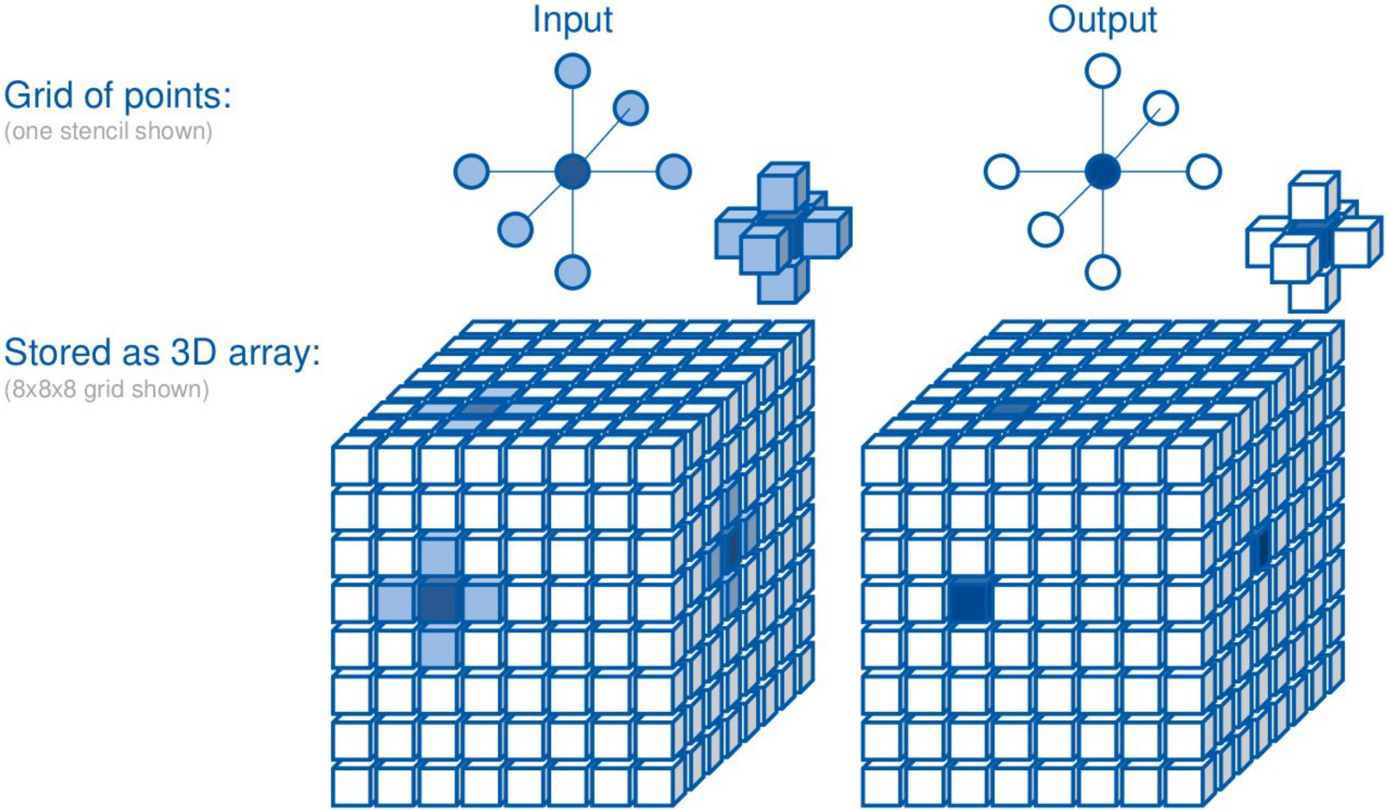
Stencil

What is a Stencil?

- The stencil computation pattern refers to a class of computations on a grid where the value at a grid point is computed based on neighboring points
    - It bears a strong resemblance to convolution

9.49	4.89	0.23	9.16	1.10	2.00	8.69
6.07	2.90	2.75	0.12	4.80	7.06	6.59
3.42	2.38	9.39	5.30	0.67	7.00	2.54
8.35	2.56	7.91	9.20	3.29	2.13	5.75
3.46	3.85	3.71	9.89	9.41	4.27	0.84

Seven-Point 3D Stencil



Why does it matter?

- Used to solve partial differential equations
  - Fluid dynamics, heat conductance, weather forecasting, electromagnetics
- The data that is processed by stencil-based algorithms consists of discrete quantities of physical significance
  - Mass, velocity, force, temperature
- Common use of stencils is to approximate the derivative values of a function based on the function values within a range of input variable values
  - Due to the numerical accuracy requirements in solving differential questions, stencils tend to use high-precision floating data that consumes more memory

Differential Equations Example

- Suppose we have *f(x)* discretized into a 1D grid array F, and we would like to calculate the discretized derivative of *f(x)*, *f'(x)*

$$f^{'}(x) = \frac{f(x+h) - f(x-h)}{2h} + O\left(h^2\right)$$

- The derivative of a function at point x can be approximated by the difference of the function values at two neighboring points divided by the distance between points
  - The value h is the spacing between neighboring points in the grid
- The error is expressed by the term $O(h^2)$
  - The lower, the better

Differential Equations Example

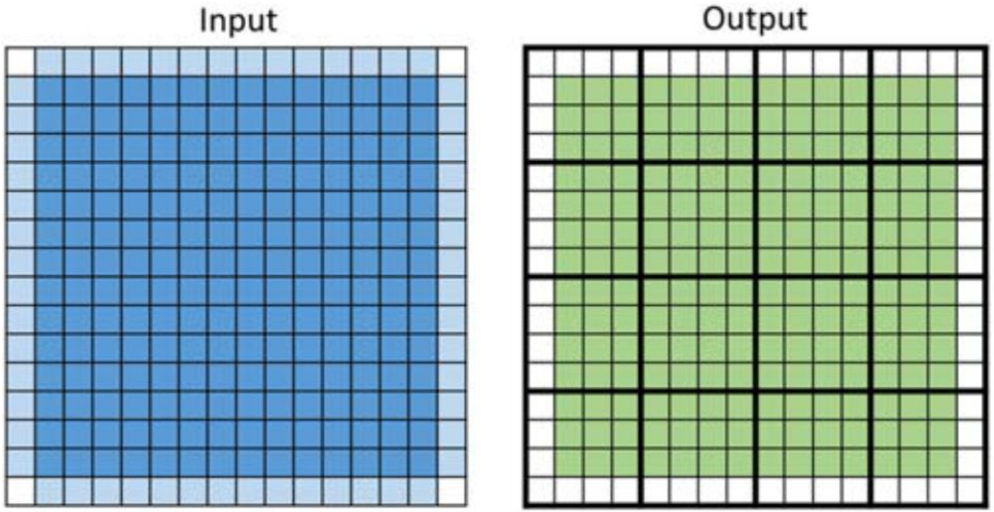
- We can rewrite the previous formula as

$$FD[i] = \frac{-1}{2*h} * F[i-1] + \frac{1}{2*h} * F[i+1]$$

- Thus, we obtain a 1D three-point stencil $\left[\frac{-1}{2h},\ 0,\ \frac{1}{2h}\right],$
- If we increase the order of the derivative also, the number of points increase
  - The second derivative will use a 1D five-point stencil

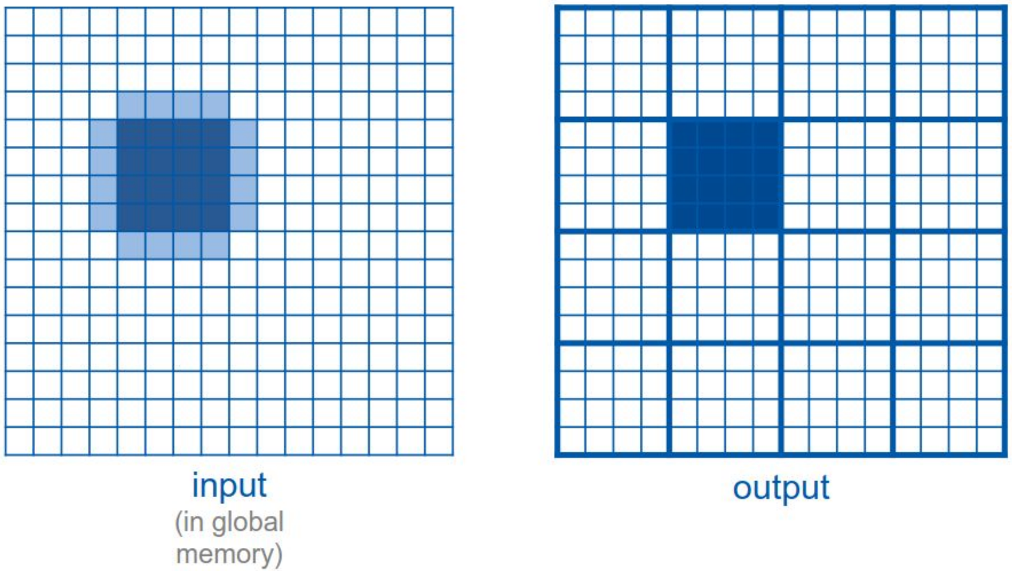
Naive Implementation

- No dependence between output grid points when generating the output grid point values within a stencil sweep
- **Parallelization Approach**
  - Assign one thread per output grid point (use 3D grid of threads)



Naive Implementation - AI

- Each thread performs 13 floating-point operations
    - seven multiplications and six additions
- Each thread loads seven four-byte value
- Thus the AI is $13/(7*4) = 0.46\ OP/B$ which we can improve



Problems and Optimizations

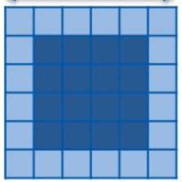
- Tiling and Privatization
- Coarsening and Slicing
- Register Tiling

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

Tiling and Privatization

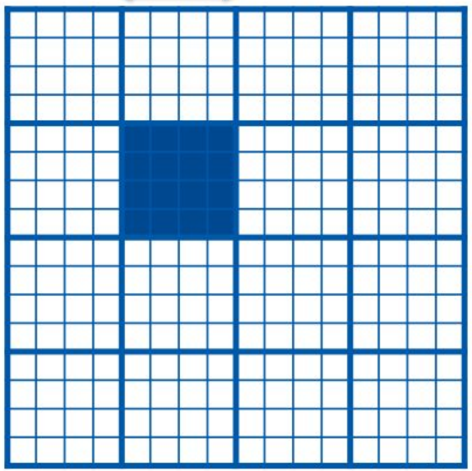
- Use Shared Memory Tiling to increase the AI
- All threads are active when loading input into shared memory
- Only internal threads are active when computing and storing the output tile

input tile dimension



$input_{tile}$
(in shared memory)

output tile dimension

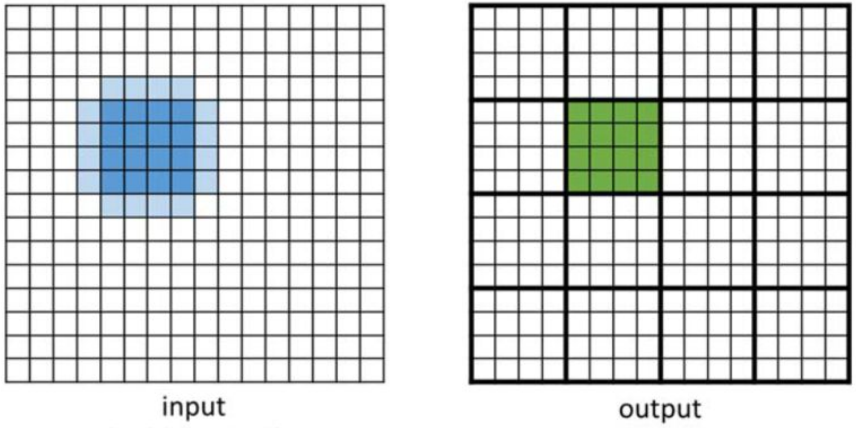


output

input tile dimension

Tiling and Privatization - Data Reuse

- Compared to a convolution, the data reuse of a stencil is lower
  - For a 2D five-point stencil is 2.5 OP/B, which is lower than a 4.5 OP/B of a 3x3 convolution
  - The greater the number of points/kernel size, the greater the differences in data reuse
- The input tiles of the five-point stencil do not include the corner grid points
  - The benefit of loading an input grid point value into the shared memory for a stencil sweep can be significantly lower than that for convolution



Tiling and Privatization - AI

- Consider T as the tile dimensions
- Each block has $(T-2)^3$ active threads calculating an output value
  - Each active thread performs 13 floating-point multiplication or addition operations
- With a total of $13*(T-2)^3$ floating-point arithmetic operation
- Each block loads an input tile by performing T^3 loads that are 4 bytes each
- Thus the AI is

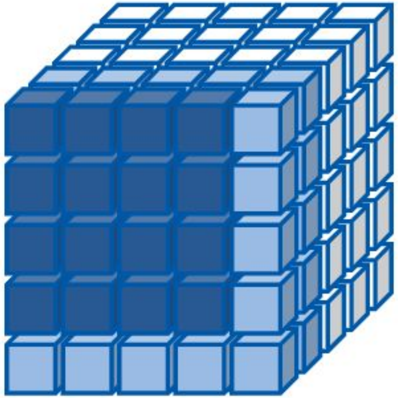
$$\frac{13*(T-2)^3}{4*T^3} = \frac{13}{4}*\left(1-\frac{2}{T}\right)^3 \quad OP/B$$

- Asymptotically, the upper bound is 3.25 OP/B
  - For $T=8$ the ratio for a seven-point stencil is only 1.37 OP/B
- We are still limited
  - in T by the number of threads we can spawn, in this case, 8x8x8
  - In the amount of shared memory available

Coarsening and Slicing

- We can overcome the block-size limitation by using thread coarsening
- The idea is for the thread block to iterate in the z-direction, calculating the values of grid points in one x-y plane of the output tile during each iteration
  - Thread blocks process tile consisting of the same number of threads as one x-y plane of the input tile
- Only store the three input planes needed for the output at that iteration
- The threads becomes T^2 instead of T^3, thus we an AI of 2.68 OP/B
  - Closer to the asymptotic limit of 3.25 OP/B
- The shared memory capacity requirement is now $3*T^2$ elements instead of T^3 elements
  - For $T=32$ the shared memory consumption is now at a reasonable level of $3*32^2*4B=12KB$ per block

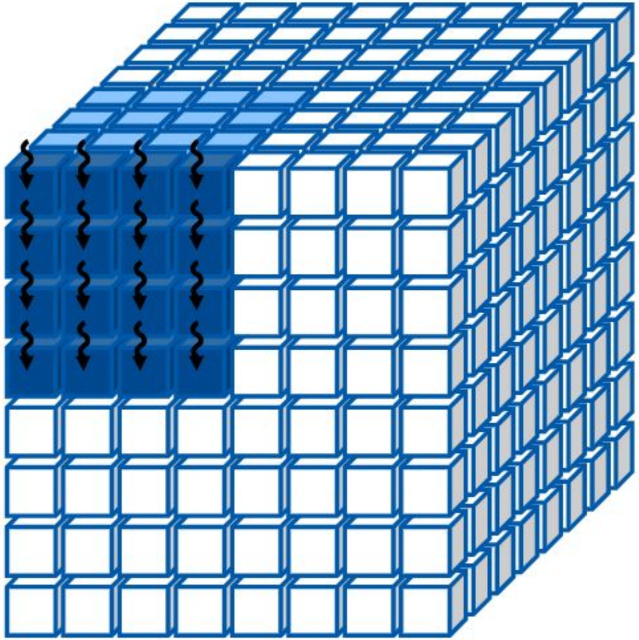
Coarsening and Slicing



$\text{input}_{\text{tile}}$
(in shared memory)

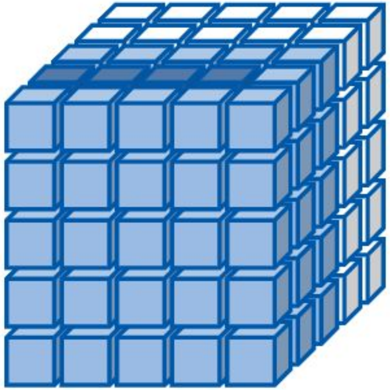
<u>Solution:</u>
Only store the three input planes needed by the output plane at a time

<u>Solution:</u>
Assign enough threads for loading one input plane and processing one output plane



output

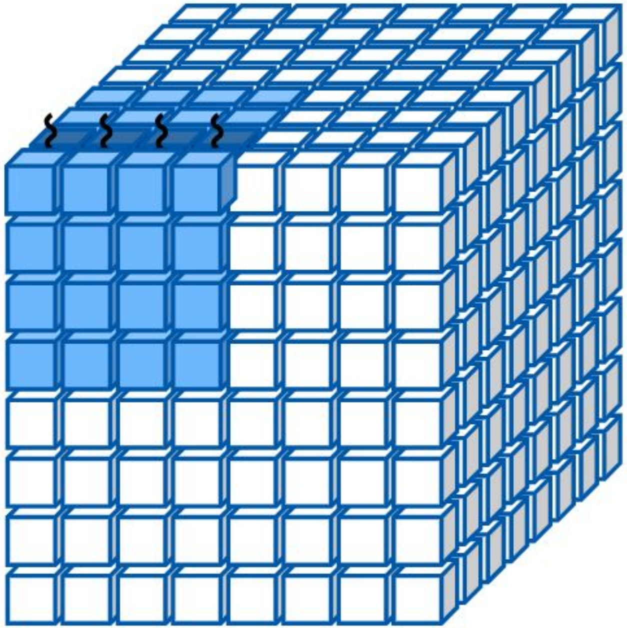
Coarsening and Slicing



$\text{input}_{\text{tile}}$
(in shared memory)

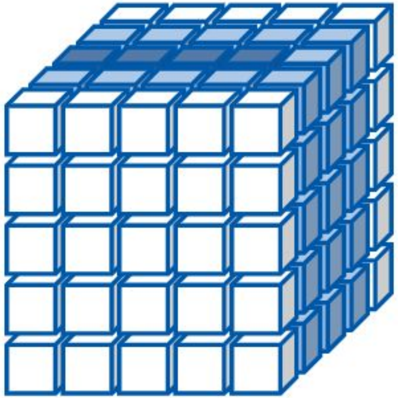
Solution:
Only store the three input planes needed by the output plane at a time

Solution:
Assign enough threads for loading one input plane and processing one output plane



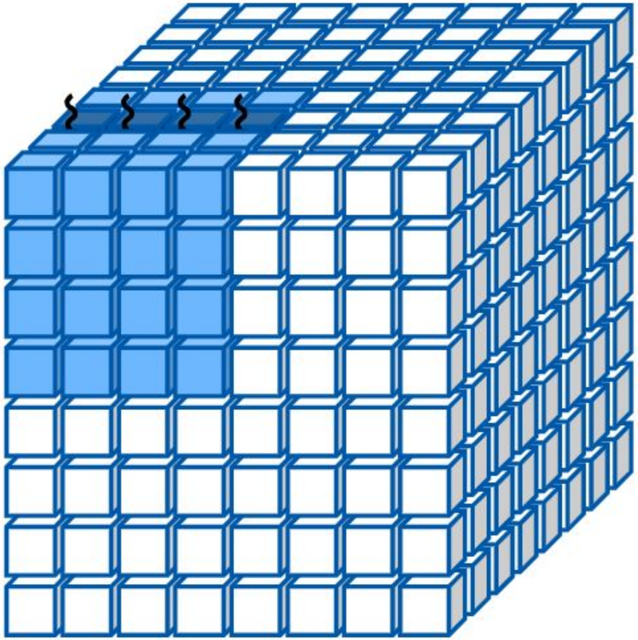
output

Coarsening and Slicing

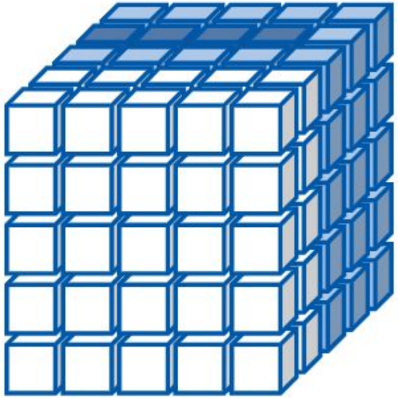


Solution:
Only store the three input planes needed by the output plane at a time

Solution:
Assign enough threads for loading one input plane and processing one output plane



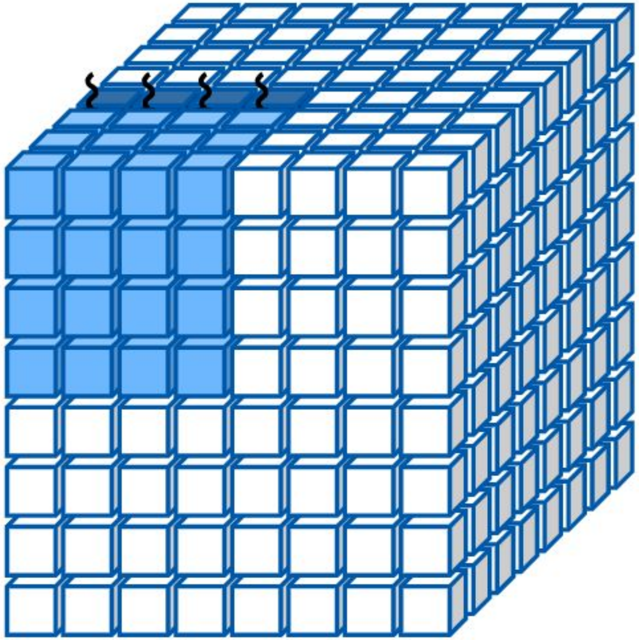
Coarsening and Slicing



input$_{tile}$
(in shared memory)

Solution:
Only store the three input planes needed by the output plane at a time

Solution:
Assign enough threads for loading one input plane and processing one output plane



output

Register Tiling

- Save shared memory by putting the next slice elements in registers
  - Moving a slice into shared memory when it becomes the current slice
  - Then, moving them back to registers when they become the previous slice
  - We only need enough shared memory for one slice
- The AI remains the same as in the previous example
- Tradeoff between shared memory and registers
  - Three more registers are used in this implementation
  - But only one slice needs to be in shared memory
- For tile size T, register plus shared memory tiling only requires $3*T$ registers and T shared memory locations as compared to T^3 shared memory locations in shared-memory-only tiling
  - This can improve occupancy

Thank you for your attention!

Gianmarco Accordi
gianmarco.accordi@polimi.it