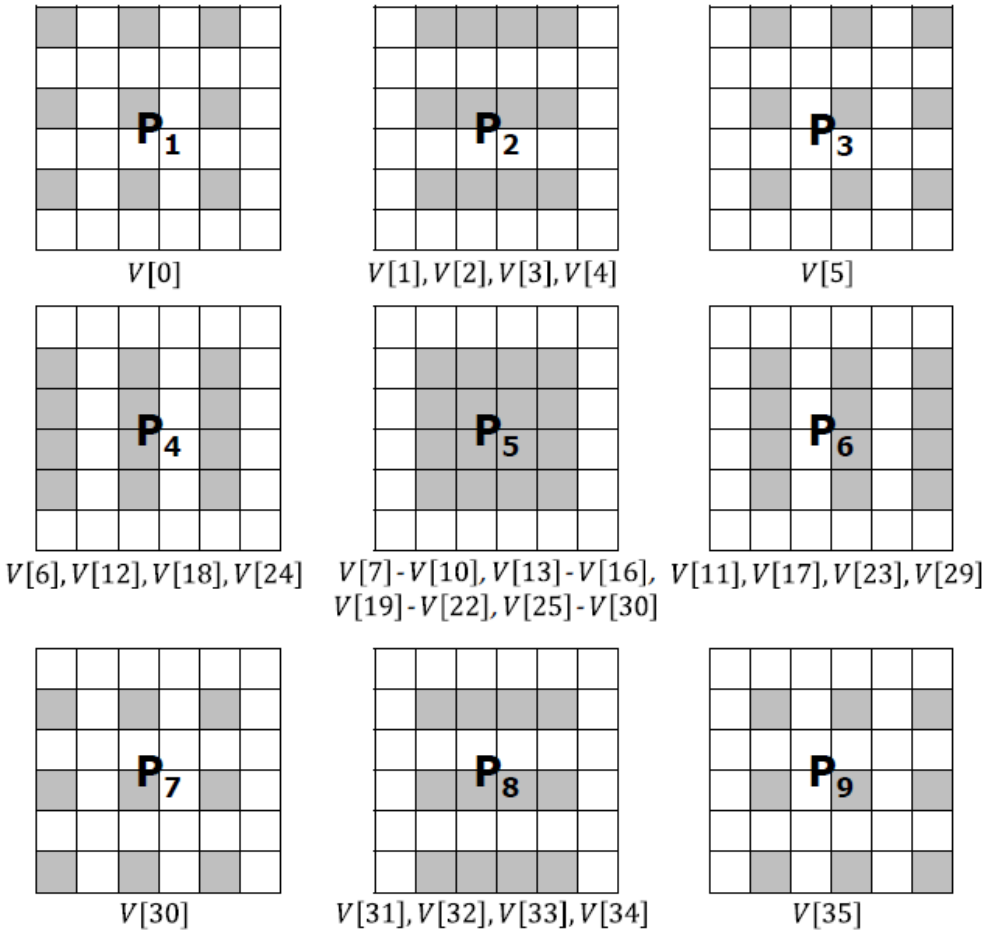
$$Y = A^{\mathrm{T}}[(GgG^{\mathrm{T}}) \odot (B^{\mathrm{T}}dB)]A$$

Step 1 ($V = B^{\mathrm{T}}dB$).

- Intra-group data reuse: Matrix V elements are grouped to maximize data reuse
  - E.g., V[11] and V[17] require the same set of data elements
- Inter-group data reuse: Groups are scheduled to maximize inter-group data reuse

$(P_1 \,\&\, P_2 \,\&\, P_3) \rightarrow P_6 \rightarrow P_5 \rightarrow P_4 \rightarrow (P_7 \,\&\, P_8 \,\&\, P_9)$

F(4x4, 3x3) case



[Park, 2016]

Tegra X1 (32 registers/thread)

- Addition runtime can be significantly reduced by reordering additions, i.e., register-level tiling