# Stencil operations in CUDA

Tobias Brandvik
Whittle Laboratory
University of Cambridge

# Motivation

We want to simulate the flow of air through turbomachines

# Overview

1. Basic stencil operations in CUDA
2. Optimisation strategies
3. A software framework for stencil applications
4. Case study: A turbomachinery flow solver

# A simple stencil

Second derivative:

$$\frac{\partial^2 u}{\partial x^2}$$

Simple Fortran implementation:

```
DO K=2,NK-1
  DO J=2,NJ-1
    DO I=2,NI-1
      D2UDX2(I,J,K) = (U(I+1,J,K) - 2.0*U(I,J,K) +
     &           U(I-1,J,K))/(DX*DX)
    END DO
  END DO
END DO
```

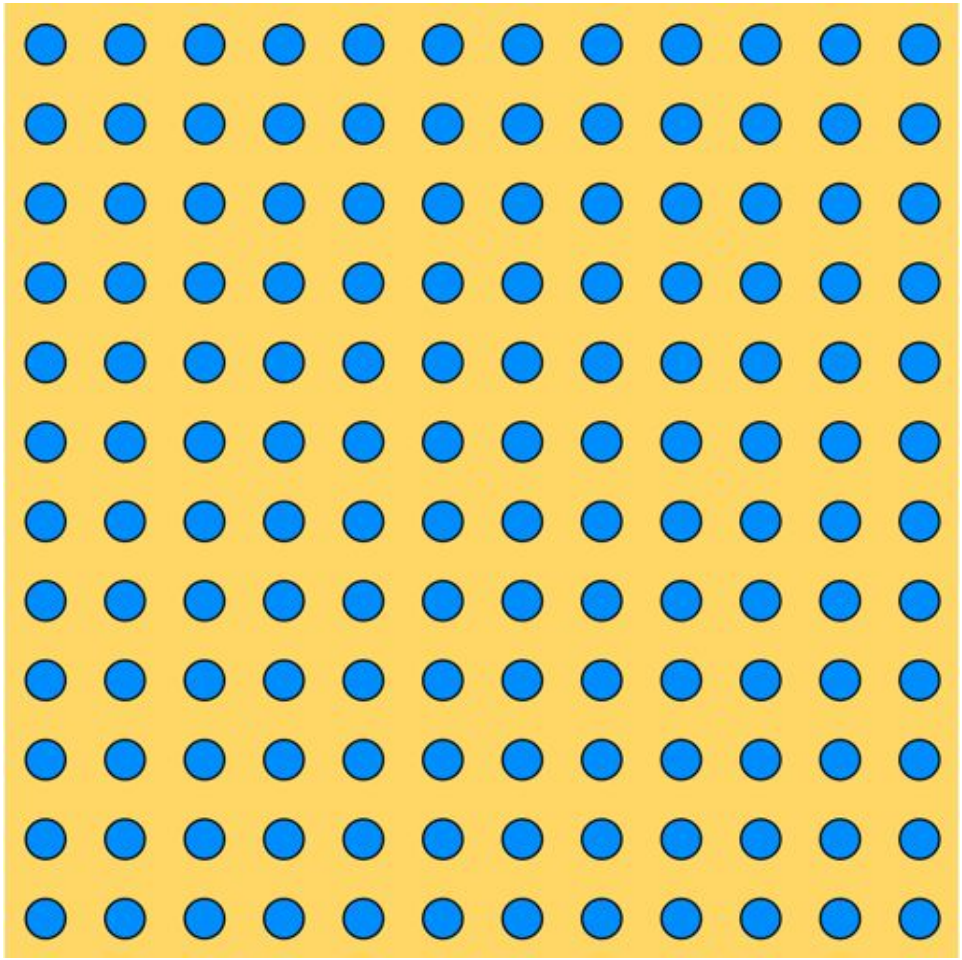
# Structured grid

- Stencil should be evaluated for every grid node in a large 3D grid (e.g. 100^3)

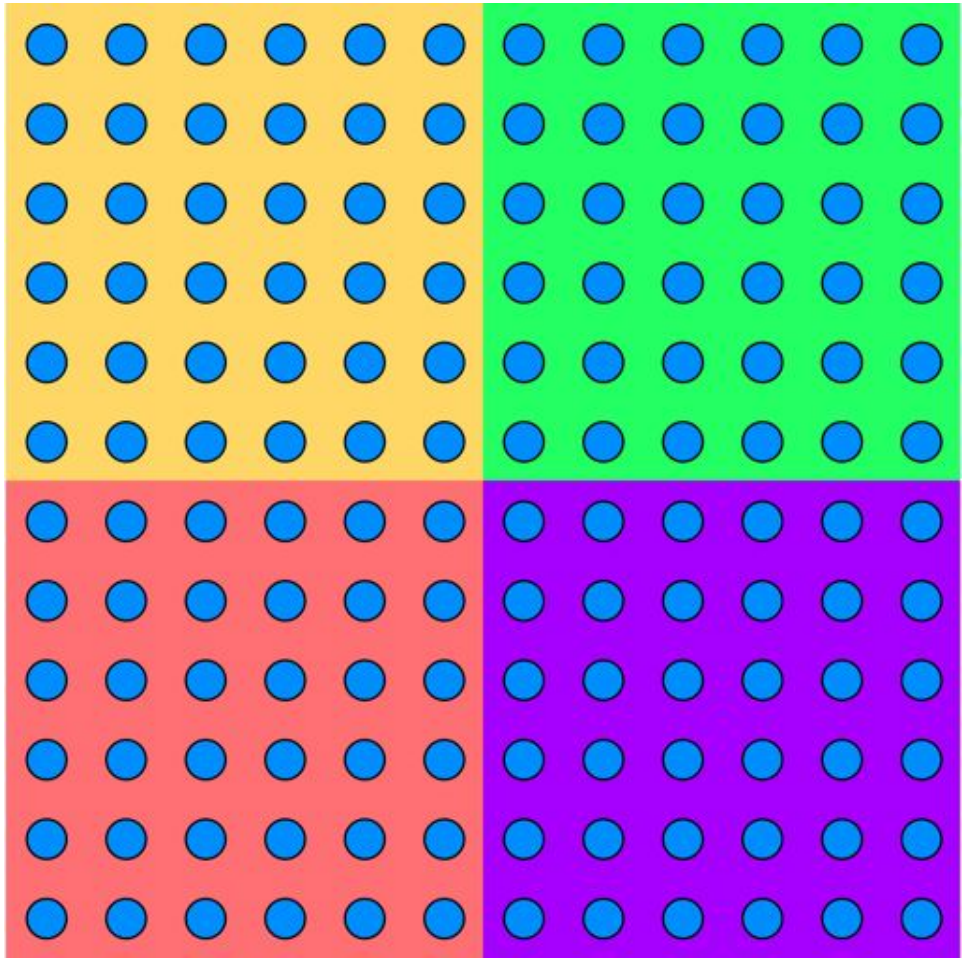
# CUDA thread block parallelism

- The grid is divided into “sub-blocks” that are assigned to a CUDA thread block

# Grid is divided into sub-blocks



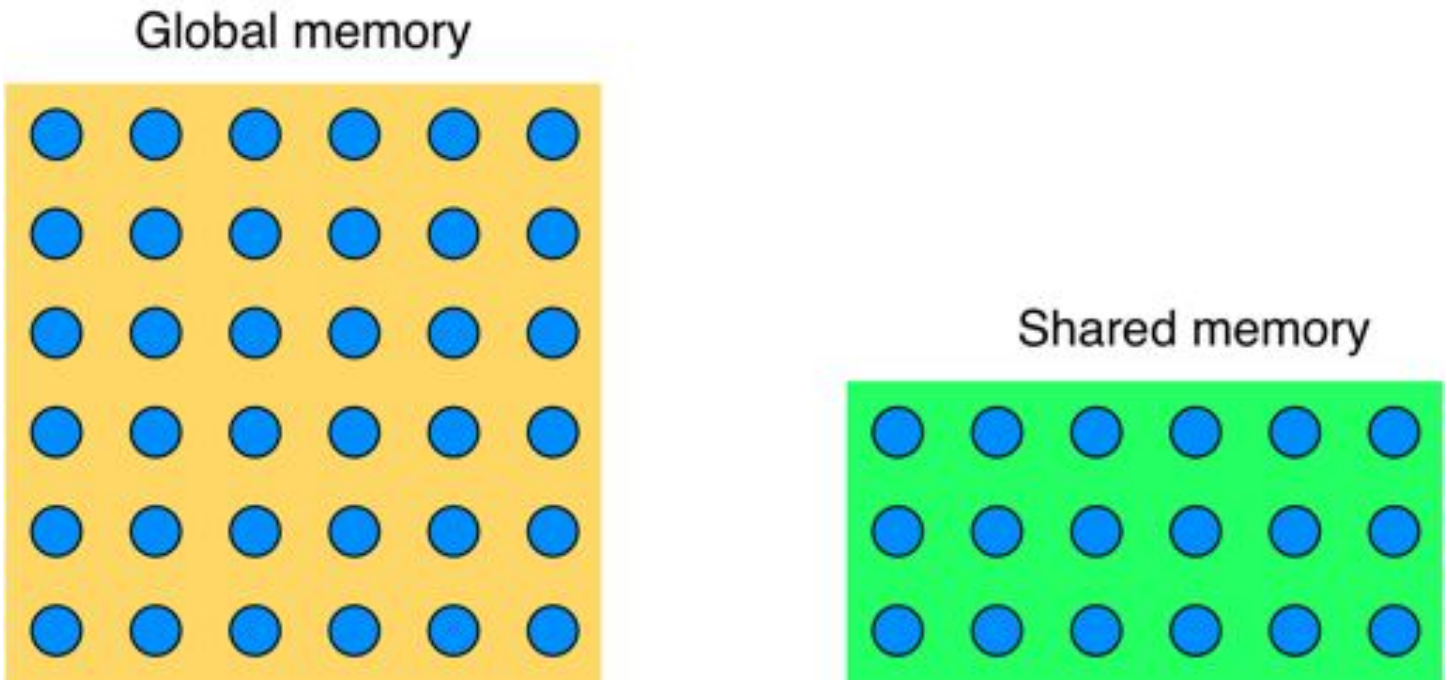
# Grid is divided into sub-blocks



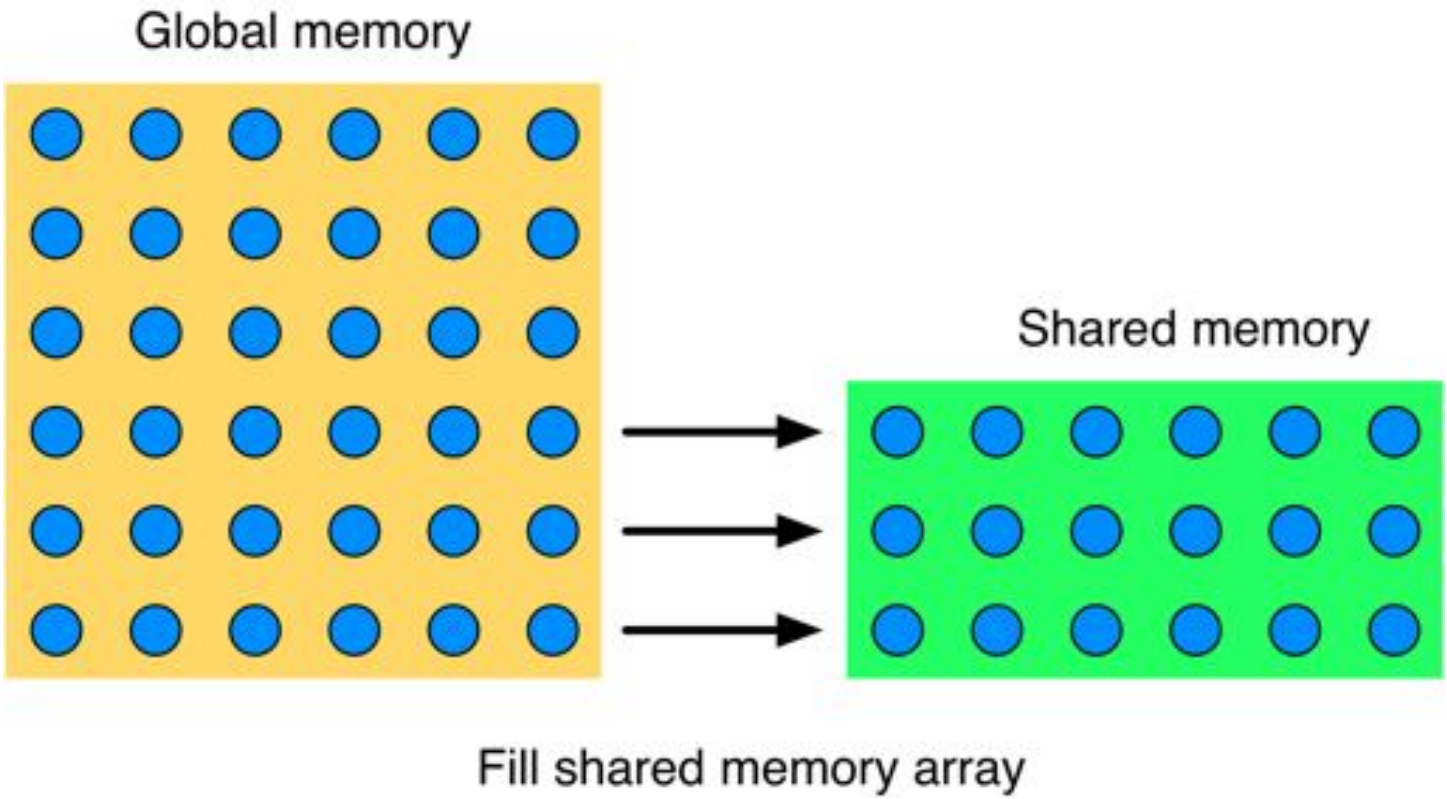
# Sub-block implementation (after Datta et al.)

- Each thread in a thread block reads sub-block data from global device memory to SM shared memory (coalesced reads for maximum bandwidth)
- Synch threads
- Update nodes in sub-block using shared memory and output result back to global memory
- But shared memory and max threads per block are limited, so best plan is to march through sub-domain plane-by-plane…

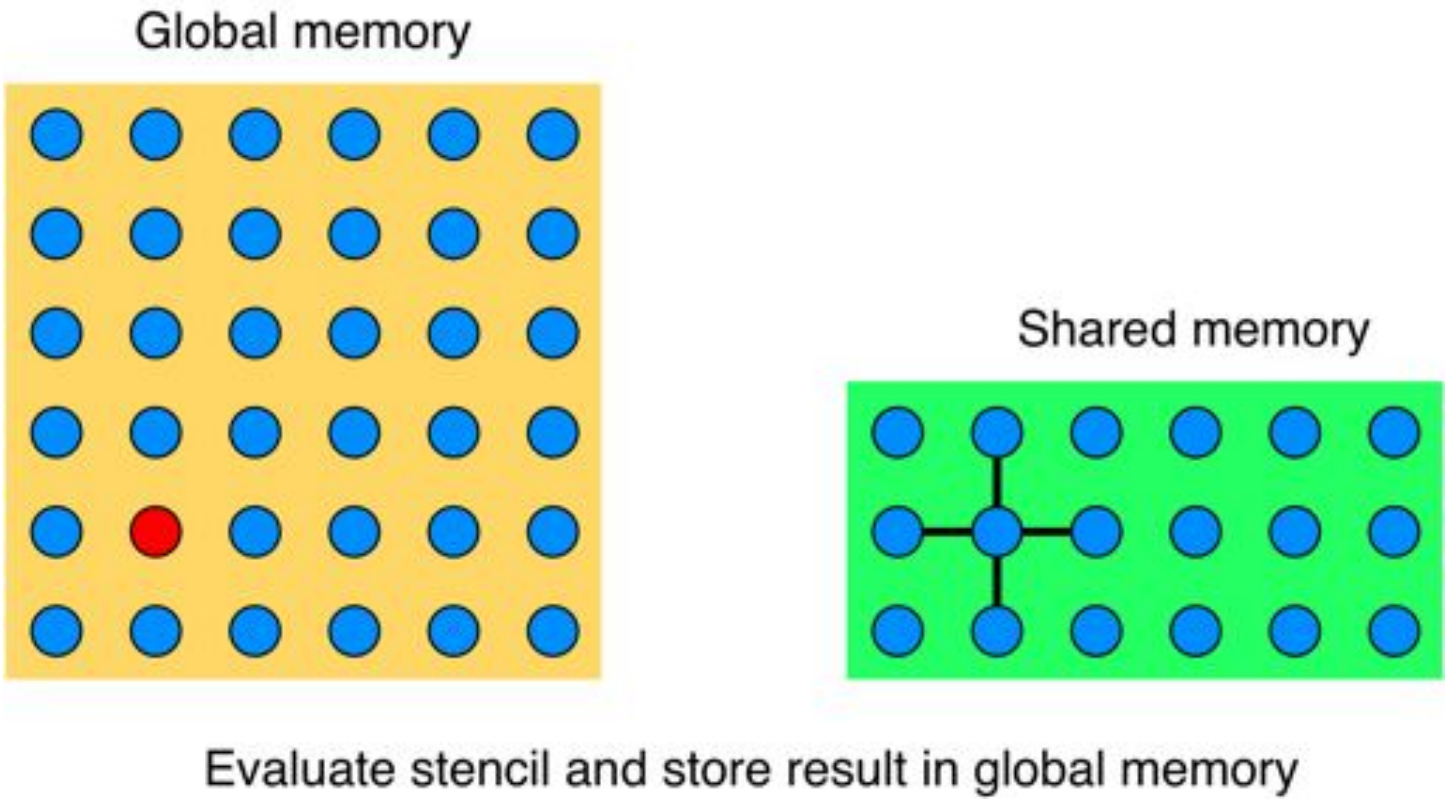
# Sub-block implementation



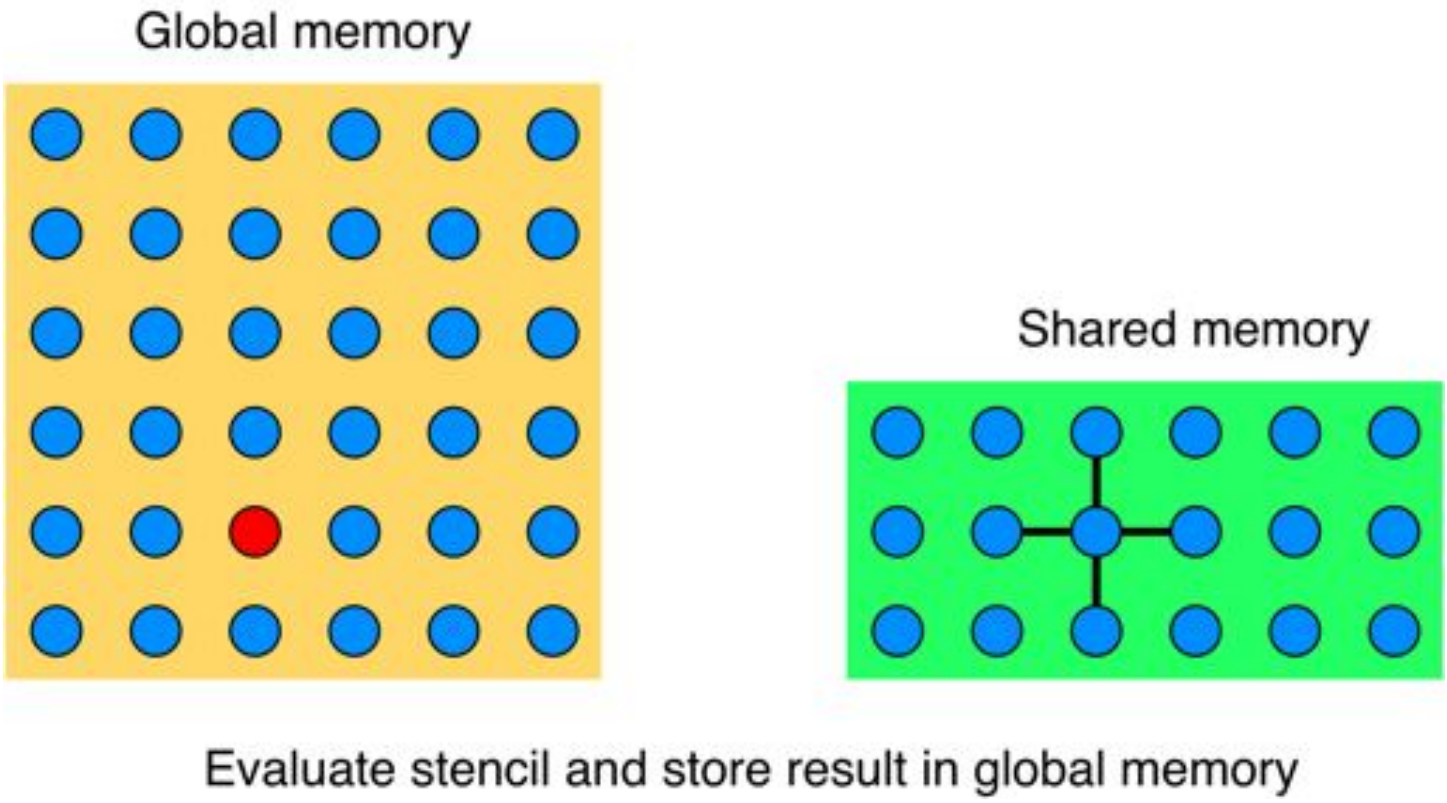
# Sub-block implementation



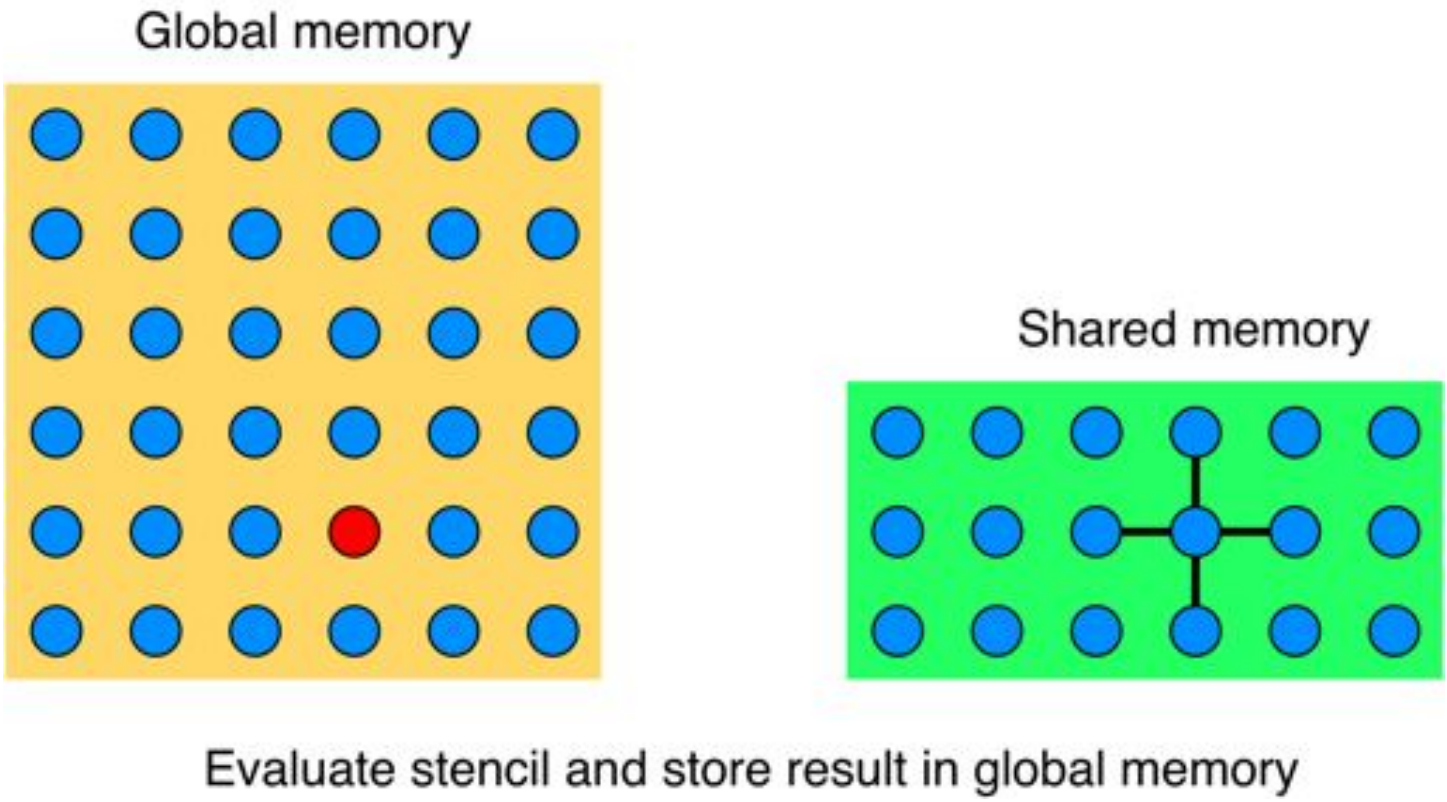
# Sub-block implementation



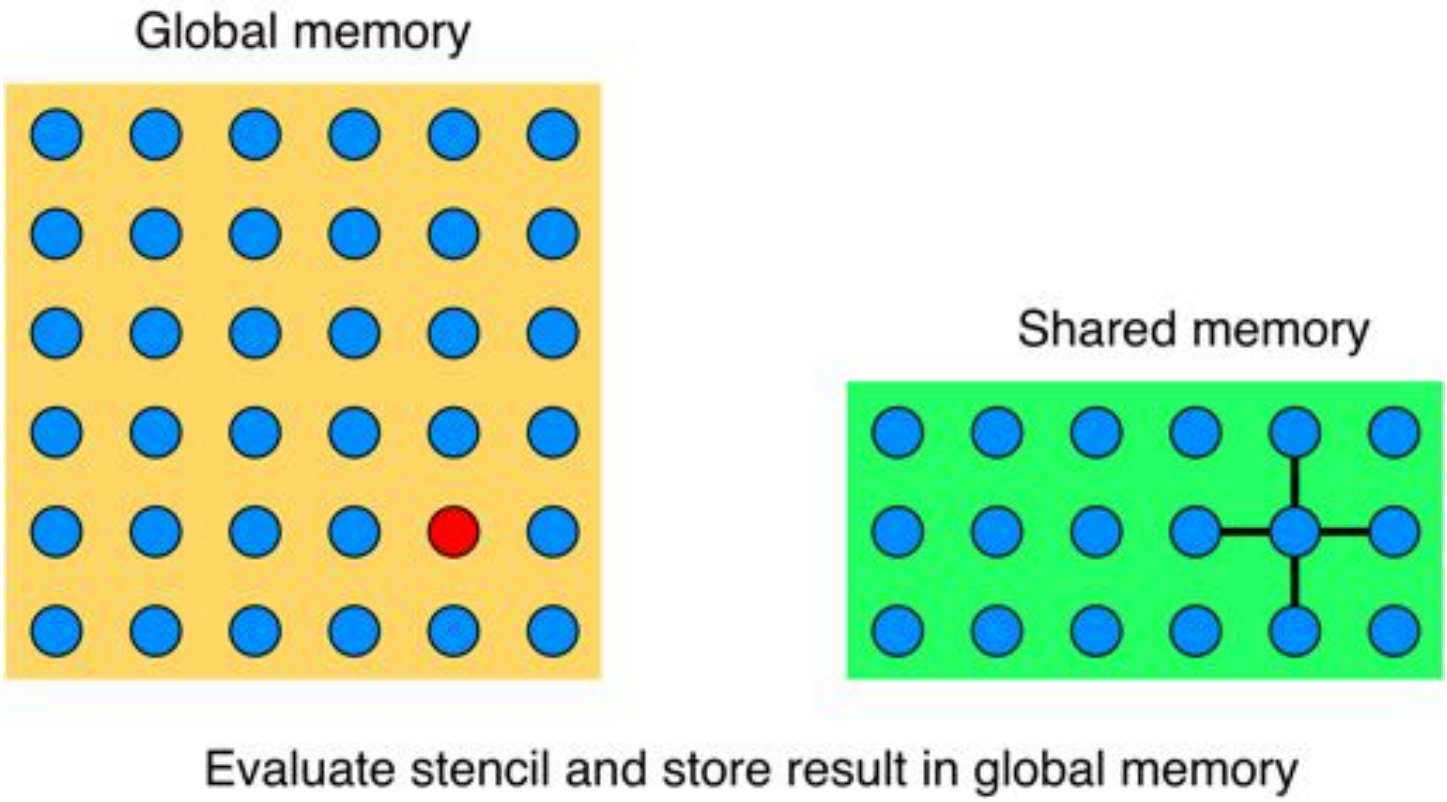
# Sub-block implementation



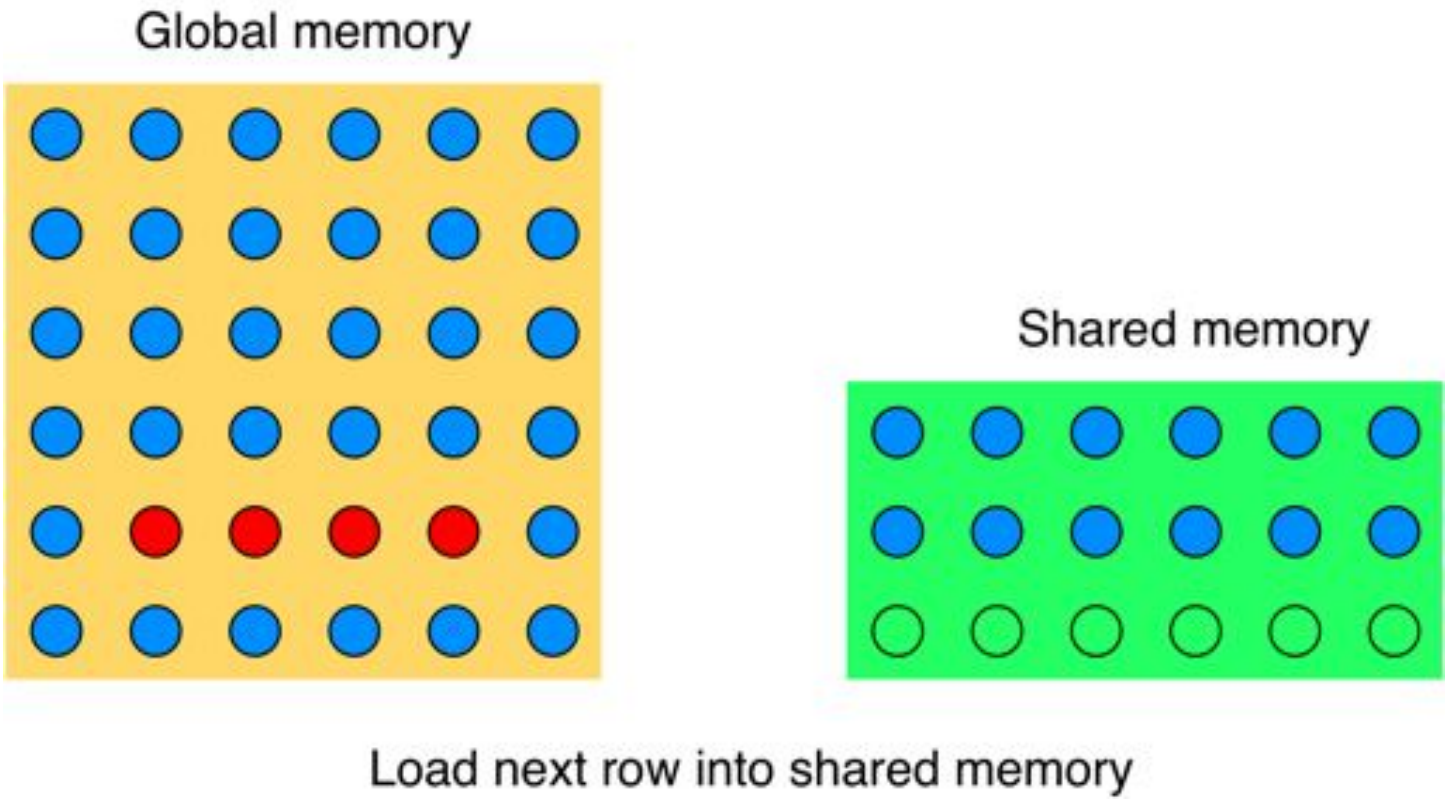
# Sub-block implementation



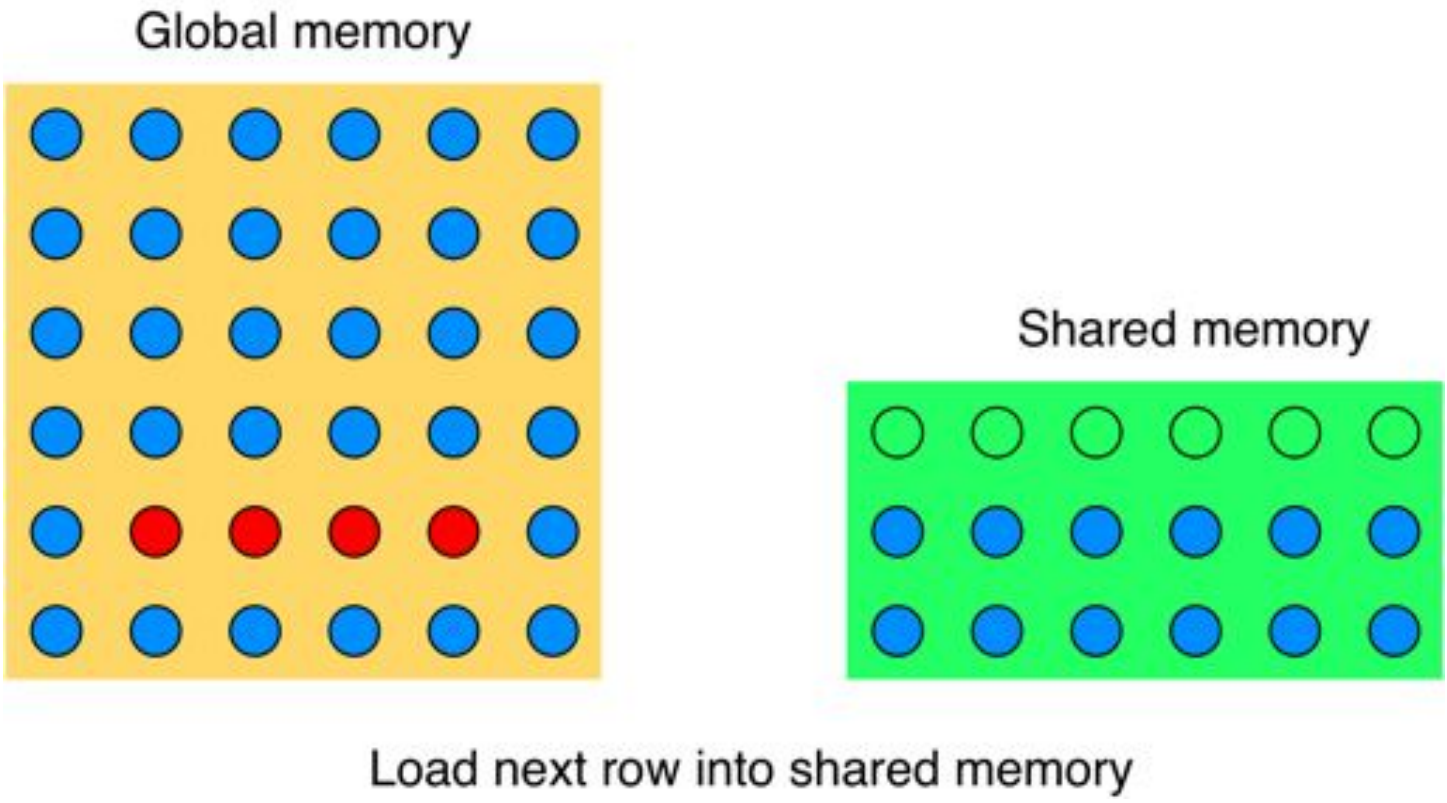
# Sub-block implementation



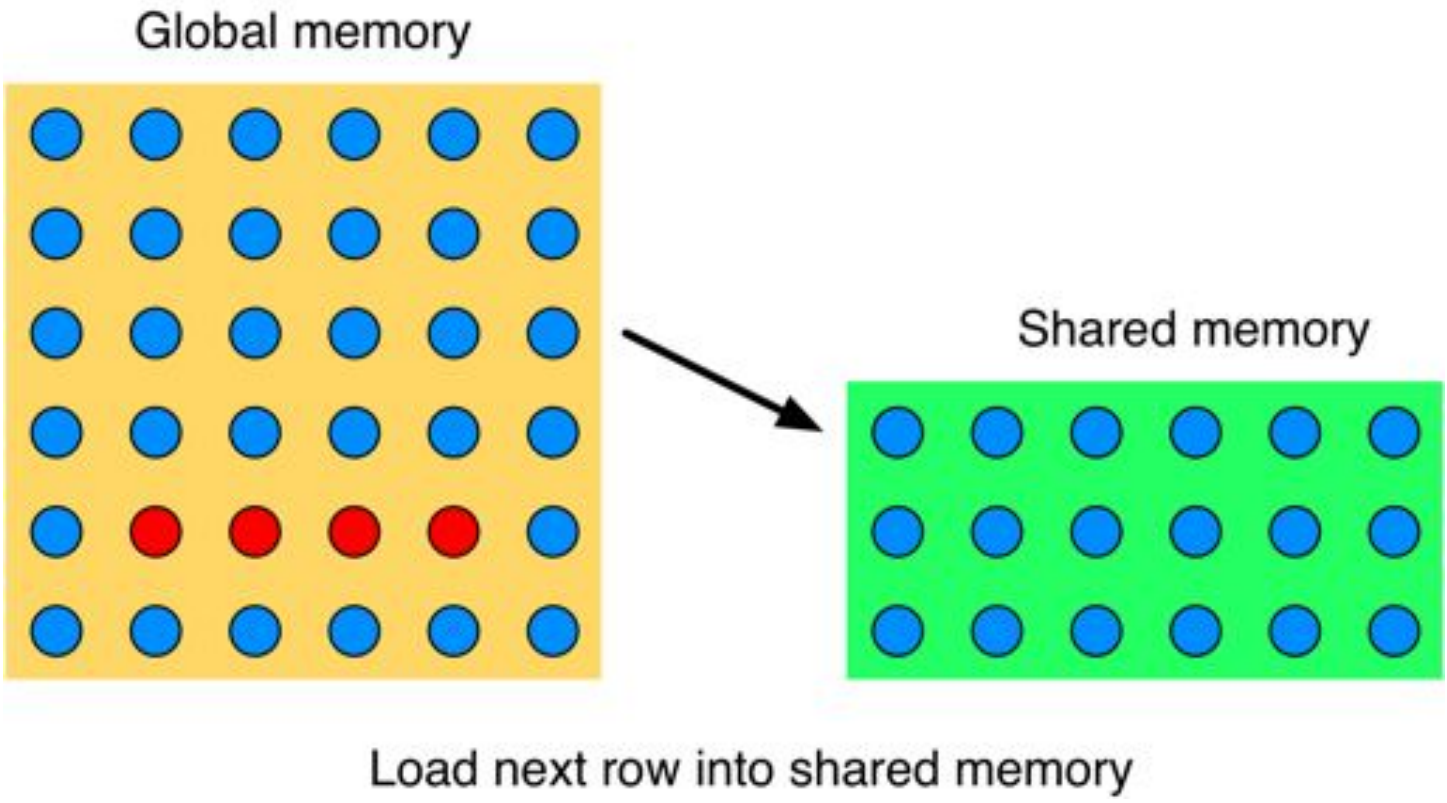
# Sub-block implementation



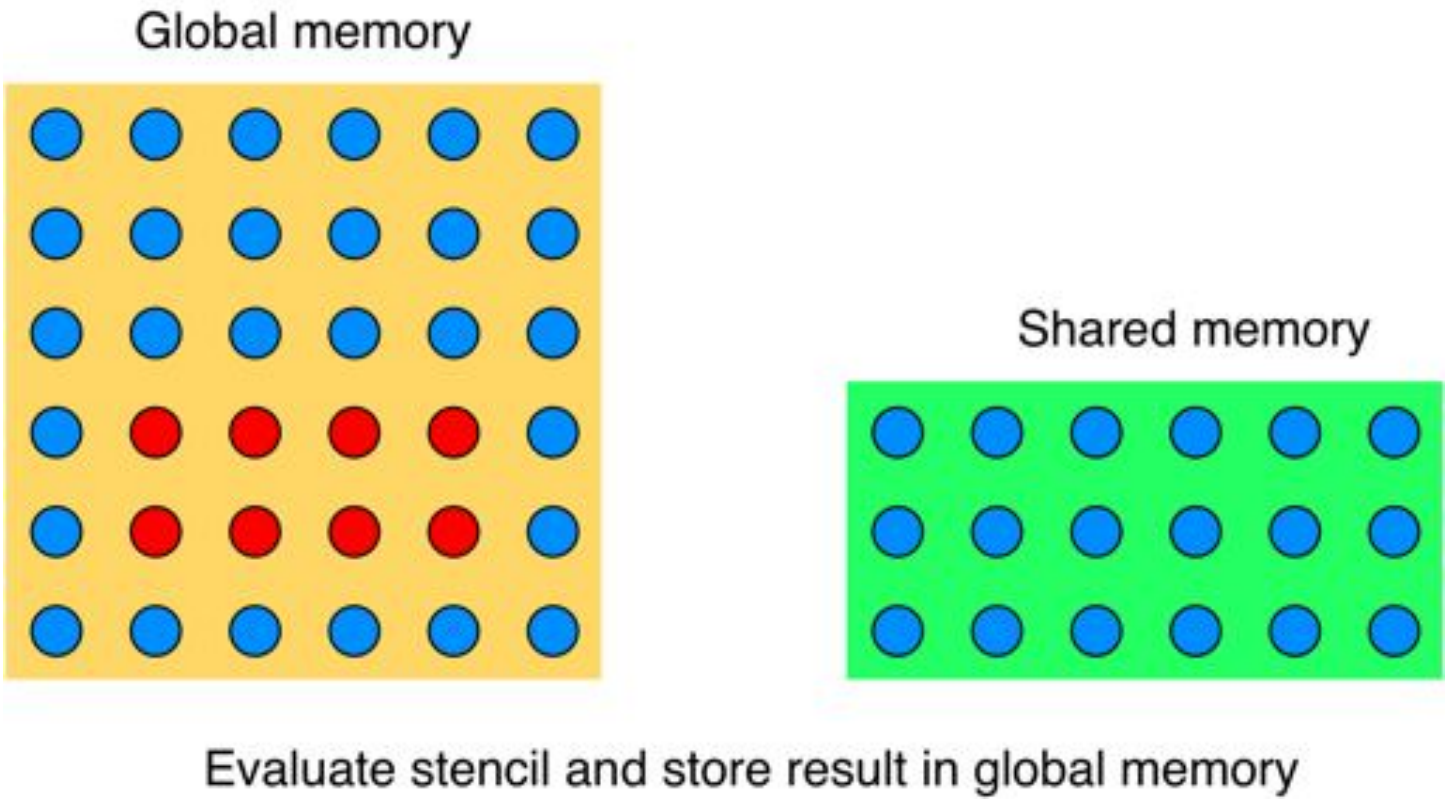
# Sub-block implementation



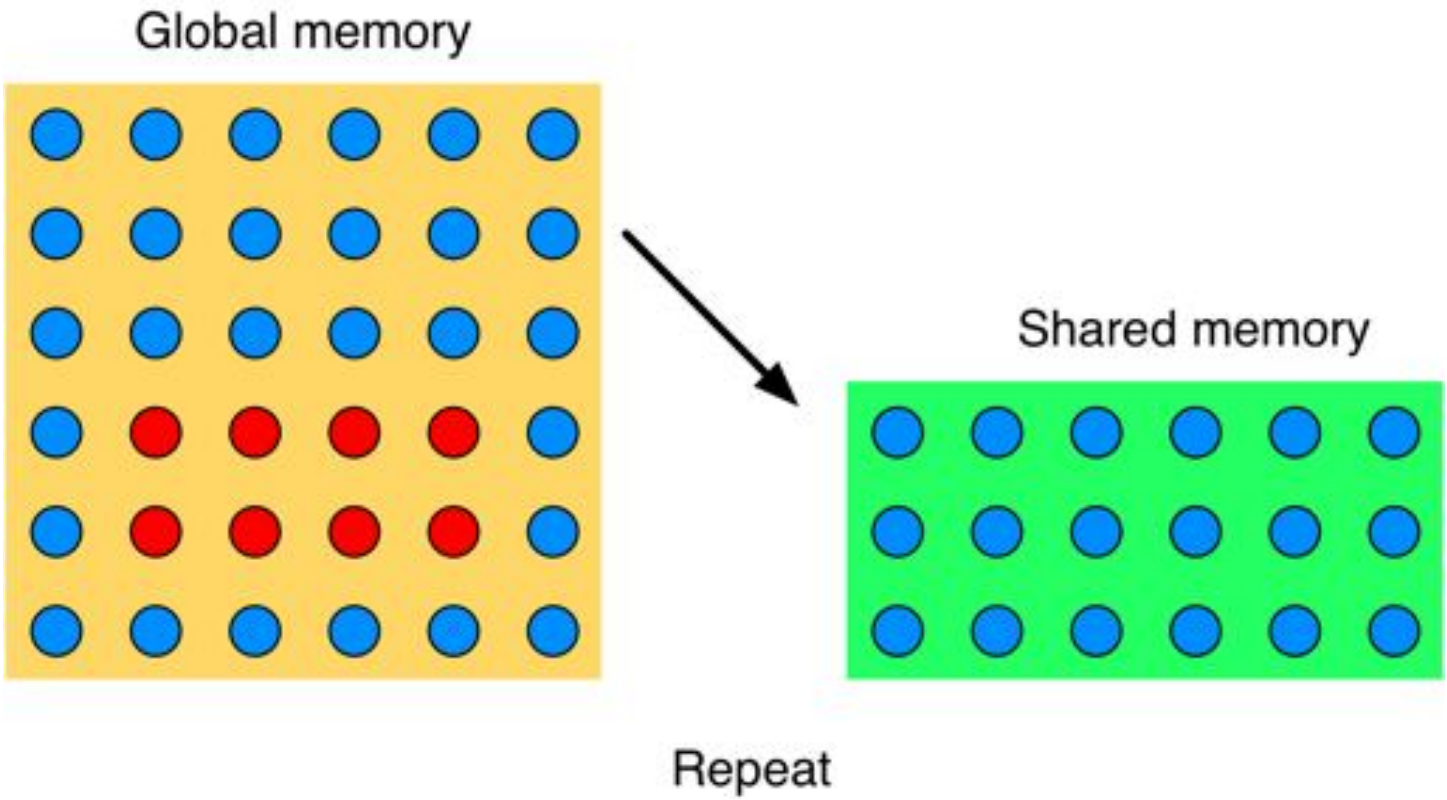
# Sub-block implementation



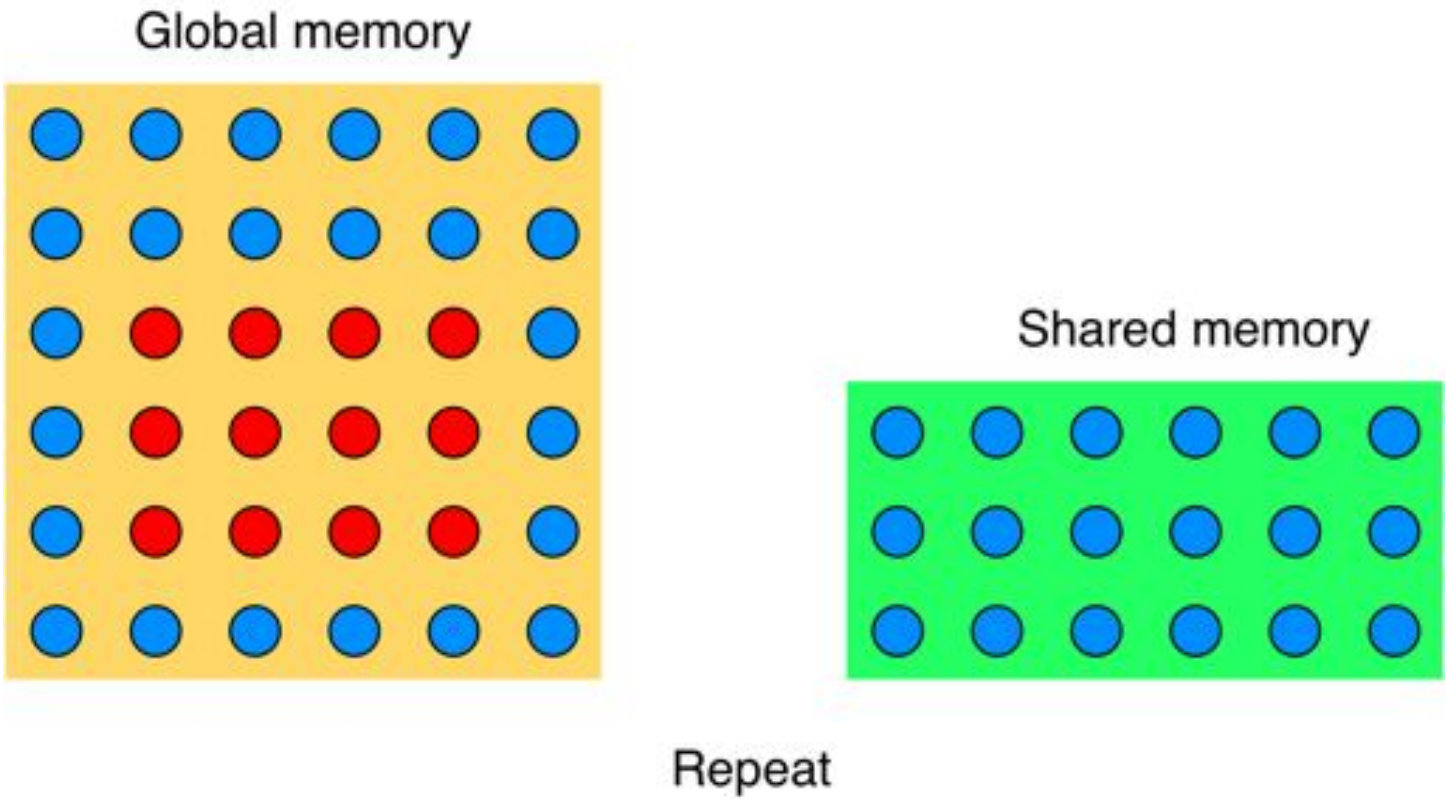
# Sub-block implementation



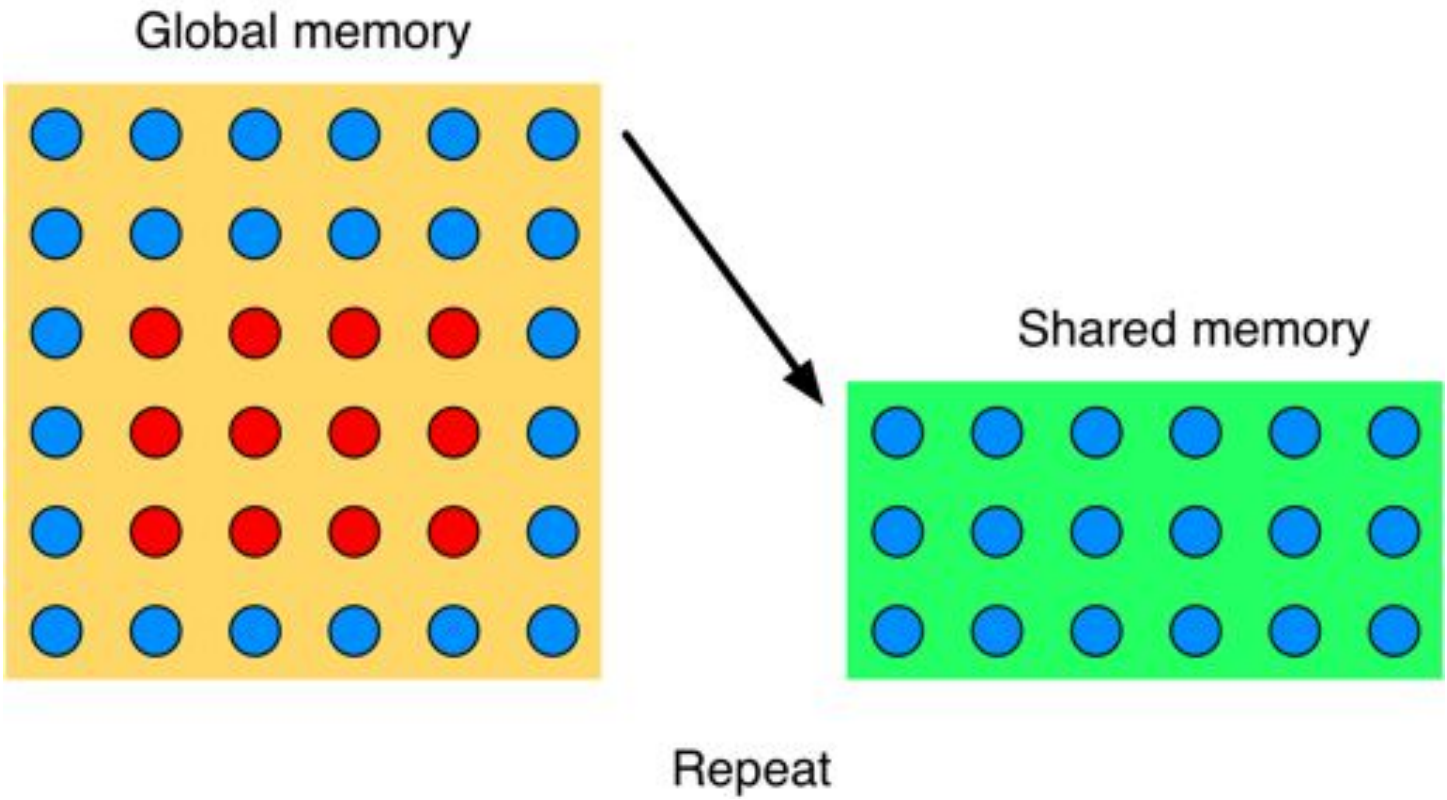
# Sub-block implementation



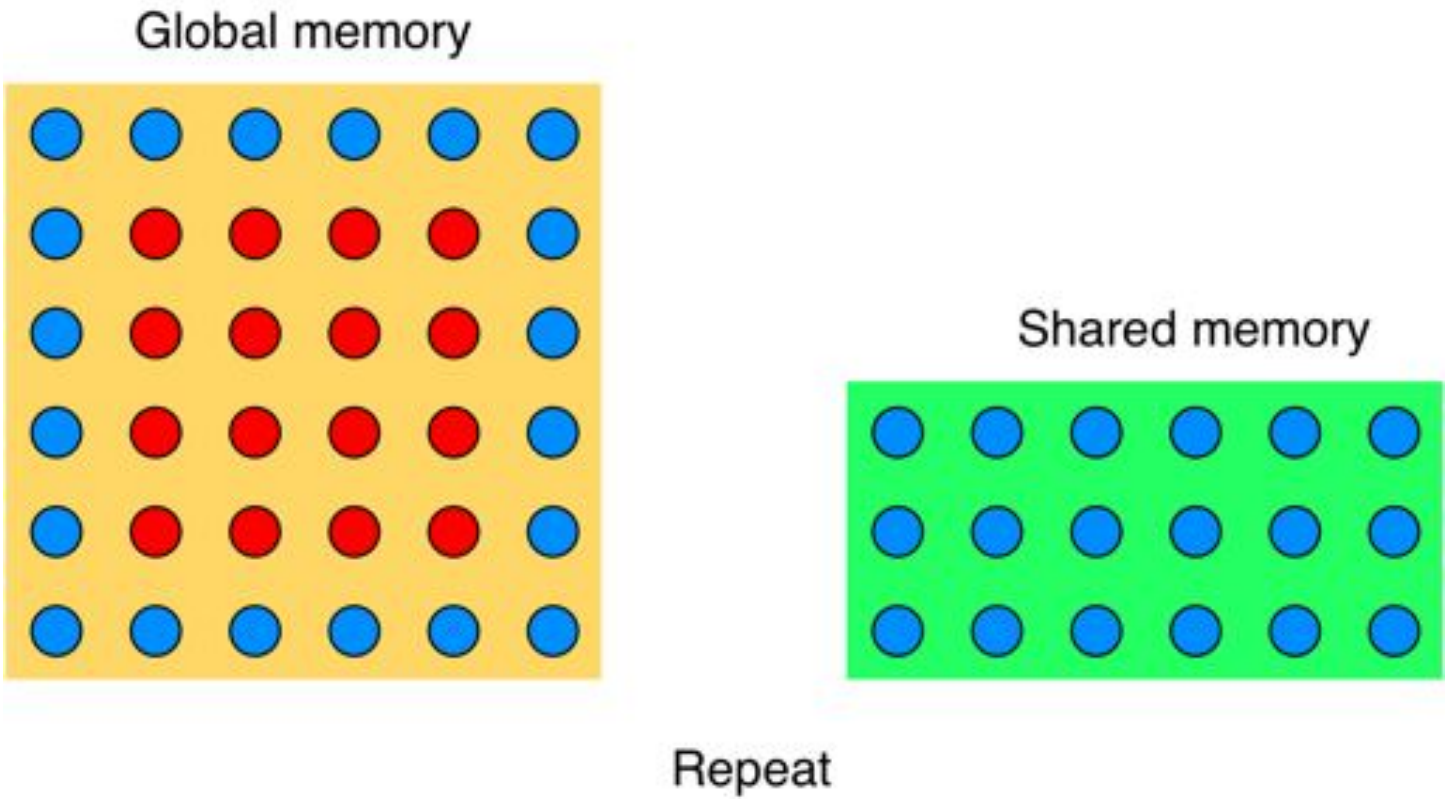
# Sub-block implementation



# Sub-block implementation



# Sub-block implementation



# CUDA Example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
```

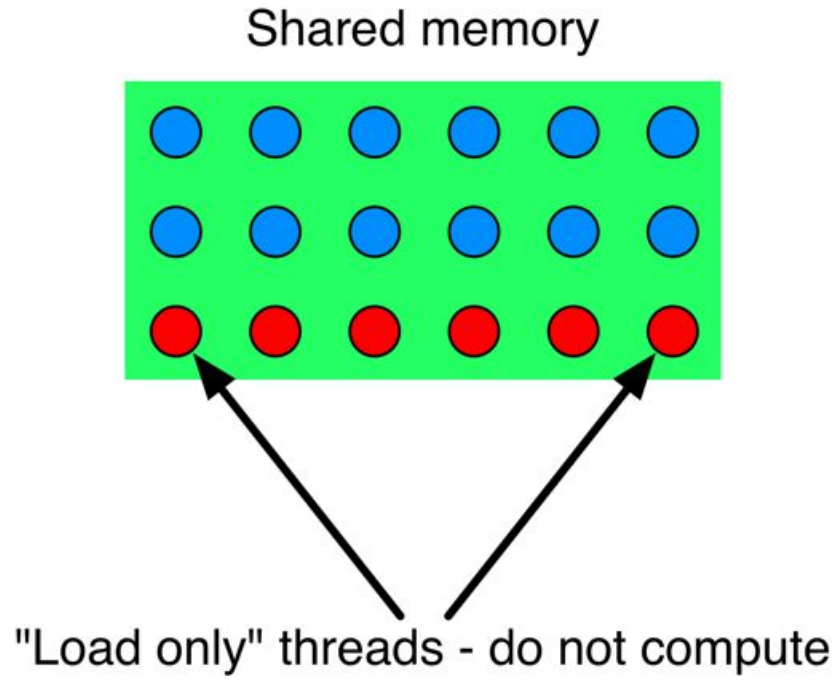
# CUDA Example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
a[i][j][0] = a_d[i0m10];       // fetch first three planes
a[i][j][1] = a_d[i000];
a[i][j][2] = a_d[i0p10];
__syncthreads();               // make sure planes are loaded
```

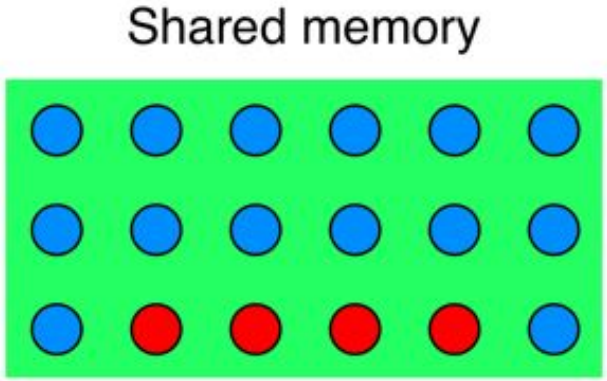
# CUDA Example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
a[i][j][0] = a_d[i0m10];       // fetch first three planes
a[i][j][1] = a_d[i000];
a[i][j][2] = a_d[i0p10];
__syncthreads();               // make sure planes are loaded
// compute the stencil: //
b_d[i000]  = sf1*a[i][j][1] +
           + sfd6*(a[i-1][j][1] + a[i+1][j][1]
                 + a[i][j][0]   + a[i][j][2]
                 + a[i][j-1][1] + a[i][j+1][1])
// load next "k" plane and repeat //
```

# Optimisaton #1: No load-only threads

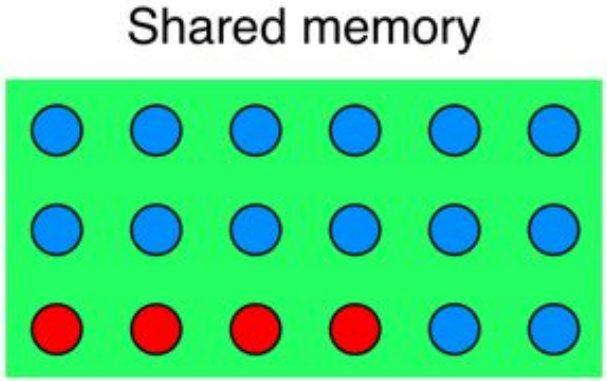


# Optimisaton #1: No load-only threads




Have only compute threads - two load operations

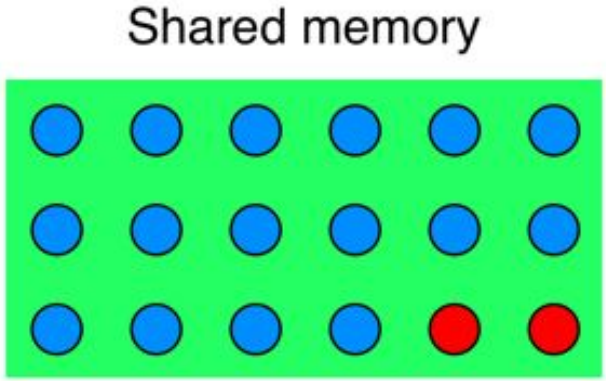
# Optimisaton #1: No load-only threads




First load - all threads read from memory

Have only compute threads - two load operations

# Optimisaton #1: No load-only threads




Second load - only two threads read from memory

Have only compute threads - two load operations

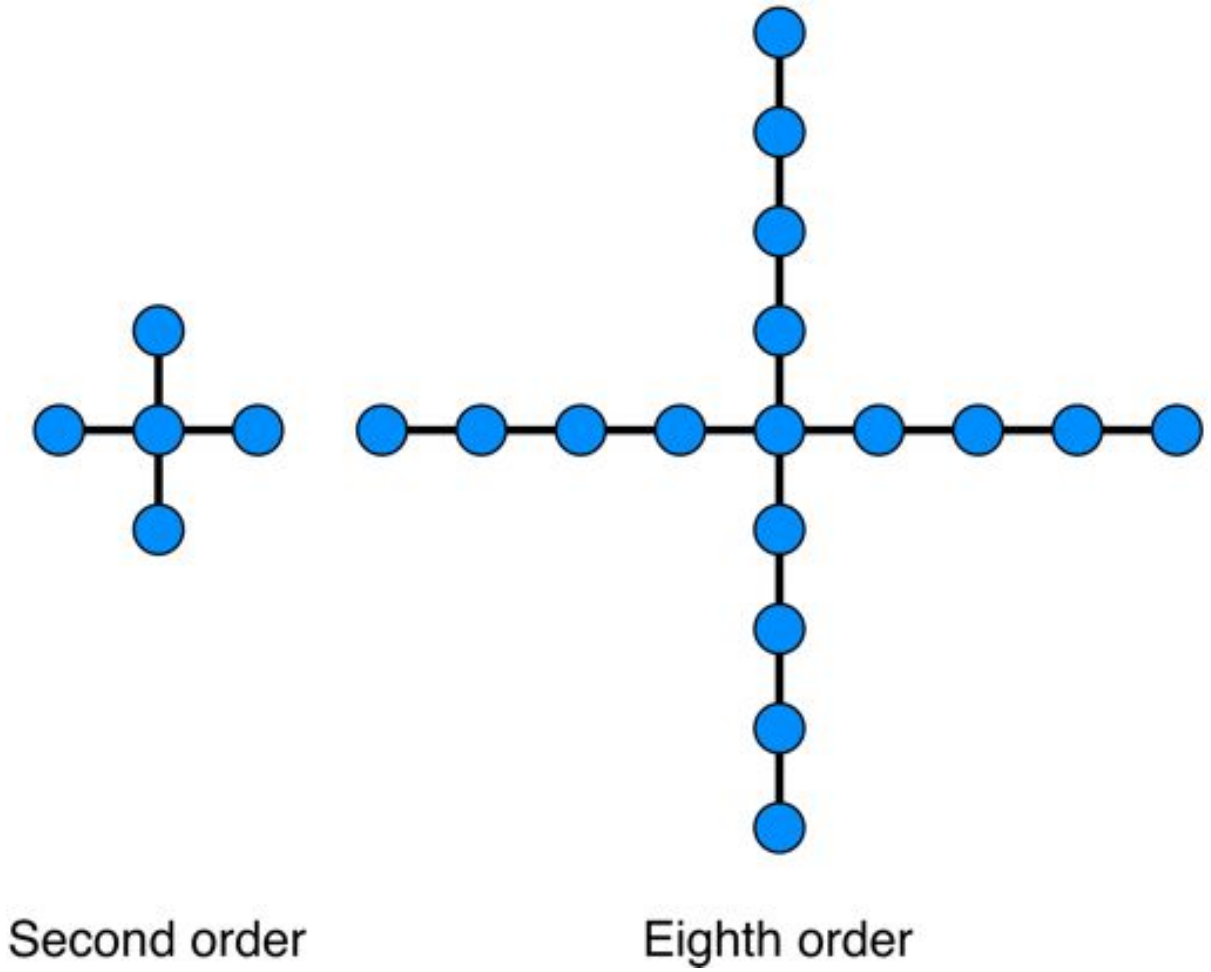
# Optimisation #2: Use all on-chip memory

- Real applications often have large stencils
- Inputs: A, B, C, D, E
- Outputs: Y1, Y2, Y3
- Stencils: f1, f2, f3
- Y1 = f1(A, B, C) + f2(D, E)
- Y2 = Y1 + F3(B, C, D)
- Y3 = Y1 + Y2 + F4(A, E)

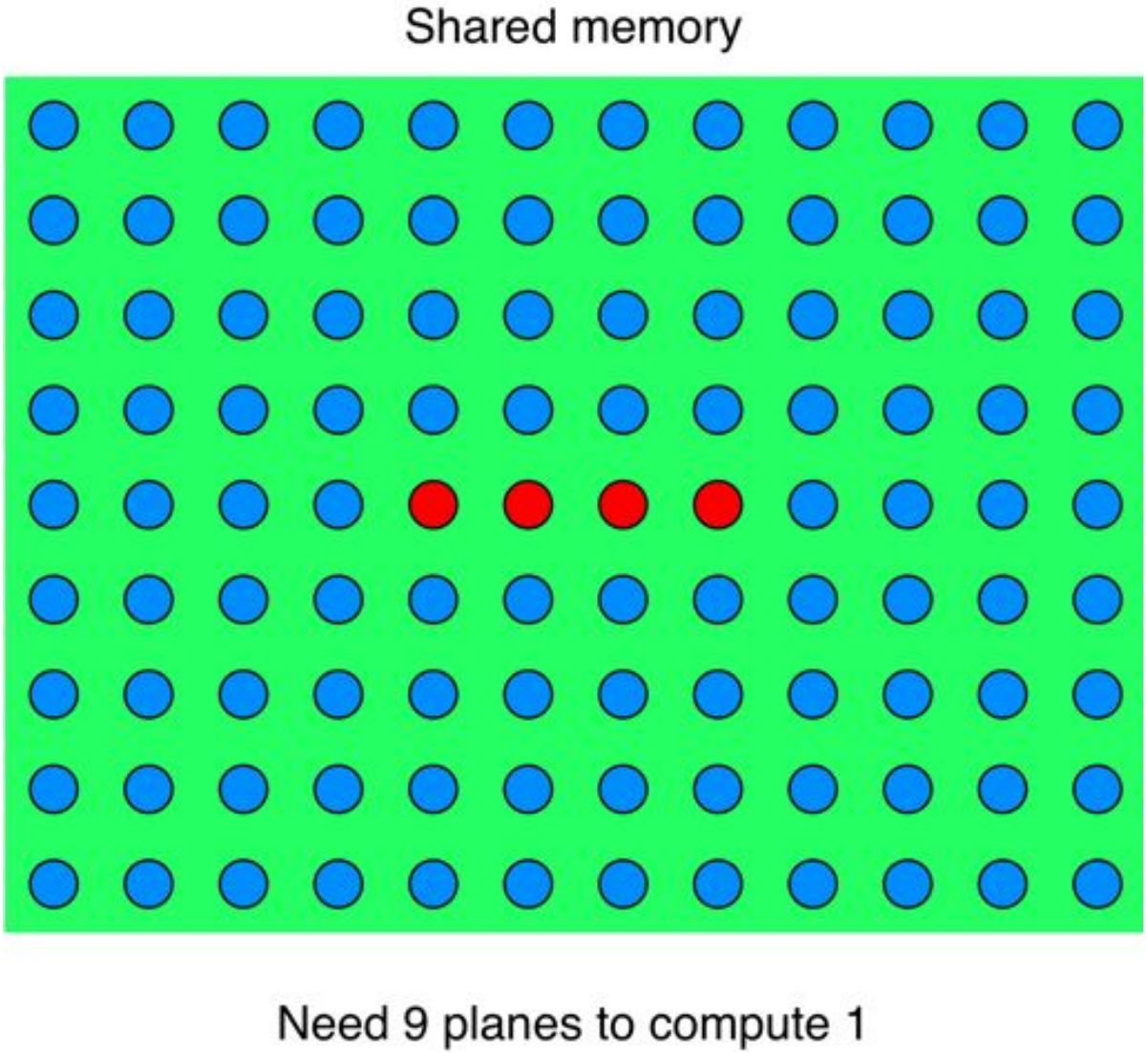
# Optimisation #2: Use all on-chip memory

- Many arrays lead to large pressure on shared memory
- Sub-blocks become small
- Halo nodes become a large fraction of total nodes in sub-block
- Place some inputs in texture cache (GT200) or normal cache (Fermi)

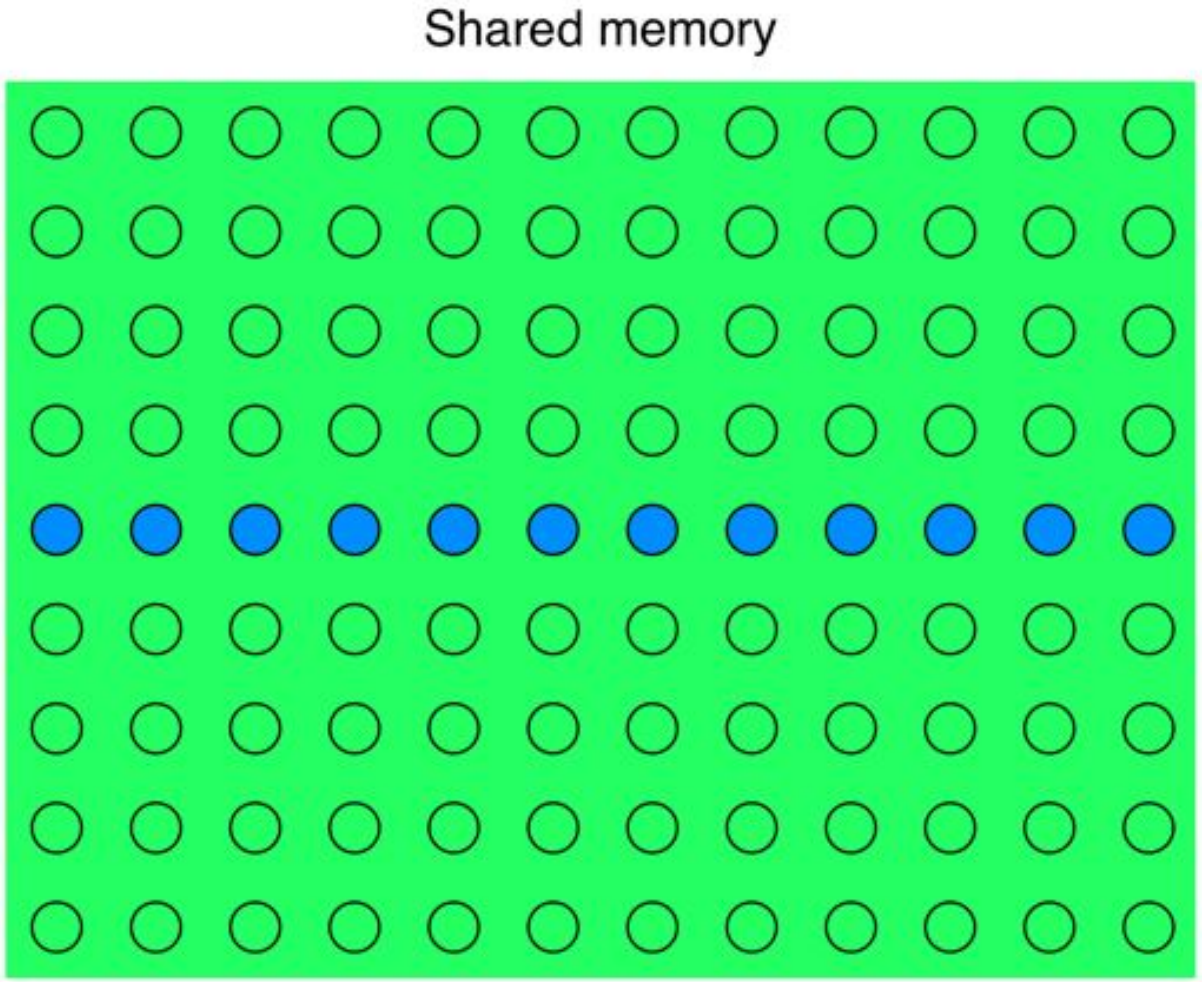
# Optimisation #3: High-order stencils



# Optimisation #3: High-order stencils

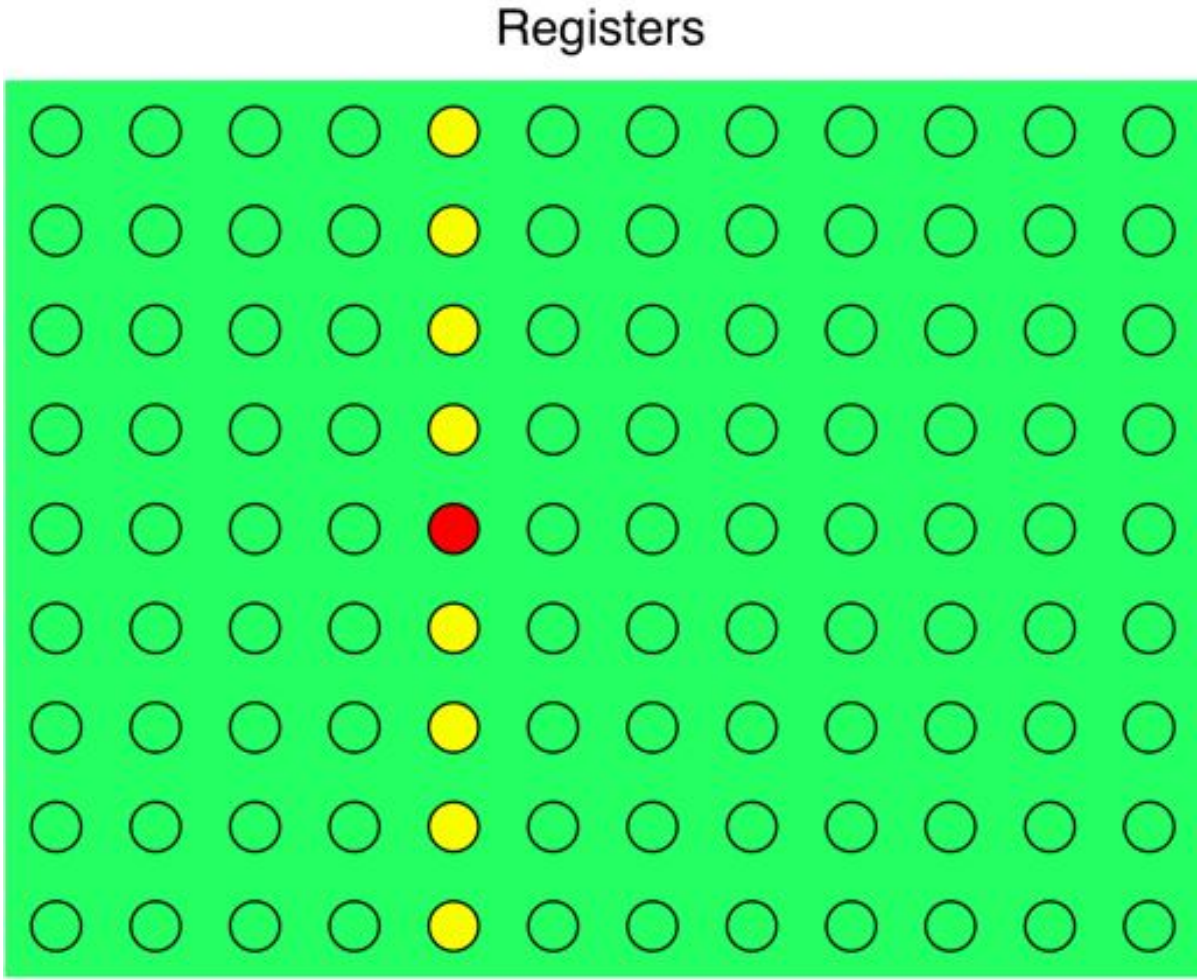


# Optimisation #3: High-order stencils



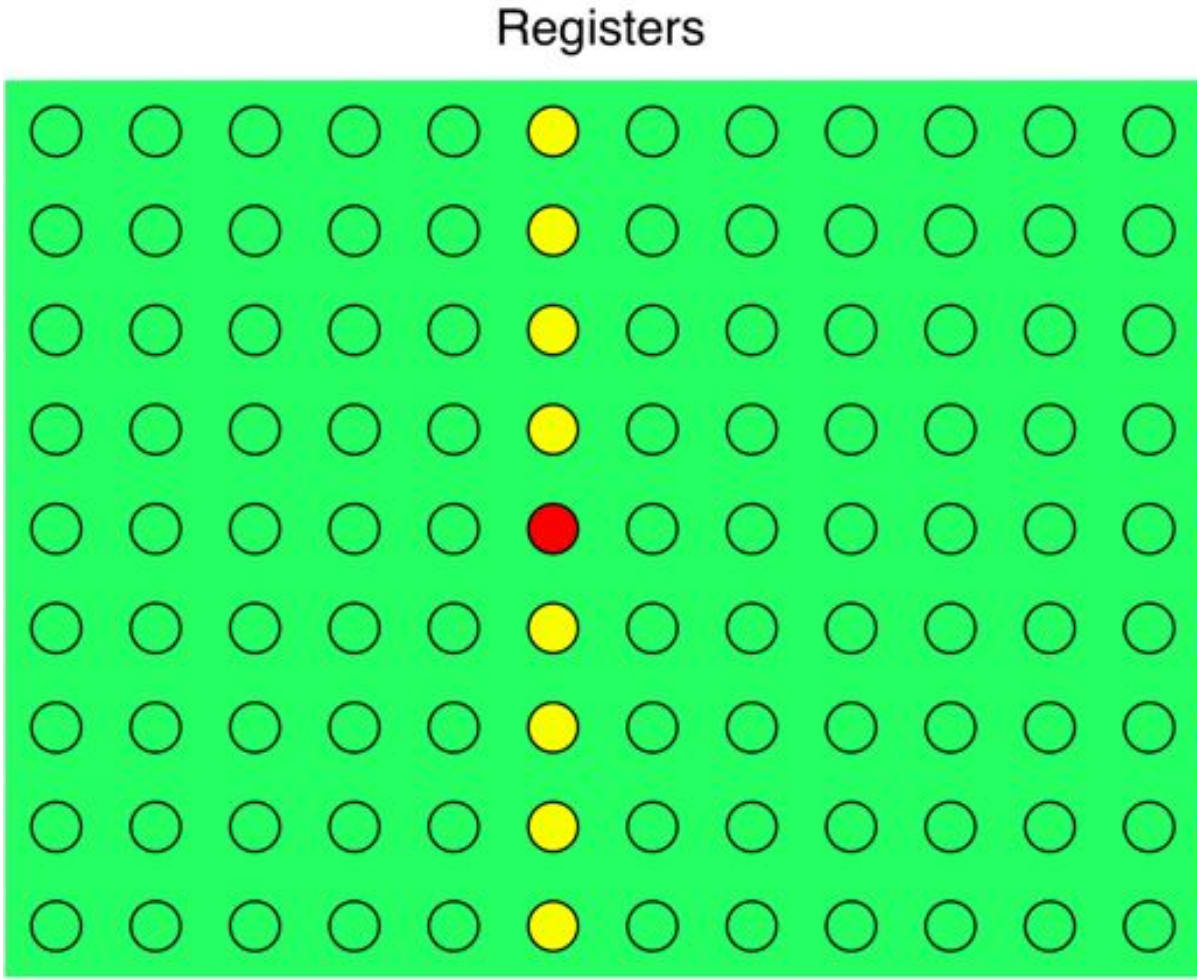

Only the grid nodes in one plane need to be shared to evaluate the stencil - store these in shared memory

# Optimisation #3: High-order stencils



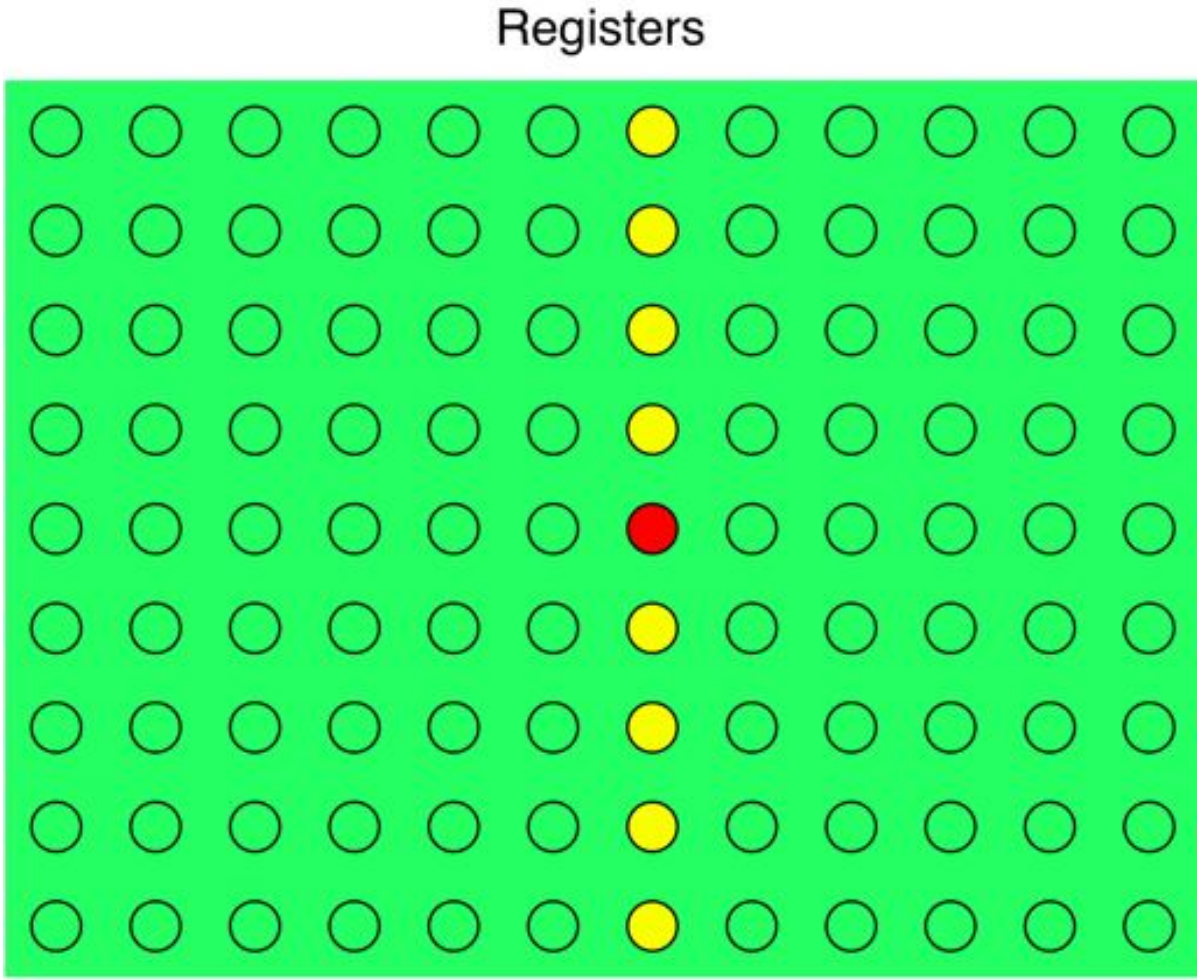

The other grid nodes can be stored in registers for each thread

# Optimisation #3: High-order stencils



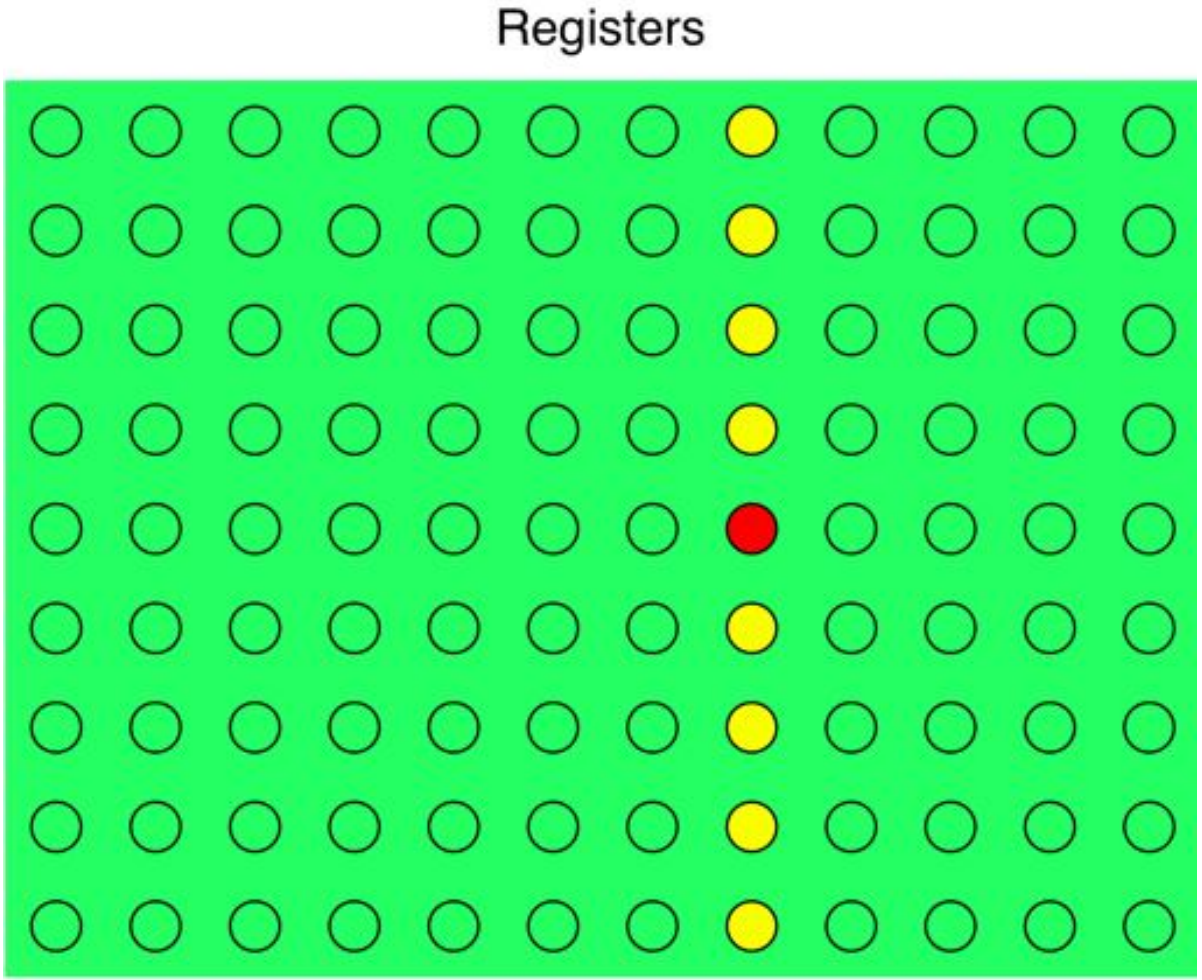

The other grid nodes can be stored in registers for each thread

# Optimisation #3: High-order stencils



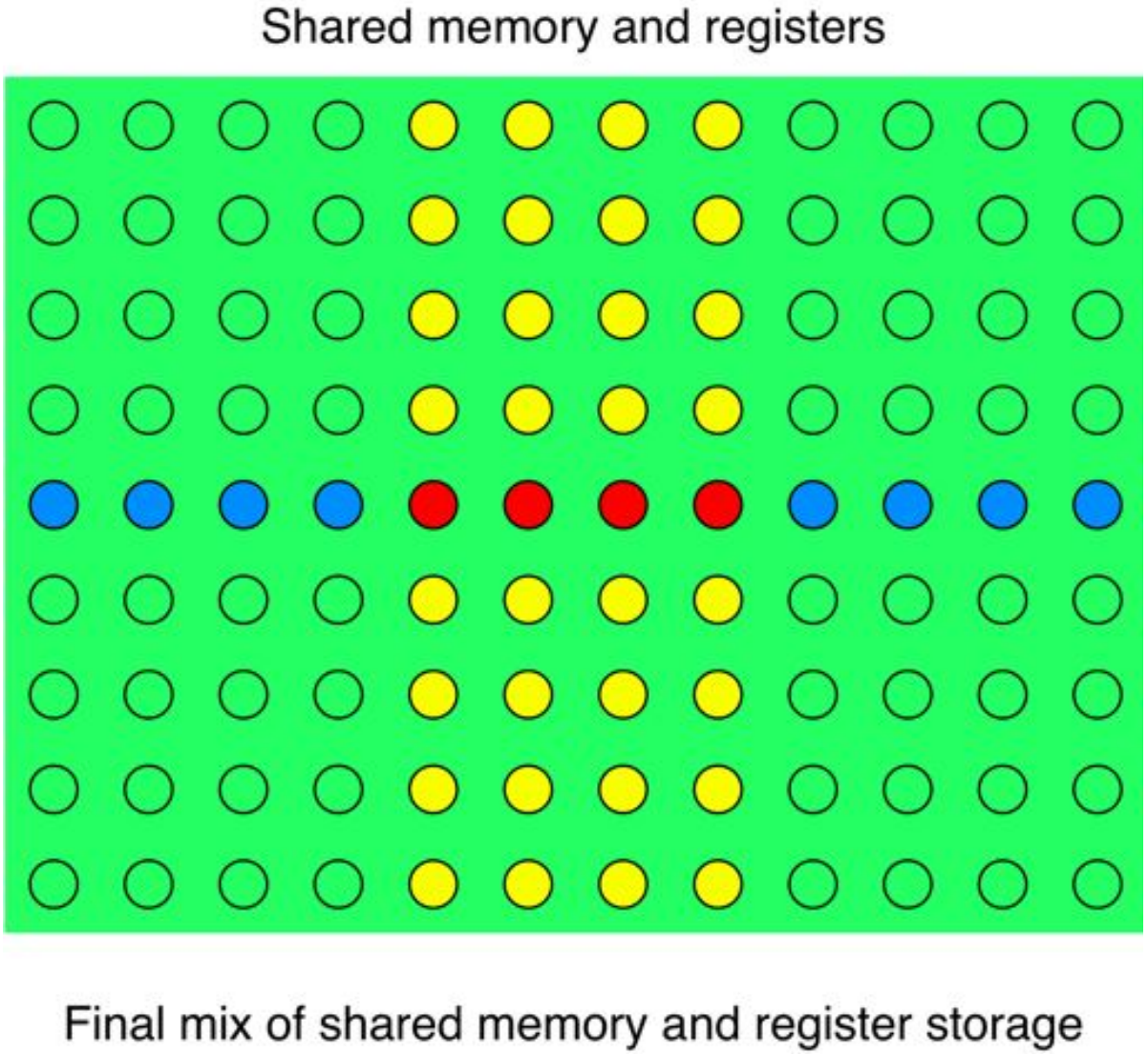

The other grid nodes can be stored in registers for each thread

# Optimisation #3: High-order stencils

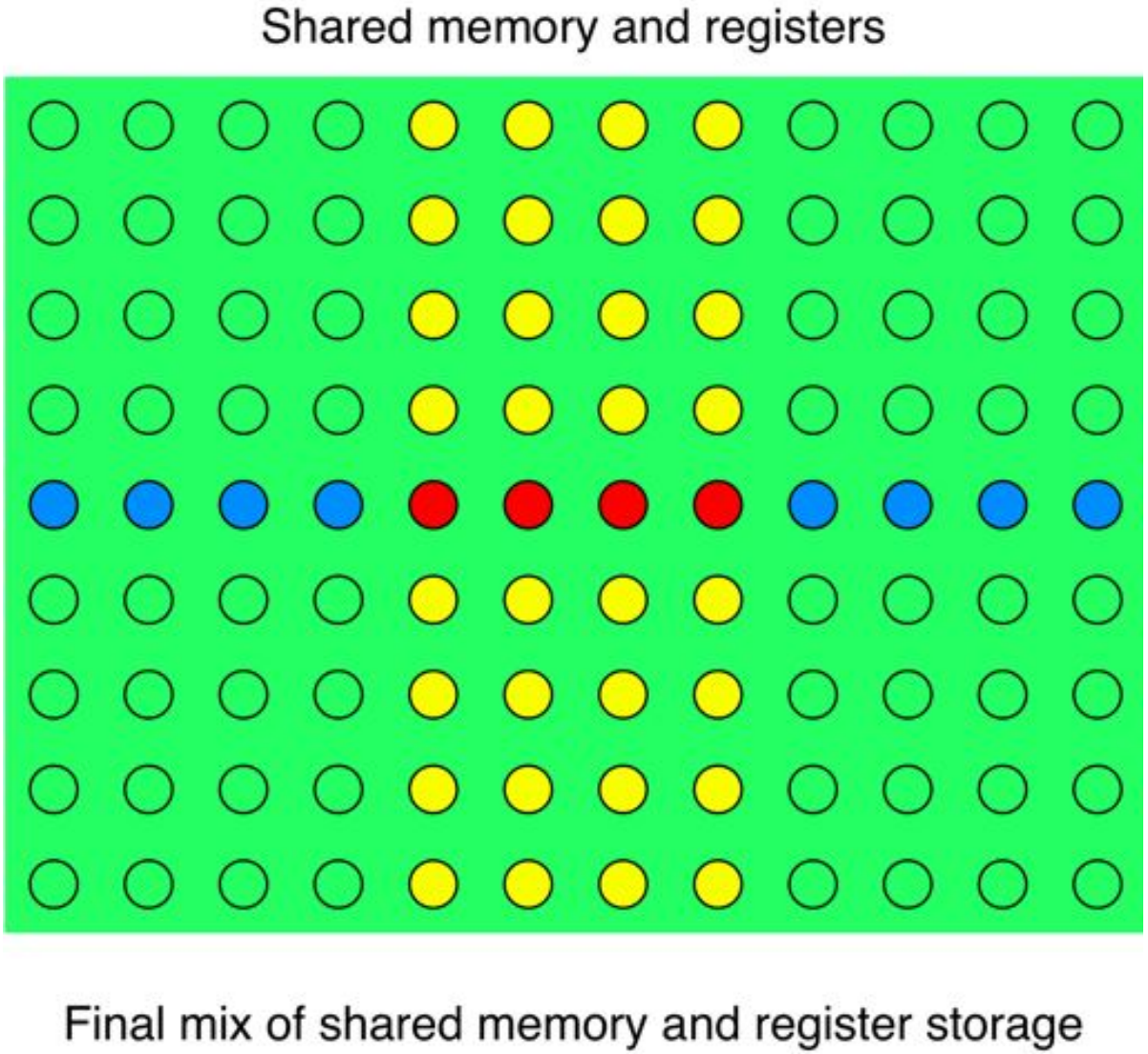



The other grid nodes can be stored in registers for each thread

# Optimisation #3: High-order stencils



# Optimisation #3: High-order stencils



# Real stencil applications

- Real stencil applications:
  - Many complicated stencils
  - Should run on any processor
  - Should scale across multiple processors

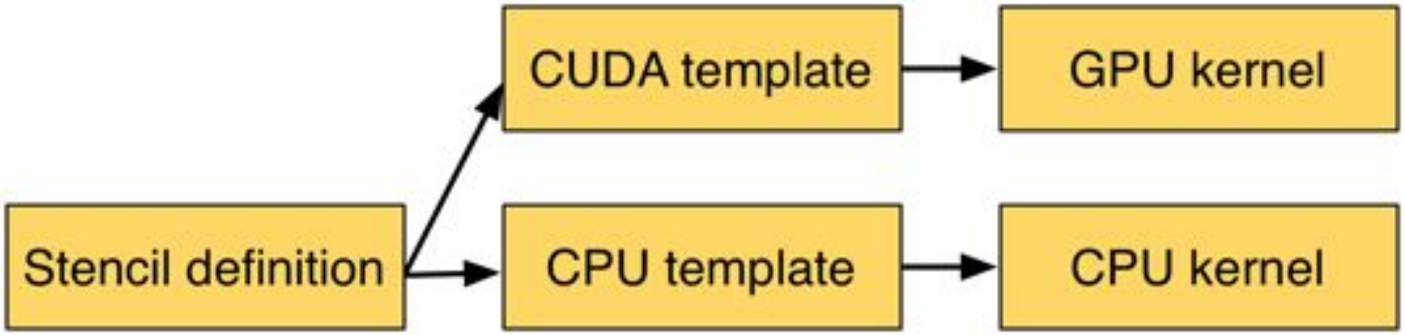
# A framework of stencil applications

- We have created a software for applications that use stencil operations
- “SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms”, T. Brandvik and G. Pullan, CIT 2010.
- Consists of two components:
  - Run-time library (memory allocation, MPI)
  - Kernel generator (source-to-source compiler)

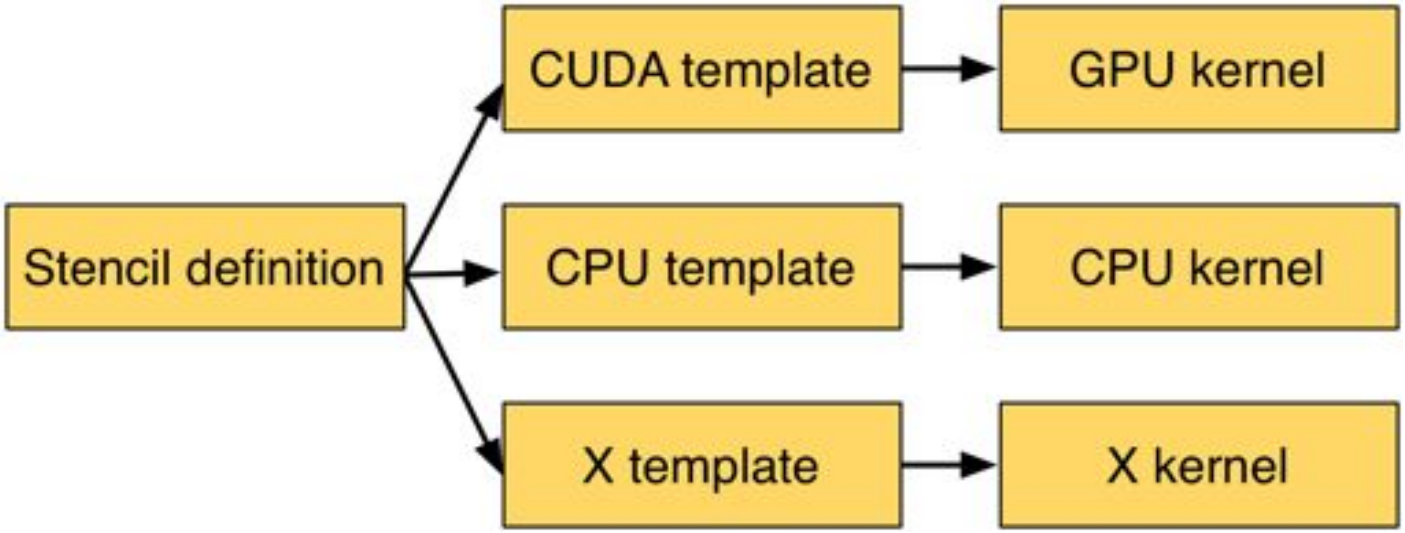
# SBLOCK kernel generator

- Different implementation strategies are necessary for different processors:
  - NVIDIA GPUs (CUDA)
  - CPUs (pthreads, SSE)
  - Other GPUs (OpenCL)
- Hand-coding each stencil is not feasible
- Kernel generator transforms a high-level definition of a stencil into optimised low-level source code

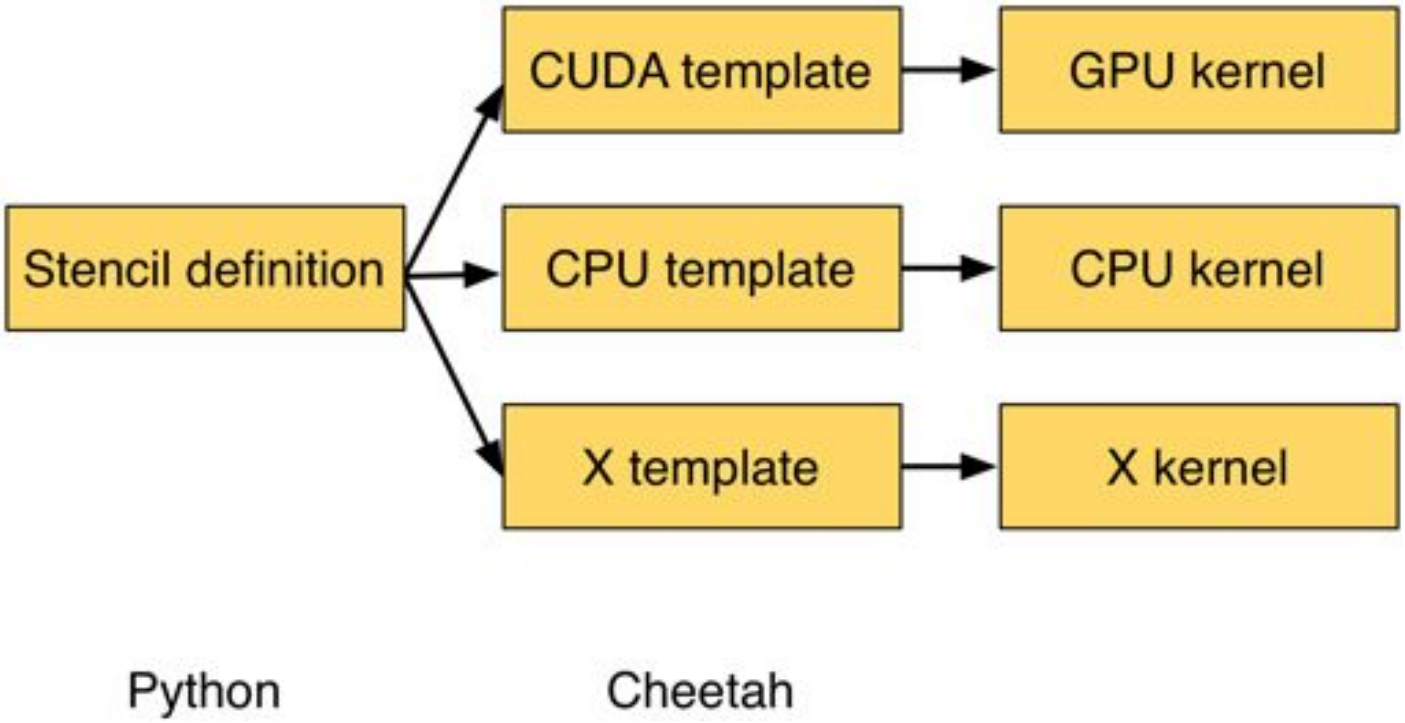
# SBLOCK kernel generator



# SBLOCK kernel generator



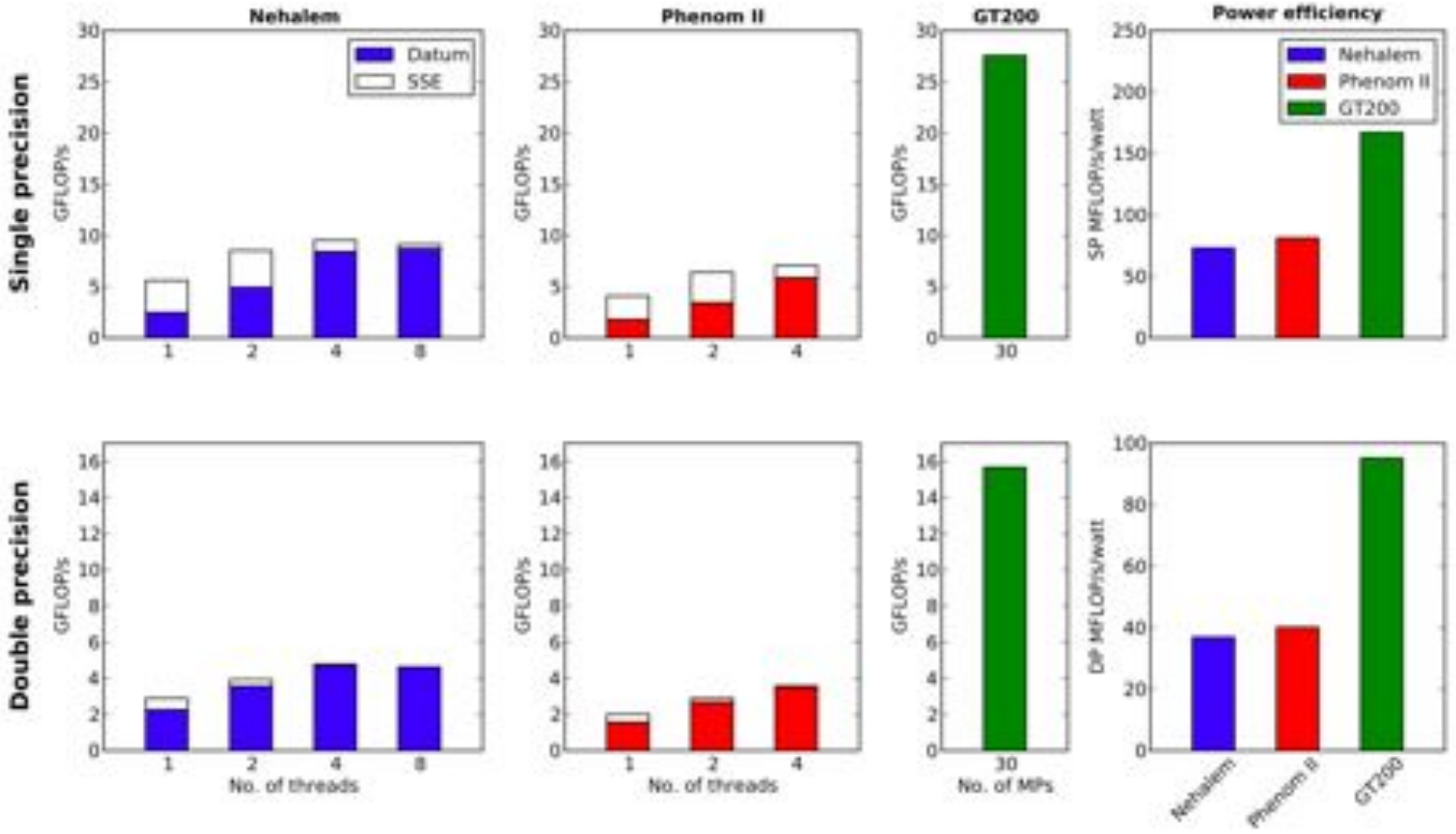
# SBLOCK kernel generator



# SBLOCK performance

- Laplace stencil
- Intel Nehalem (quad-core, 2.66 GHz)
- AMD Phenom II (quad-core, 3.0 GHz)
- NVIDIA Tesla (GT200)

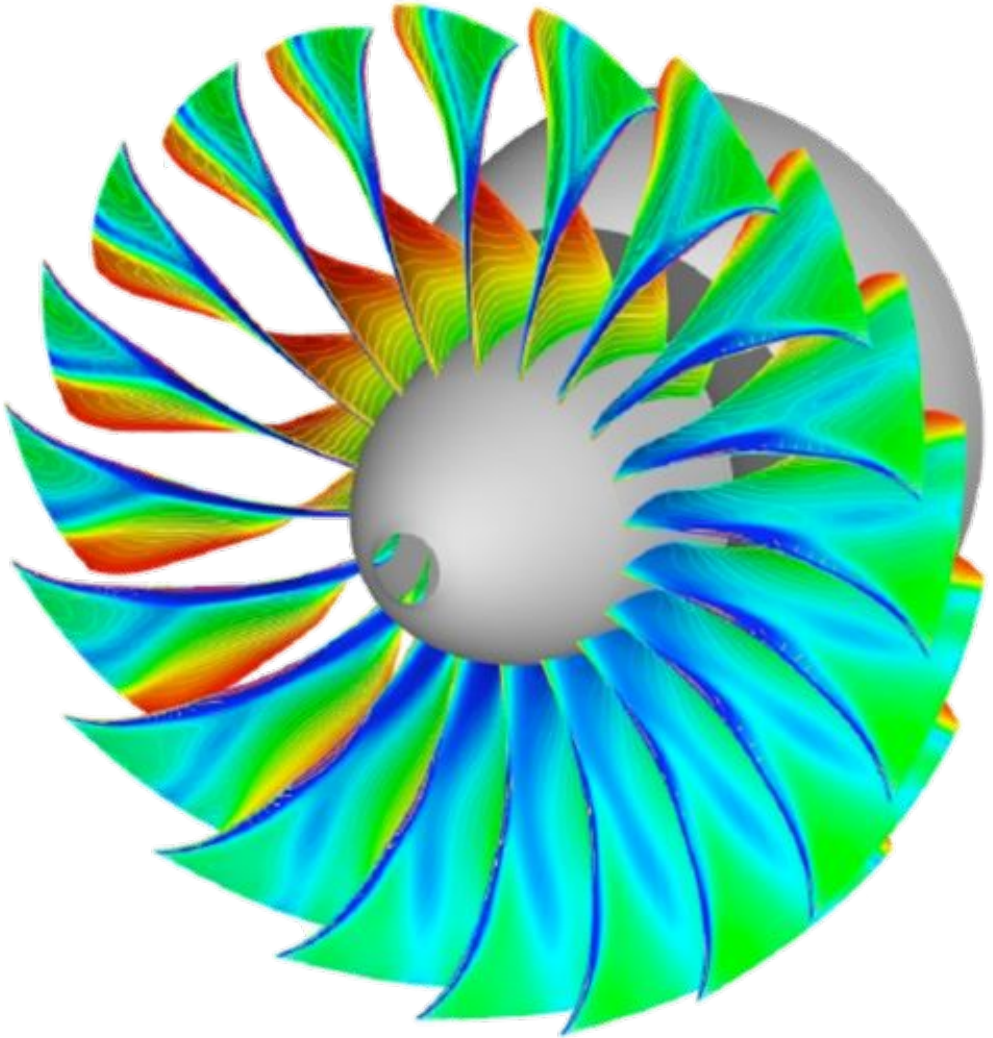
# SBLOCK performance



# Turbostream: A CFD solver implemented with SBLOCK

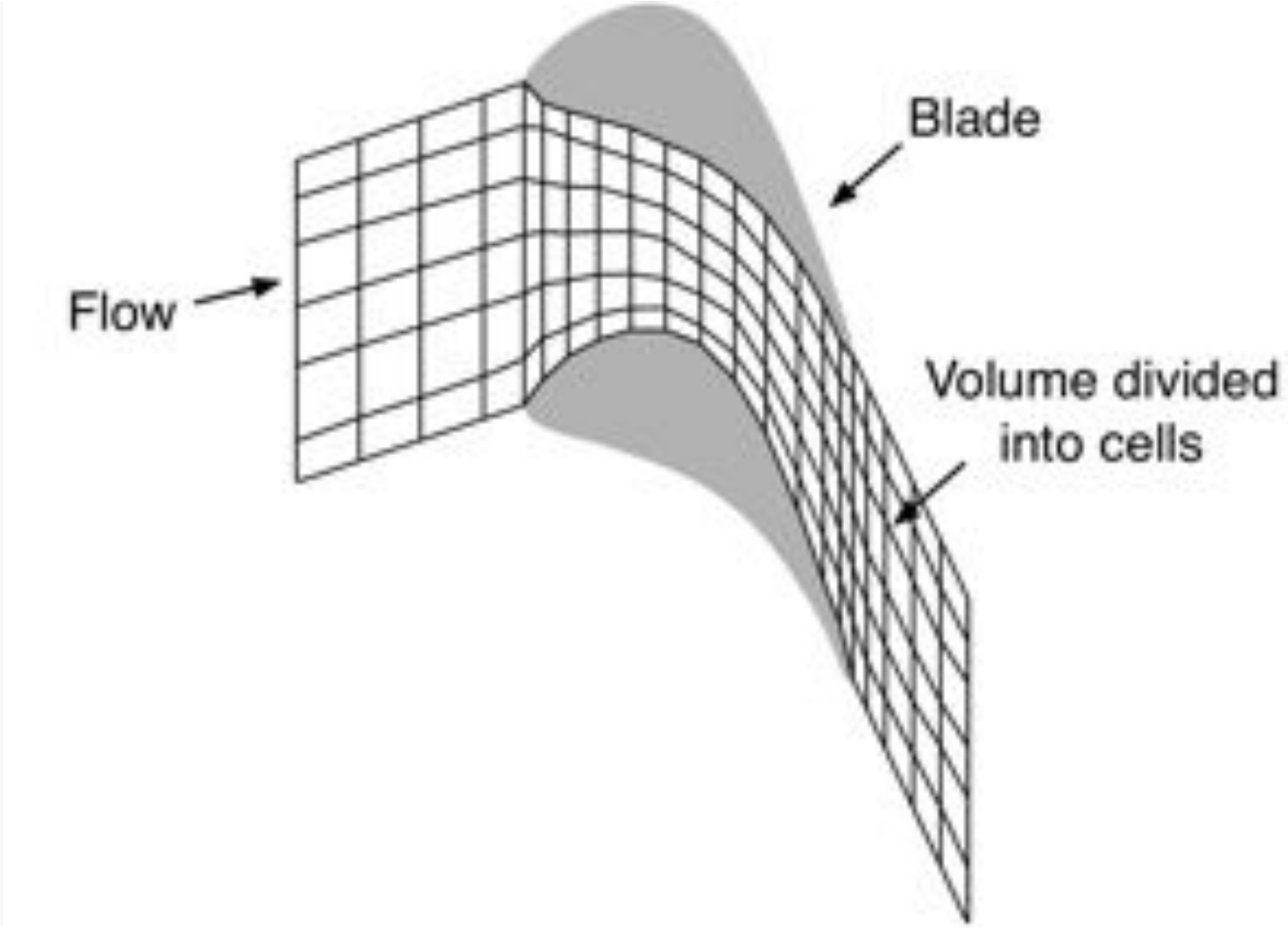
- Turbostream is a new CFD solver for the flow in turbomachines
- Based on the existing “Denton codes” from John Denton (Fortran+MPI)
- The Denton codes date back to the 1970s and are used in some part of the design process of most turbomachinery manufacturers

# CFD of a jet engine

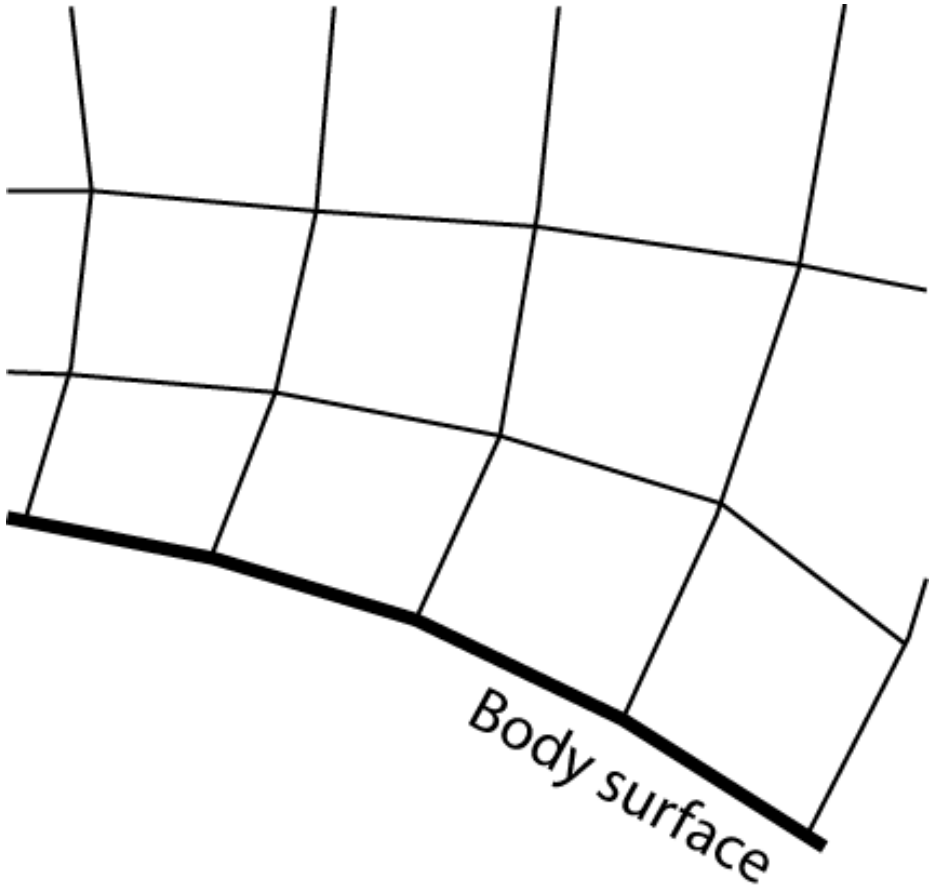


Blades coloured by pressure

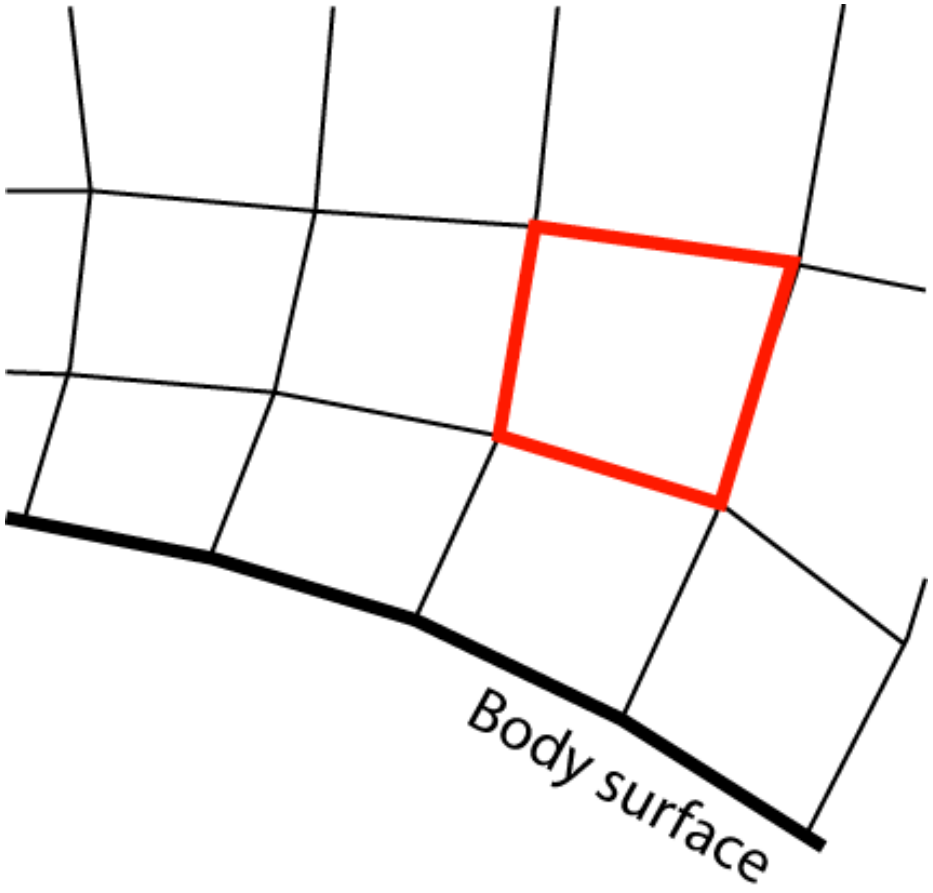
# Introduction to CFD



# Governing equations for each cell



# Governing equations for each cell

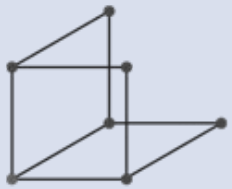
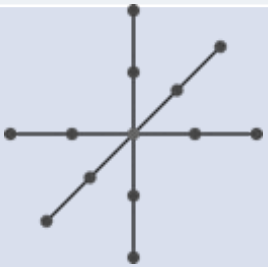



Conserve:
- Mass
- Momentum
- Energy

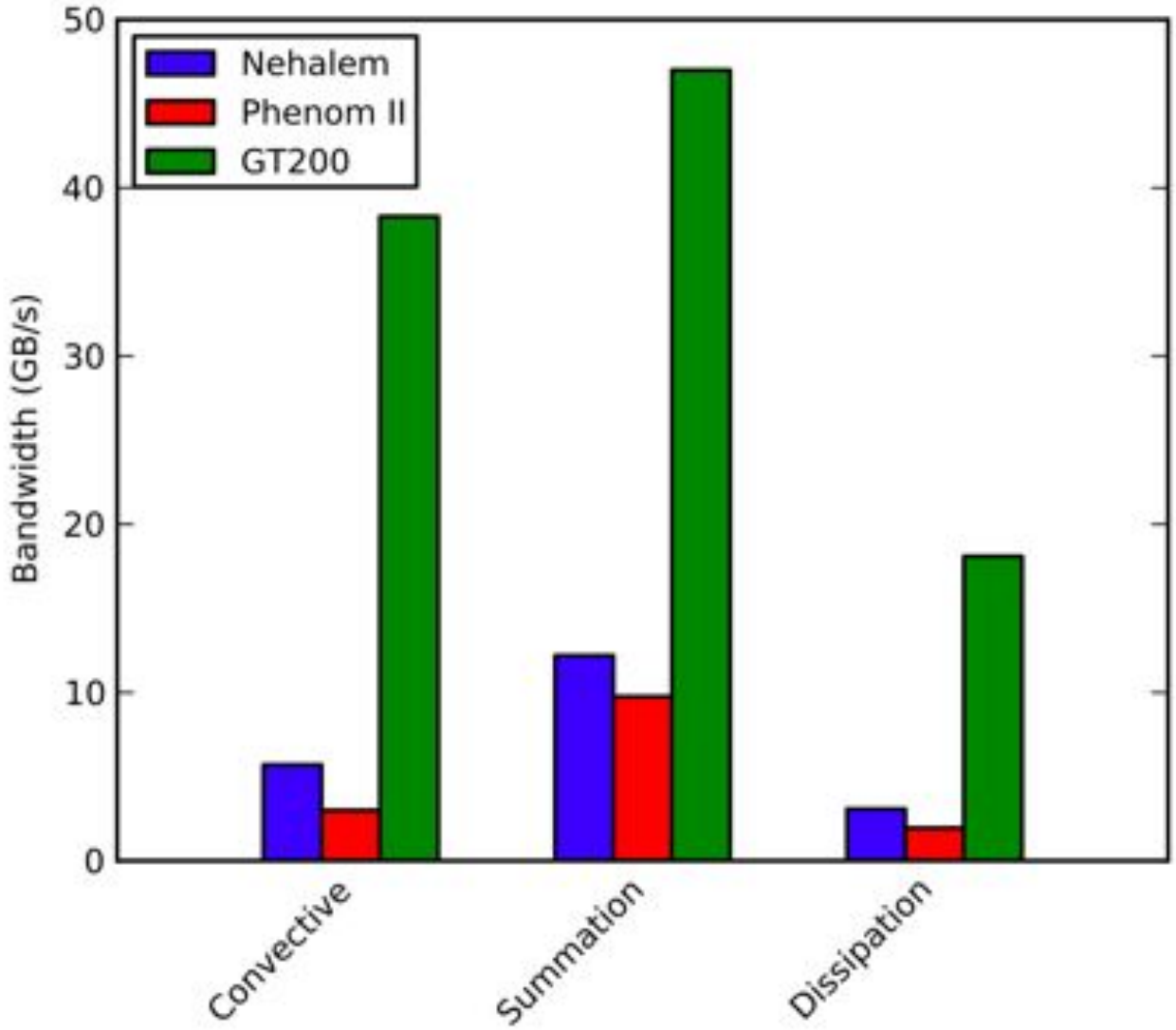
# Turbostream overview

- 20 stencil kernels
- Many different boundary conditions
- 7,500 lines of Python stencil definitions
- 20,000 lines of low-level source code after templates

# Turbostream kernels

| Kernel | Stencil | SP Flops | Bytes | AI |
|---|---|---|---|---|
| Convective fluxes |  | 260 | 139 | 1.87 |
| Flux summation |  | 8 | 28 | 0.29 |
| Artificial dissipation |  | 242 | 44 | 5.5 |

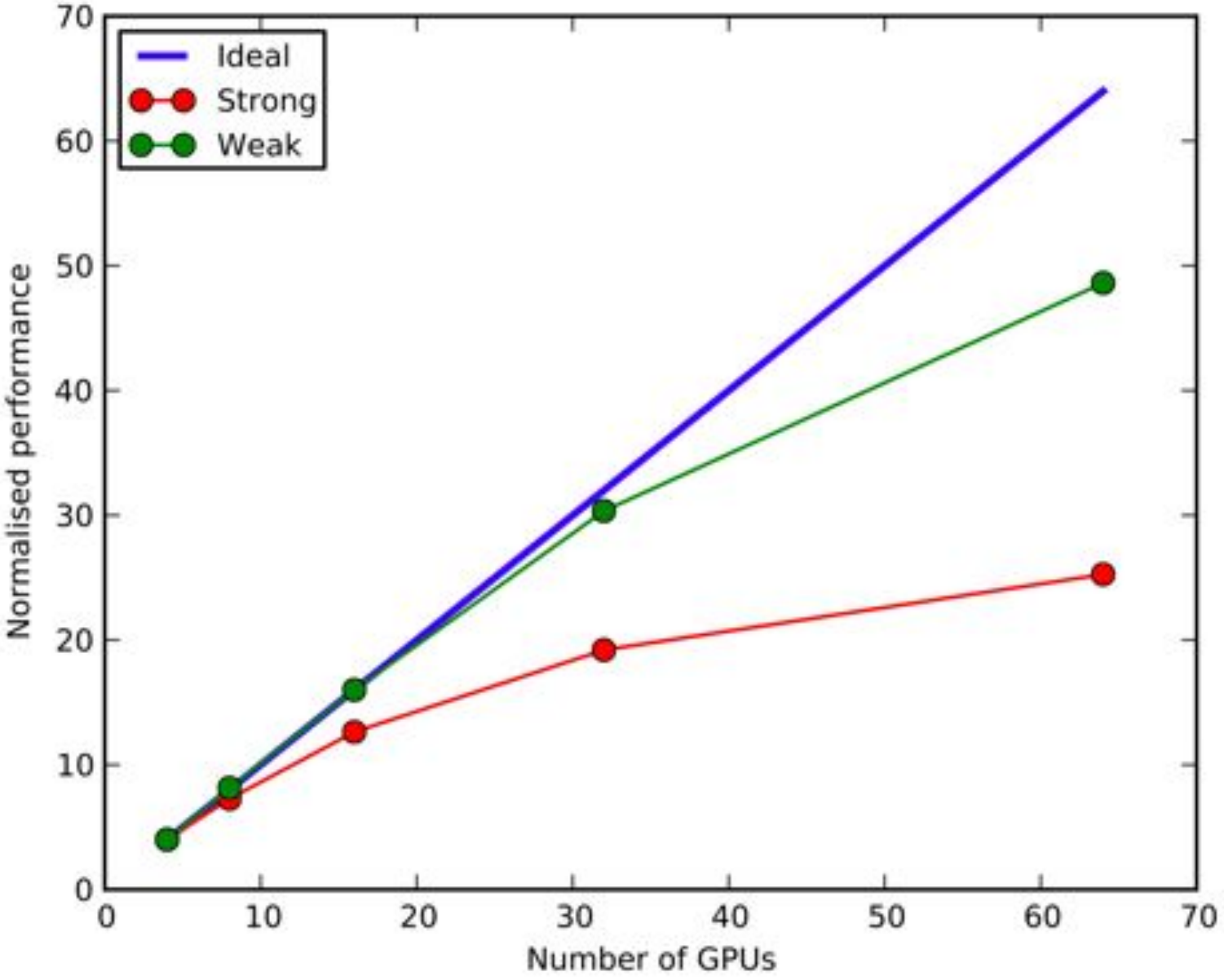
# Turbostream kernels



# Single-processor performance

| Processor | Original Fortran | Turbostream |
|---|---|---|
| Phenom II | 1 | 0.89 |
| Nehalem | 1.21 | 1.48 |
| GT 200 | - | 10.2 |

# Multi-GPU performance



# Impact on Turbomachinery CFD

- Typical design calculations:
  - 1-2 million grid nodes
  - Hours -> 10 minutes
  - Many more design iterations
- High-resolution simulations:
  - 20–200 million grid nodes
  - Months -> 1-2 weeks
  - Can be used in industry for the first time

# Turbostream Ltd

- A spin-off company has been formed around the Turbostream solver
- www.turbostream-cfd.com
  - Case studies
  - All papers and presentations

# Case study

- Full-annulus unsteady simulation of a compressor test rig at Siemens, UK
- 160 million grid nodes
- 32 NVIDIA C1060 GPUs
- 24 hours per revolution

# Animation

- http://vimeo.com/20403928