

# **Programming inverse memory hierarchy: case of stencils on GPUs**

Vasily Volkov

UC Berkeley

May 18, 2010

# Outline

- I got speedups in GEMM and FFT
  - 1.6x vs SGEMM in CUBLAS 1.1, now in CUBLAS
  - 3x vs CUFFT 1.1
- Lessons learned:
  - Offload storage from shared memory to registers
  - Compute few outputs per thread
- This talk:
  - Apply lessons to 3D stencils: more 2x speedups
  - Put in larger context

# Need more faster memory

- Bandwidth to DRAM is limited
- Keep working set in faster on-chip memory
  - Such as shared memory on GPU
  - Access shared memory instead of DRAM if possible
- However, shared memory is small
  - G80/GT200: only 16 KB per multiprocessor
  - May be not enough
- Wouldn't it be nice if we had more fast memory?
  - Hint: how 16,384 registers compare to shared memory?

# Trend: inverse memory hierarchy (1/3)

- Per multiprocessor/SIMD on GPUs:

	8800GTX	GTX280	Fermi	HD4870
registers	32KB	64KB	128KB	256KB
L1 storage	16KB	16KB	64KB	16KB
ratio	2x	4x	2x	16x

- L1 storage:
  - shared memory , L1 cache or both; LDS on ATI GPUs
- You have more registers than shared memory

# Trend: inverse memory hierarchy (2/3)

- New level of memory hierarchy on Fermi:

	Fermi (aggregate)
registers	2MB
L1 storage	1MB
L2 storage	768KB
ratio	2/1/0.75

- Further from processors, but smaller
  - Usually otherwise

# Trend: inverse memory hierarchy (3/3)

- Same trend on x86 architectures:

	Quad-core (total)	Larrabee (per core)
SIMD registers	1KB	8KB
L1 D\$	128KB	32KB
L2 cache	8MB	256KB
ratio	1/128/8192	1/4/32

- Gap between memory hierarchy levels decreases

# Why inverse?

- Single thread won't see inverse hierarchy:
  - Up to 256B registers per thread on Fermi
  - Up to 64KB shared memory/L1 cache
  - Up to 768 KB L2 cache
- Inversion comes from parallelism
  - Now run 1024 threads on same multiprocessor
  - This is 256KB registers!
- Registers scale with SIMD and multithreading
  - Shared memory/L1 cache don't have to

# Private storage requires replication

- Can't access shared memory on other multiprocessor
  - Get your own copy instead
- Can't access registers in other threads
  - Get your own copy instead
  - Is it common?
  - Yes: pointers, counters, temporary values

# Register usage in CUBLAS 1.1 SGEMM

warp 1	warp 2	warp 3	warp 4	warp 5	warp 6	warp 7	warp 8	warp 9	warp 10	warp 11	warp 12	warp 13	warp 14	warp 15	warp 16
A's pointer															
A's pointer															
B's pointer															
B's pointer															
counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter
C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data
C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data
C's index															
C's index															
B's pointer in shared memory															
lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****

Exact replicas (512x): counter, lda

Redundant storage: pointers/indices – 7 in total

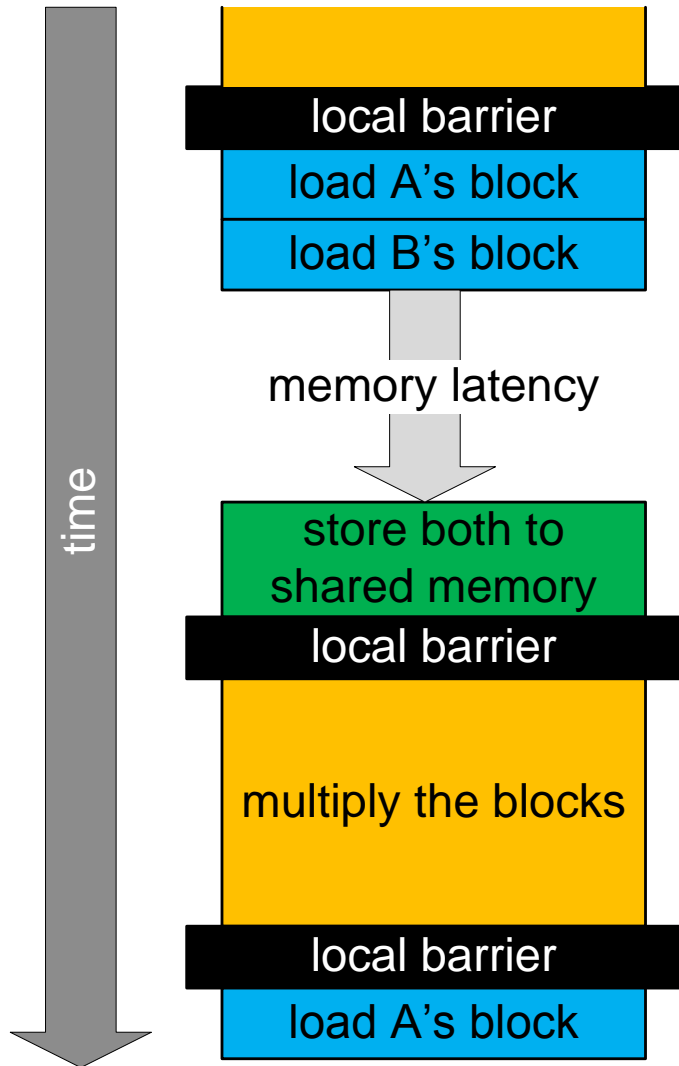
Scratch space: 4 registers/thread



# Hiding latency using fewer threads

- Less thread parallelism = poor latency hiding?
  - Not necessarily!
- Hiding latency requires memory parallelism
  - it can be supplied using thread parallelism
  - Or instruction-level parallelism
- Little's law:
  - data in transit [B] = latency [s] \* bandwidth [B/s]
- **Same work with fewer threads = same memory parallelism**

# Need thread blocks to hide latency

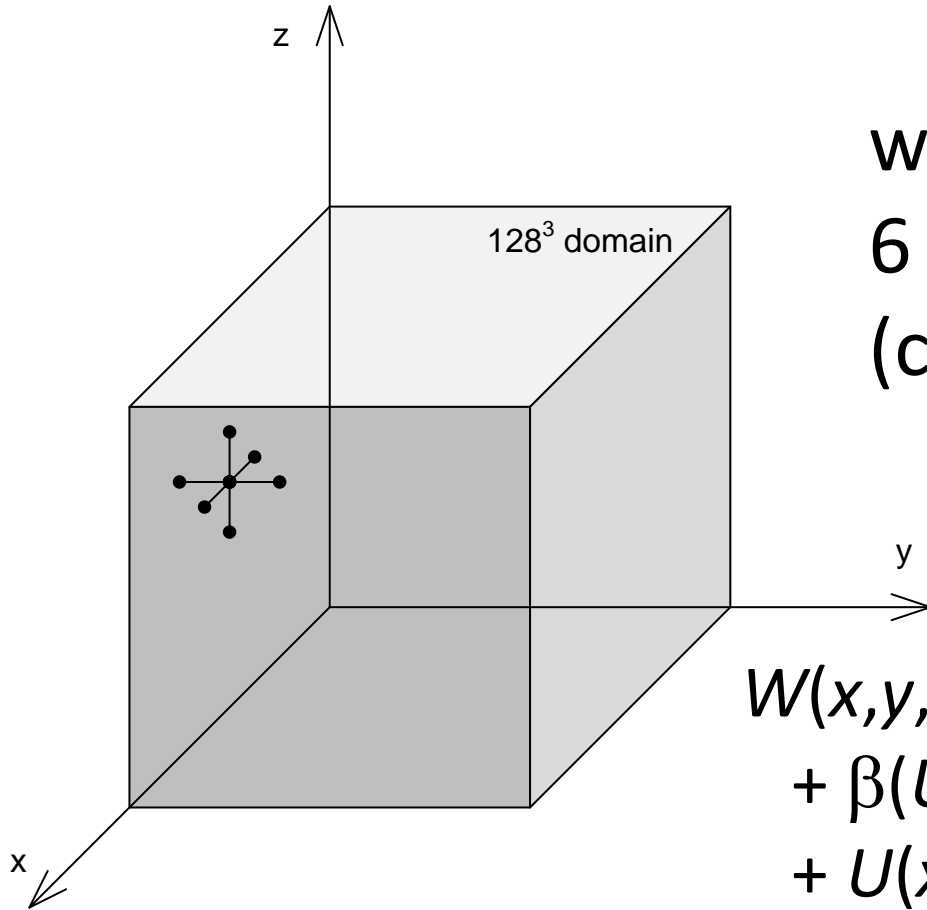


Data dependencies in thread block may prevent overlapping latency with computation

Need several thread blocks per SM  
2-4 usually suffice

- How to apply this to 3D stencils

# 7-point stencil in 3D



weighted average with  
6 neighbors in 3D  
(common in FDM)

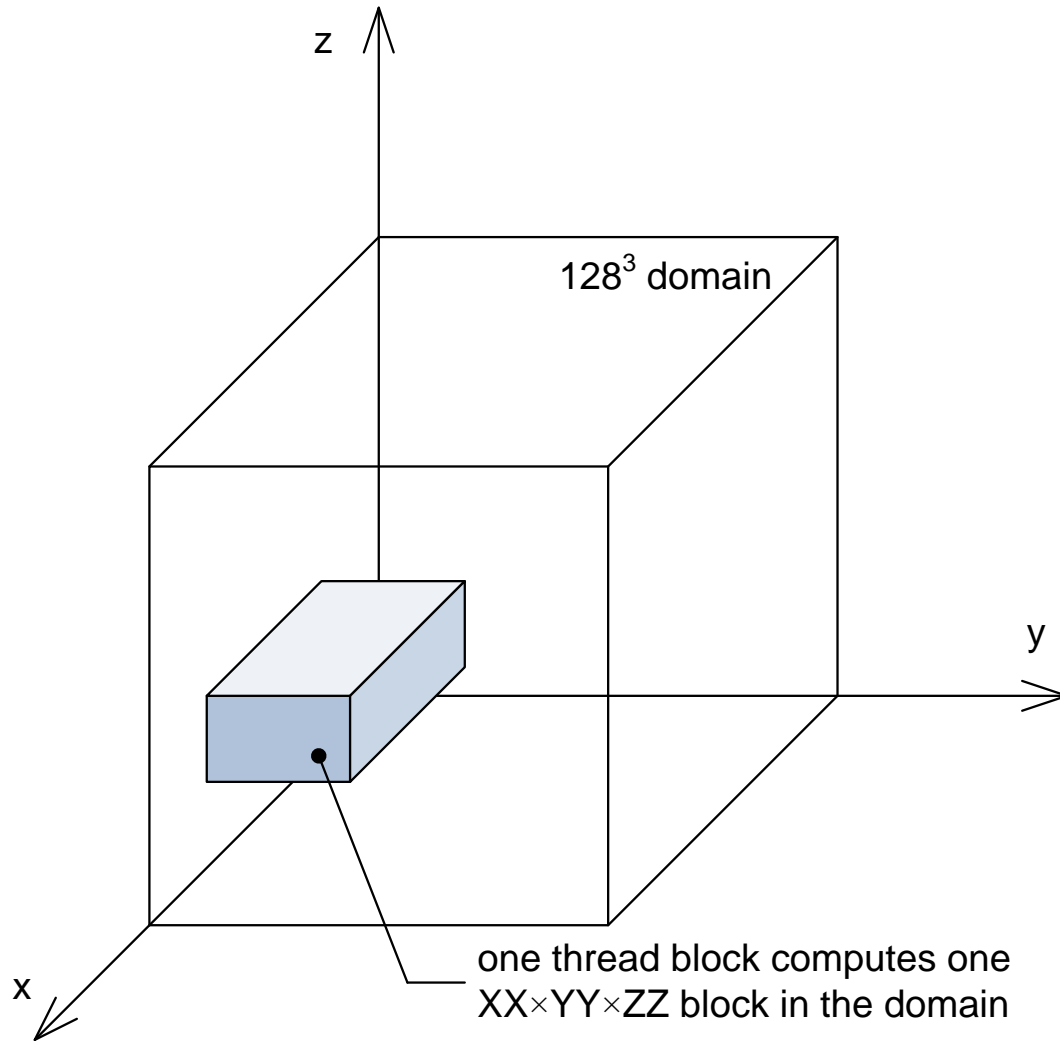
$$\begin{aligned} W(x,y,z) := & \alpha U(x,y,z) \\ & + \beta(U(x-1,y,z) + U(x+1,y,z) \\ & + U(x,y-1,z) + U(x,y+1,z) \\ & + U(x,y,z-1) + U(x,y,z+1)) \end{aligned}$$

# Its arithmetic intensity is low

$$W(x,y,z) := \alpha U(x,y,z) + \beta(U(x-1,y,z) + U(x+1,y,z) + U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1))$$

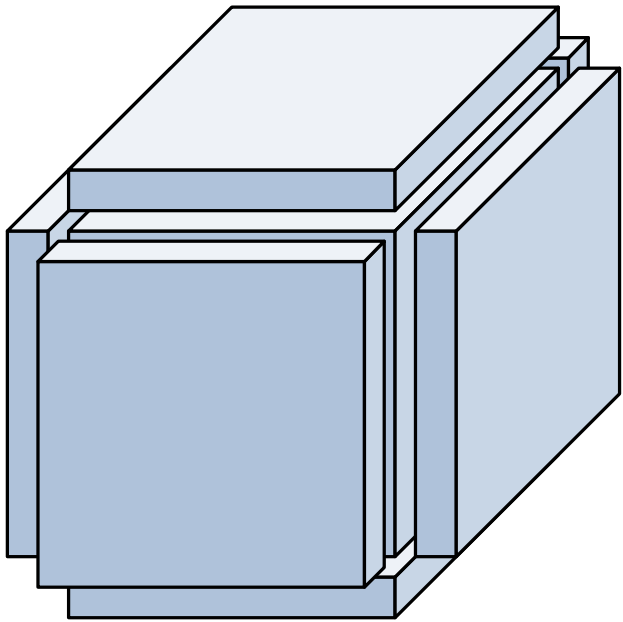
- Stencil: 8 flops, 7 reads, 1 write
  - Arithmetic intensity of stencil = **1 flop/word**
- GPU: 624 Gflop/s, 142 GB/s
  - Arithmetic intensity of GPU = **18 flop/word**
  - Can do many more flops per each word fetched
  - ALUs will be underutilized
- That's why we want to use fast memory

# Partition the domain into blocks

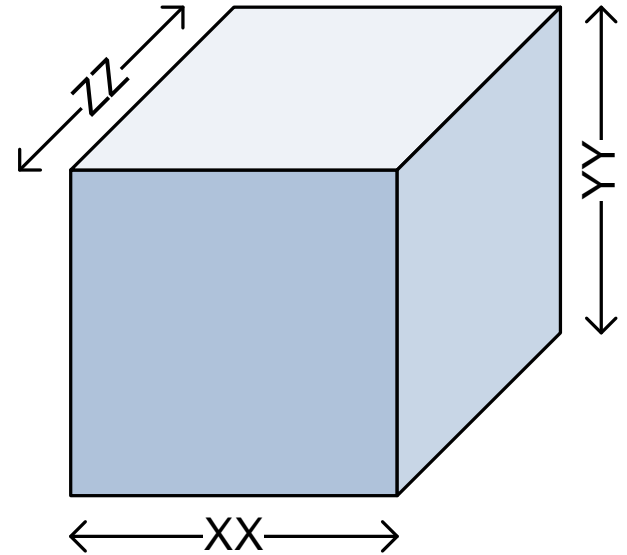
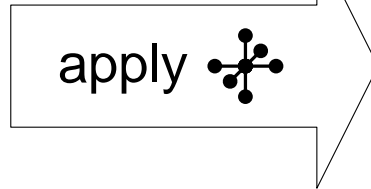


# What's the new arithmetic intensity?

need so much data



to compute so many results



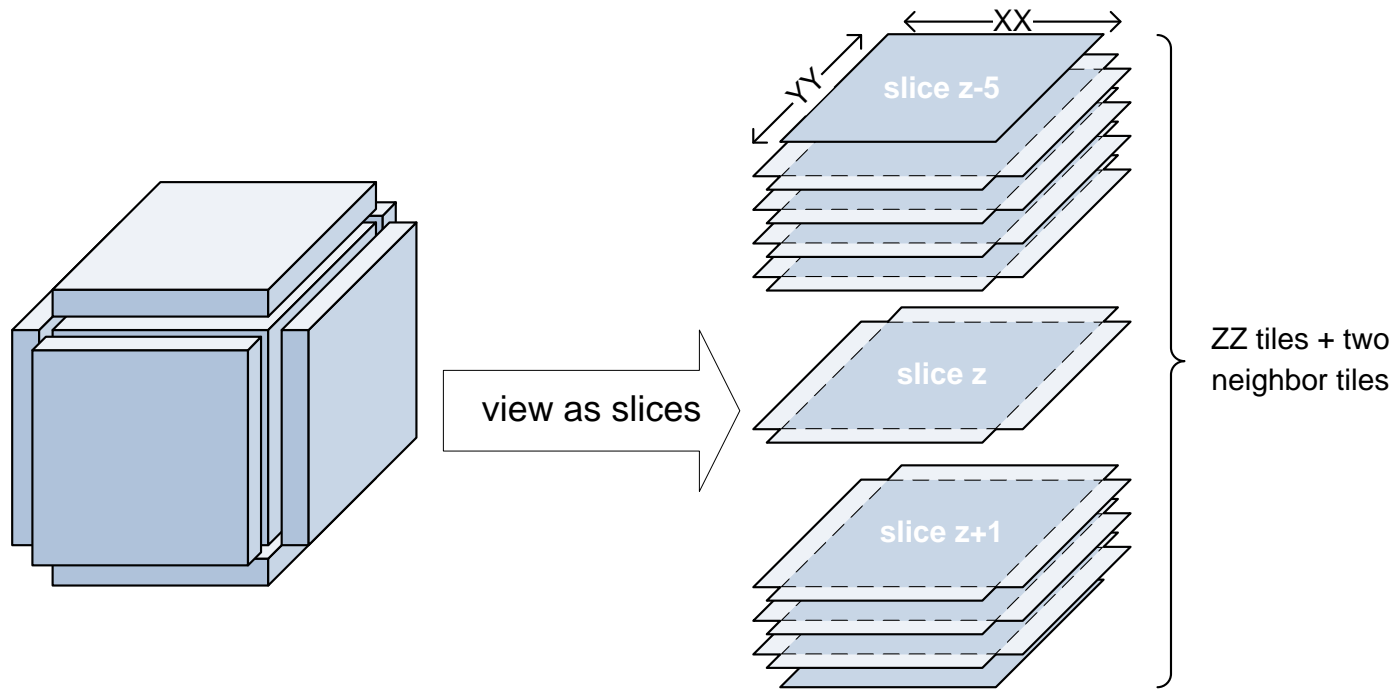
Arithmetic intensity =  $4/(1+1/XX+1/YY+1/ZZ)$

32x32x32 blocks yield AI = **3.7 flop/word**

- Can't get above 4 anyway

# Circular queue technique

- Storing 3D blocks in fast memory is expensive
  - 128KB for 32x32x32 block, single precision



- Split into 2D slices
  - Need only 3 input slices to compute 1 output

# Where to store the slices?

- Three 32x32 slice takes 12KB
- Only one thread block will fit
- Might need using smaller slices instead
- Or, **offload storage into registers**

# Offload slices to registers

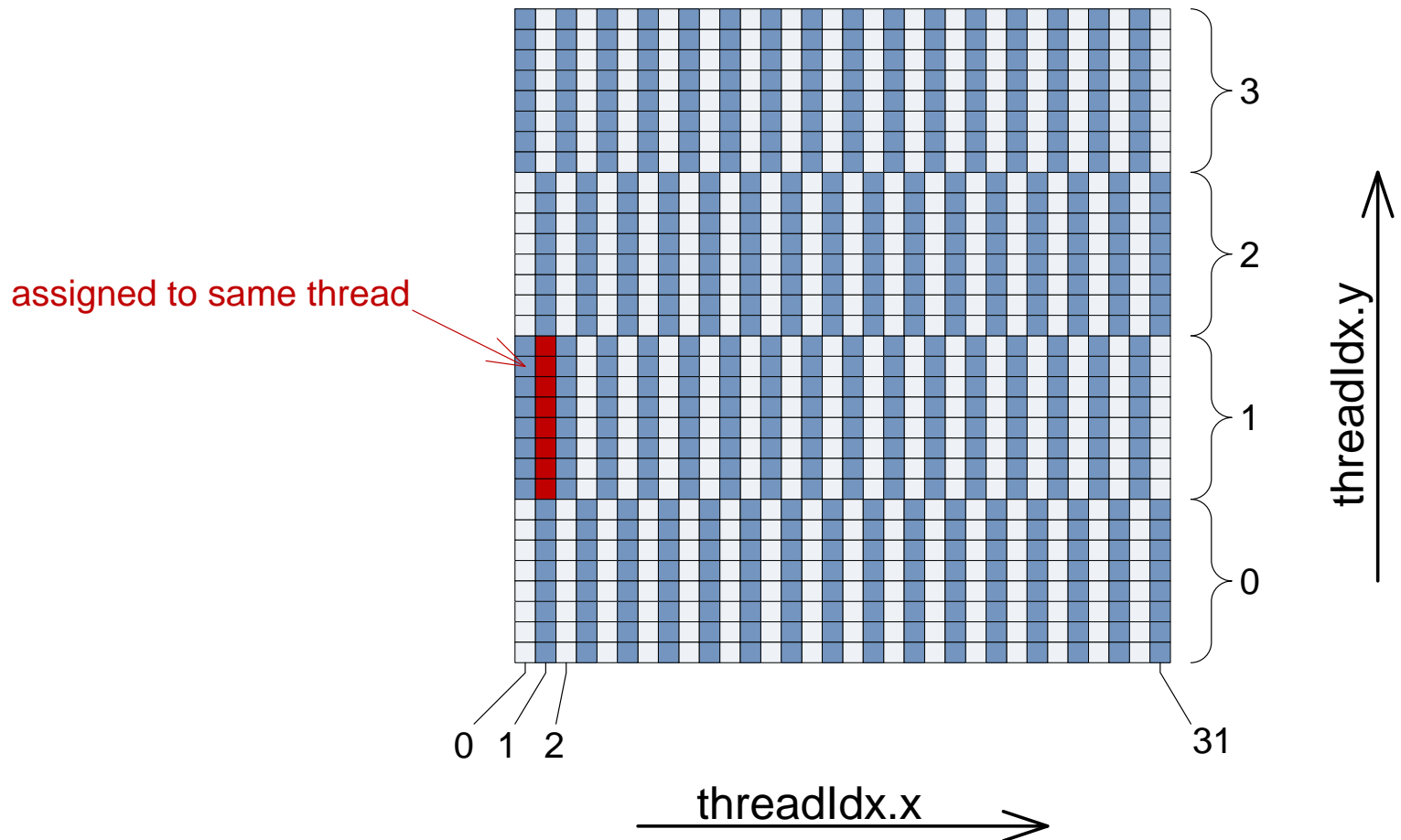
- Distribute slices across thread registers
- Thread  $(x,y)$  gets  $U(x,y,z)$ ,  $U(x,y,z)$  and  $U(x,y,z)$

$$W(x,y,z) := \alpha U(x,y,z) + \beta (U(x-1,y,z) + U(x+1,y,z) + U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1))$$

- 4 out of 8 memory accesses are to **local registers**
- 4 other accesses are to **registers in other threads**
- But they all are to same slice
- Keep 1 slice in shared memory, 2 in registers

# Compute few outputs per thread

32×32 tile requires 1024 threads if computing 1 output per thread  
But only 128 threads by computing 8 elements per thread



# 27-point stencil

$$\begin{aligned} W(x,y,z) = & C_0 * U(x,y,z) + C_1 * (U(x-1,y,z) + U(x+1,y,z) + \\ & U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1)) + \\ & C_2 * (U(x-1,y-1,z) + U(x-1,y+1,z) + U(x+1,y-1,z) + \\ & U(x+1,y+1,z) + U(x,y-1,z-1) + U(x,y-1,z+1) + \\ & U(x,y+1,z-1) + U(x,y+1,z+1) + U(x-1,y,z-1) + \\ & U(x-1,y,z+1) + U(x+1,y,z-1) + U(x+1,y,z+1)) + \\ & C_3 * (U(x-1,y-1,z-1) + U(x-1,y-1,z+1) + \\ & U(x-1,y+1,z-1) + U(x-1,y+1,z+1) + \\ & U(x+1,y-1,z-1) + U(x+1,y-1,z+1) + \\ & U(x+1,y+1,z-1) + U(x+1,y+1,z+1)) \end{aligned}$$

- 30 flops, 27 reads, 1 write — **1.1 flop/word**
- Arithmetic intensity =  $30 / (1 + (1 + 2/XX) * (1 + 2/YY) * (1 + 2/ZZ))$
- For 32x32x32 this is **14 flop/word**

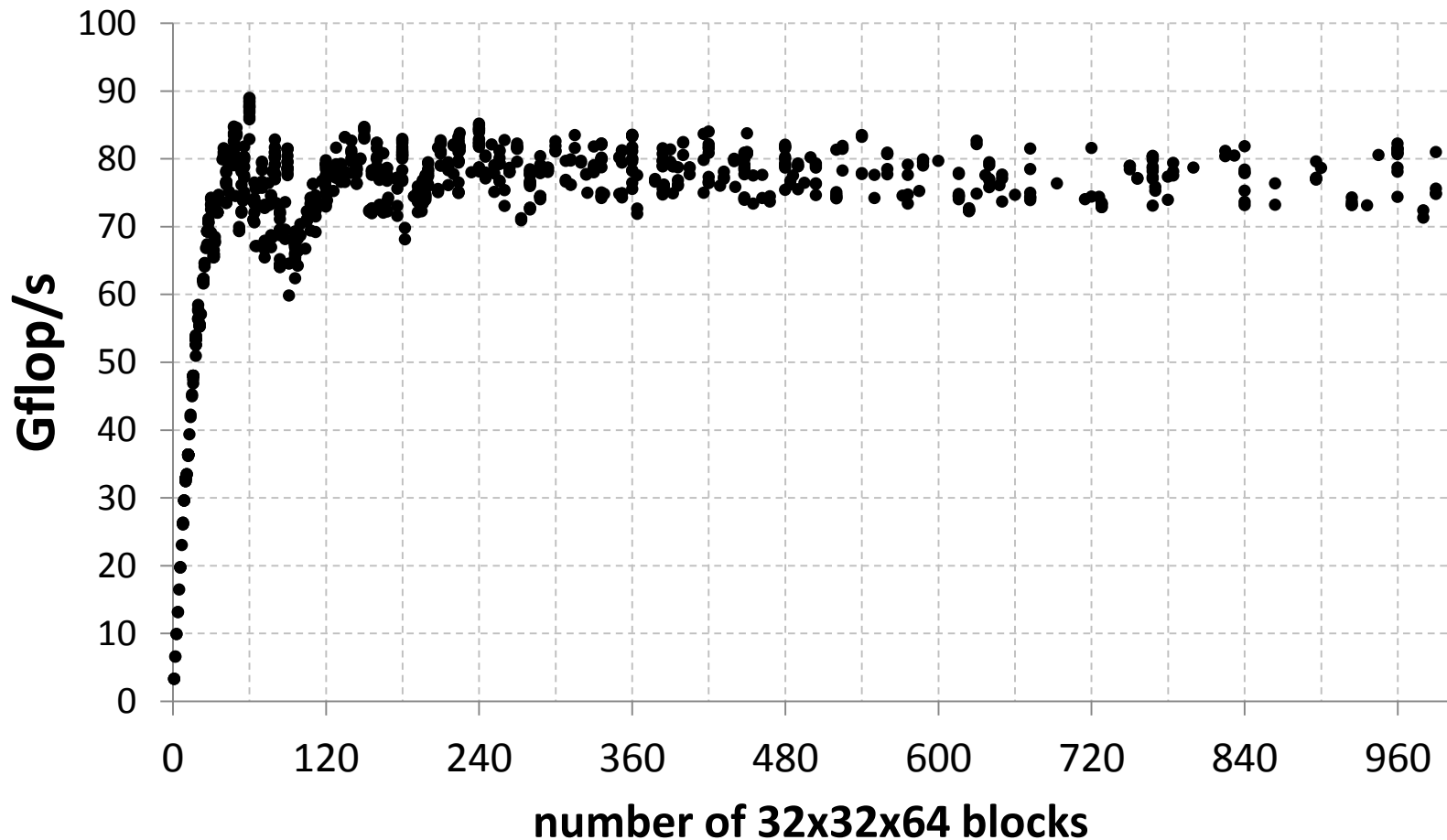
# 27-point stencil

$$\begin{aligned} W(x,y,z) = & C_0 * U(x,y,z) + C_1 * (U(x-1,y,z) + U(x+1,y,z) + \\ & U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1)) + \\ & C_2 * (U(x-1,y-1,z) + U(x-1,y+1,z) + U(x+1,y-1,z) + \\ & U(x+1,y+1,z) + U(x,y-1,z-1) + U(x,y-1,z+1) + \\ & U(x,y+1,z-1) + U(x,y+1,z+1) + U(x-1,y,z-1) + \\ & U(x-1,y,z+1) + U(x+1,y,z-1) + U(x+1,y,z+1)) + \\ & C_3 * (U(x-1,y-1,z-1) + U(x-1,y-1,z+1) + \\ & U(x-1,y+1,z-1) + U(x-1,y+1,z+1) + \\ & U(x+1,y-1,z-1) + U(x+1,y-1,z+1) + \\ & U(x+1,y+1,z-1) + U(x+1,y+1,z+1)) \end{aligned}$$

- Only 4 out of 28 accesses are to local registers
- Trick: have 1 input plane, 3 output planes instead

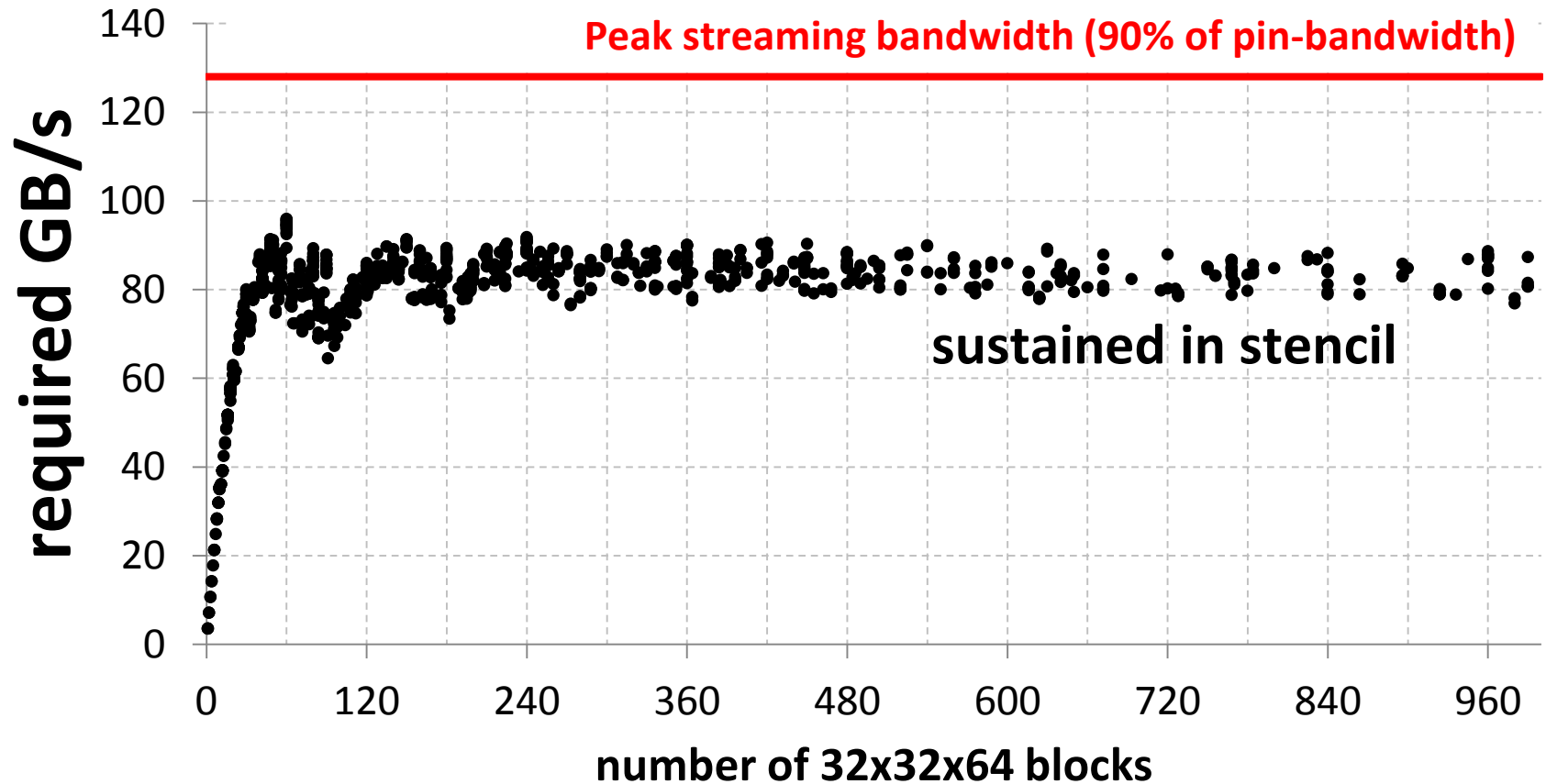
- Performance results

# Single precision, 7-point, GTX280



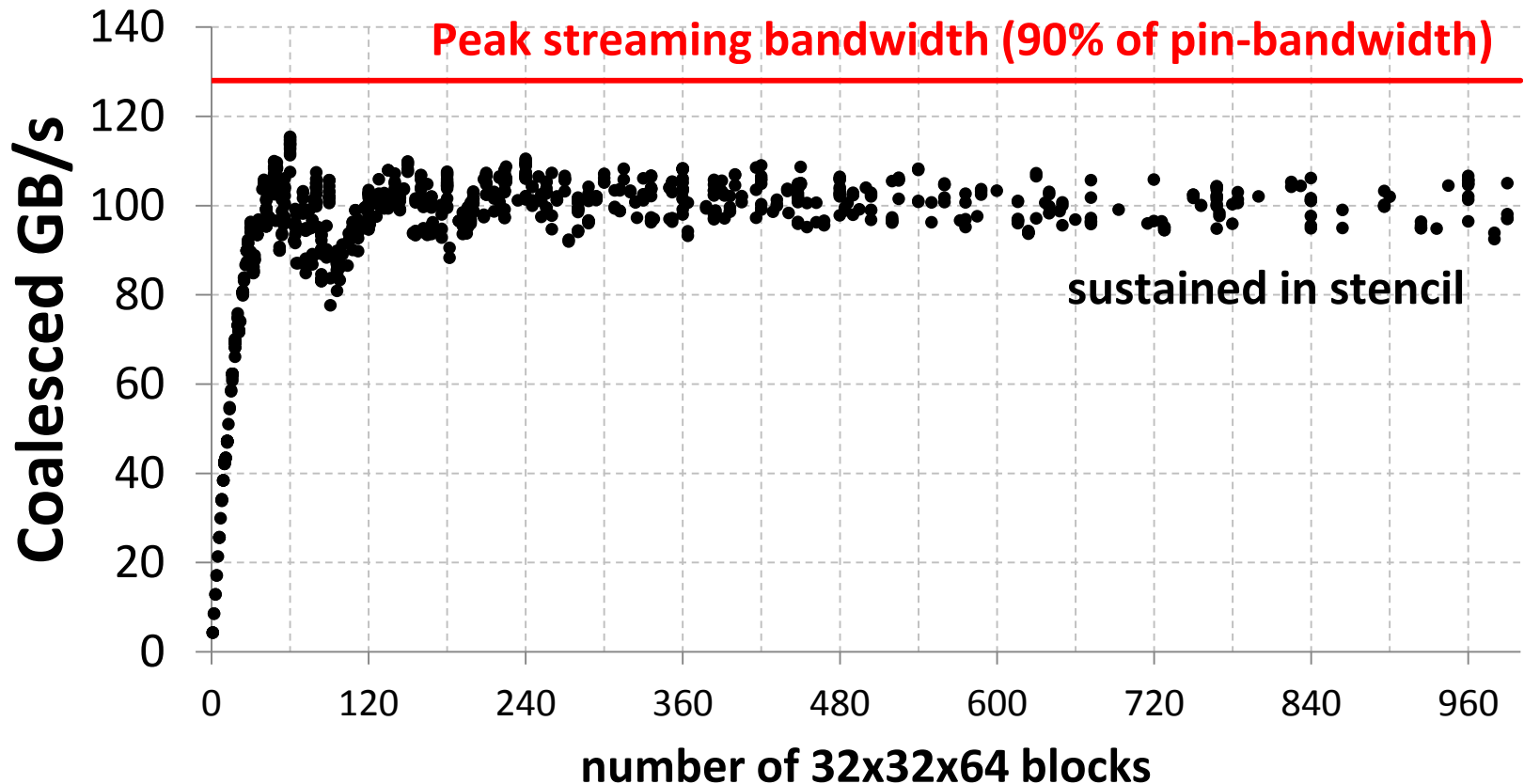
90 Gflop/s sustained; peak in add = 312 Gflop/s  
Still very far from being compute bound

# Single precision, 7-point, GTX280



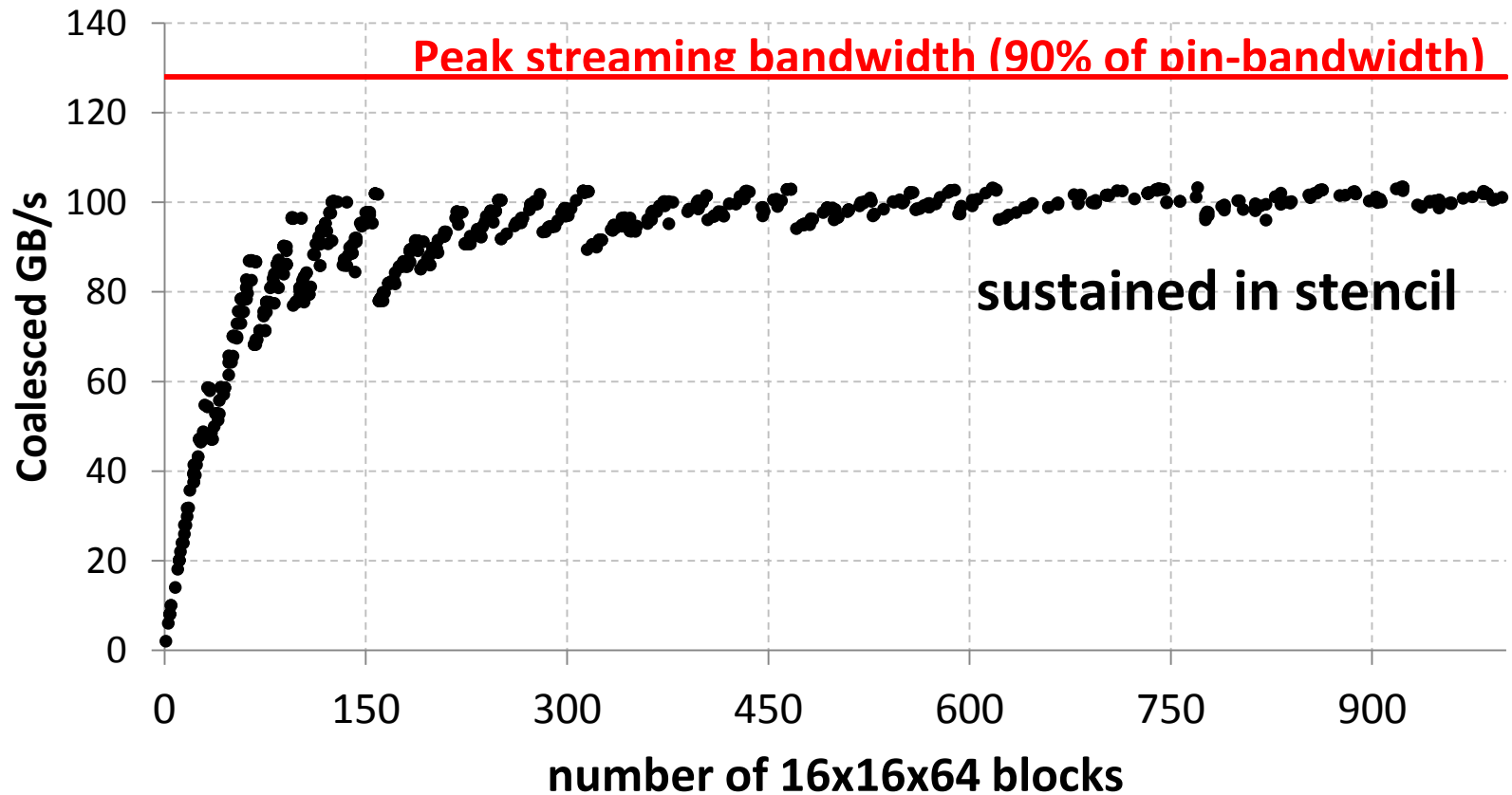
- Result: 75% of peak streaming bandwidth

# Single precision, 7-point, GTX280



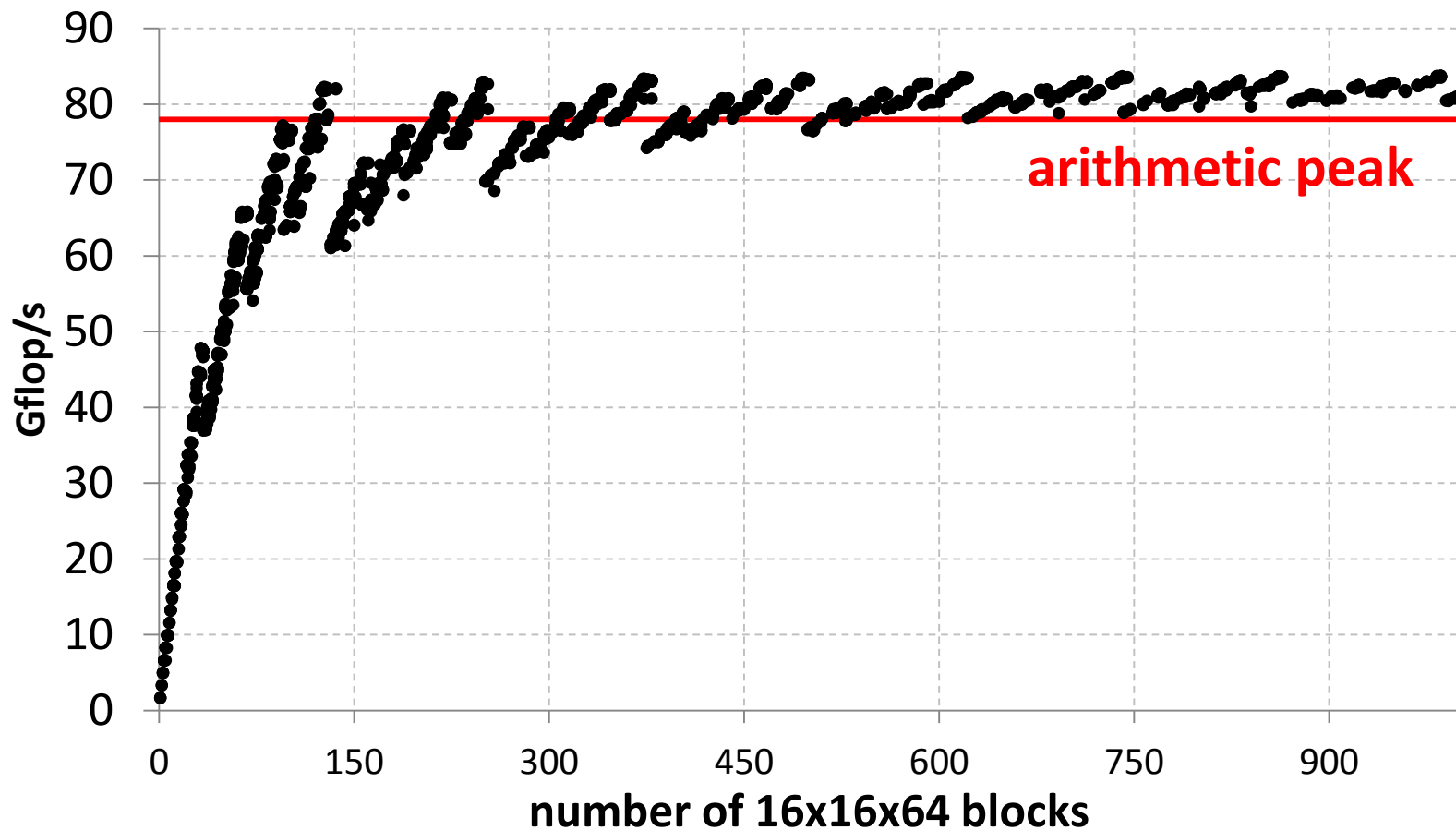
- Count coalescing overhead (+20%): 90% of peak streaming bandwidth
- Can you do better: use larger or non-square tiles, jam stencils

# Double precision, 7-point, GTX280



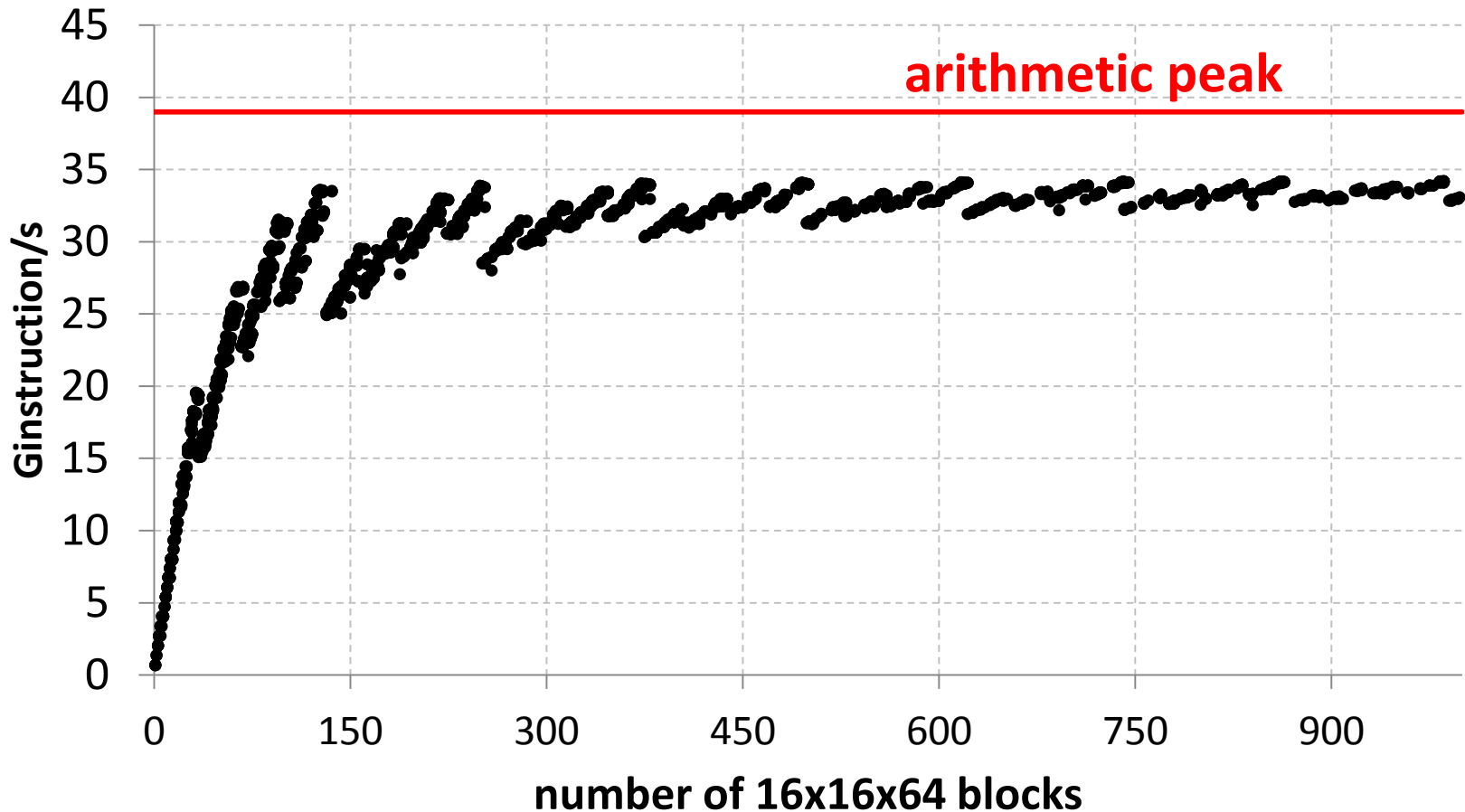
- Bandwidth-bound, close to peak

# Double precision, 27-point, GTX280



- Faster than peak? Hidden compiler optimization
  - 120 flops gets compiled into 49 instructions

# Double precision, 27-point, GTX280



- Compute bound, close to peak

# Comparison with Sparse matrix-vector multiply

- Match vs. SpMV by Bell and Garland [2008]
- SpMV does not exploit stencil structure
  - More general but slower

	Single precision	Double precision	
	7-point	7-point	27-point
NVIDIA's SpMV	0.42 ns/stencil	0.87 ns/stencil	3.2 ns/stencil
Our stencil code	0.09 ns/stencil	0.21 ns/stencil	0.36 ns/stencil
Speedup	4.7x	~4.2x	~8.9x

# Red-Black Gauss Seidel

- Work by Jonathan Cohen
  - Similar to shown at last ParCFD
- Highly optimized, uses texture caches
  - 23 Gflop/s for  $128^3$  volume
  - 61ms for multigrid
- My code is  $\sim 2x$  faster
  - 52 Gflop/s for  $128^3$  volume
  - 34ms for multigrid

# Conclusion

- Look for early insights into inverse hierarchies
- May get another 2x speedup on GPU if:
  - Offload storage from shared memory to registers
  - Compute multiple outputs per thread