

# Example application: Two-dimensional Euler solver

Philip Blakely

Laboratory for Scientific Computing, Cambridge

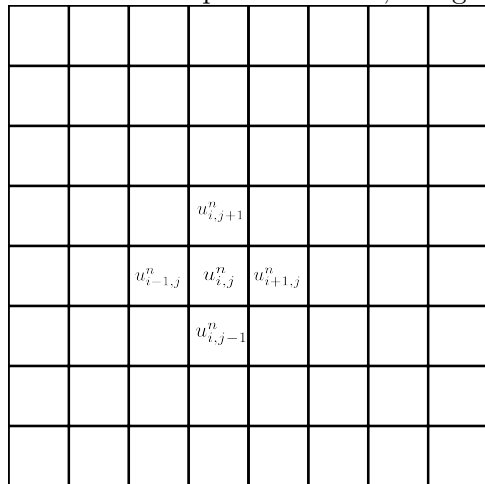
# From problem to optimized solution

Given a computational problem, you have to decide:

- Where the bottlenecks are in a standard CPU version (profile it)
- What speed-up is possible if these are optimized (Amdahl's law)
- Is this part of the code suited to GPUs?
- What degree of optimization can be attained (what factor speed-up)
- Whether this represents good payoff for effort invested
- Whether it would be easier to use MPI and multi-core machine(s)

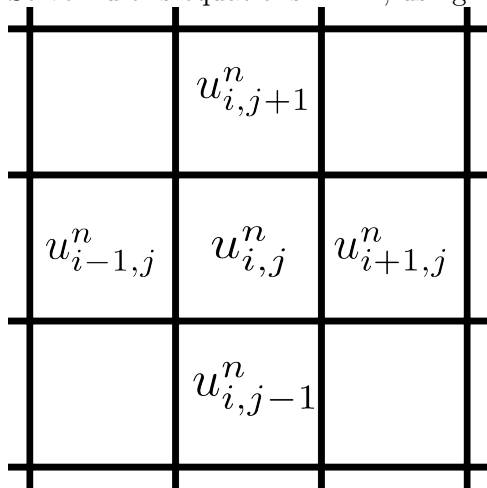
# Problem

Solve Euler's equations in 2D, using finite-volume approach:



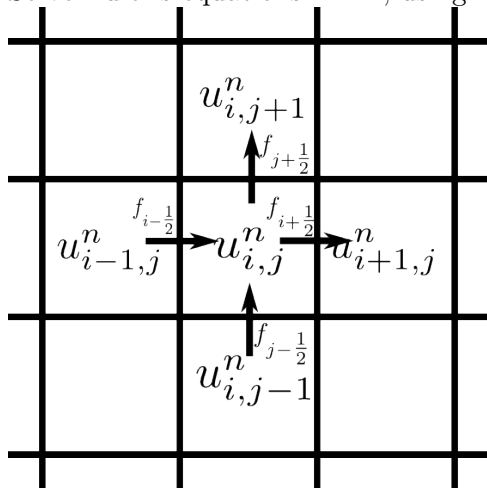
# Problem

Solve Euler's equations in 2D, using finite-volume approach:



# Problem

Solve Euler's equations in 2D, using finite-volume approach:



# Update formula

To update a cell of the grid:

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\Delta t}{\Delta x} (F_{i-1/2,j} - F_{i+1/2,j}) + \frac{\Delta t}{\Delta y} (F_{i,j-1/2} - F_{i,j+1/2})$$

Using  $\Delta t = 0.95 \times \frac{\Delta x}{\max_{i,j} v}$

Hard part is computing fluxes  $F$ .

# Potential gain analysis

- Bottleneck is flux calculation ( $\gtrsim 97\%$  of time) and time-step calculation (most of the rest)
- Suggests at least  $30\times$  speed-up
- Fluxes are computed independently - well suited to GPU threads
- HRSC schemes are floating-point-intensive - suggests fairly easy gain
- Many simulations take days to run -  $30\times$  speed-up worth it
- MPI is routinely used - probably use both GPUs and MPI

- Evolve Euler equations in 2D for an ideal gas
- Second order method - MUSCL-Hancock (Slope-limited reconstruction with exact Riemann solver)
- Run-time specified parameters:
  - Gamma
  - Grid size
  - Slope-limiter
  - CFL
  - Initial data



# Application outline

- Read in parameters
- Allocate memory on GPU
- Set initial data on GPU
- Loop over:
  - Determine time-step on GPU
  - Determine fluxes (x and y directions) on GPU
  - Add fluxes to data on GPU
- Output data (transfer from device to host)

- The full code for this is in `Examples/GPUeuler`
- Code takes input from `euler.in`
- CPU/GPU version can be chosen at run-time.
- Methods shown here give  $\approx 100\times$  speed-up (comparing a Tesla GPU with a single core of an Intel i7)
- Not necessarily best way to do everything - some aspects chosen for teaching purposes rather than optimal performance.
- No boundary conditions - just leave initial data in ghost cells
- Output in `out*.ppm`

## constants.cu

```
__constant__ int limiter;  
__constant__ float g[9];  
  
// Euler equations in 2D needs variables:  
enum Vars{RHO, V_X, V_Y, P, NUM_VARS};  
  
enum Limiter{firstOrder, minBee, vanLeer, superBee};  
  
enum Processor {CPU, GPU};
```

- `limiter` stores the slope-limiter
- `g[]` stores the adiabatic index  $\gamma$  and related constants (pre-computed for optimization)
- $g[0] = \gamma$ ,  $g[1] = \frac{\gamma-1}{2\gamma}$ ,  $g[2] = \frac{\gamma+1}{2\gamma}$ , ...

# Setting global variables

## main.cu

```
std::cin >> limiter_CPU;
std::cin >> gamma_CPU[0];
gamma_CPU[1] = (gamma_CPU[0]-1)/(2*gamma_CPU[0]);
gamma_CPU[2] = (gamma_CPU[0]+1)/(2*gamma_CPU[0]);

// Put simulation parameters onto GPU
cudaMemcpyToSymbol(limiter, &limiter_CPU, sizeof(limiter),
    0, cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(g, &gamma_CPU, sizeof(gamma_CPU), 0,
    cudaMemcpyHostToDevice);
```

# Setting initial data

- We must choose a memory layout.
- Two options:

①  $\rho_1, v_1^x, v_1^y, p_1, \rho_2, v_2^x, v_2^y, p_2, \dots$

```
struct solVector{
    float rho;
    float v_x;
    float v_y;
    float p;};
solVector data[];
```

②  $\rho_1, \rho_2, \dots, v_1^x, v_2^x, \dots, v_1^y,$

```
struct solVectors{
    float rho[];
    float v_x[];
    float v_y[];
    float p[];};
```

- For CPU, use (1), since a single flux calculation needs to access  $(\rho, v^x, v^y, p)$ , all close in memory, caches well.
- For GPU, use (2), so that adjacent threads access adjacent  $\rho$ s in adjacent memory locations - coalesced global memory access.

The general rule is:

- CPU: Array of Structures
- GPU: Structure of Arrays

This holds true for most applications, even on the latest cards.  
We shall be computing based on one thread per cell.

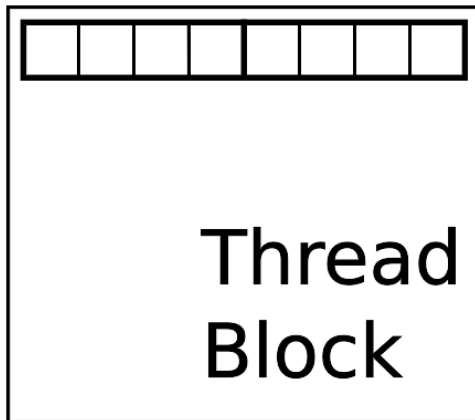
# Determine time-step

- Need to determine a maximum across whole grid
- Want a reduction on a GPU
- Ignore communication between blocks for the moment.
- So, reduce on each block (maximum size 1024) to a single value and reduce the values for each block on the CPU.
- See <https://devblogs.nvidia.com/paralleforall/faster-parallel-reductions-kepler/> for full details.
- Reduction is one of the more complex operations on a GPU; we don't cover it in detail here.

## Determine time-step ctd

Sketch diagram of reduction of 8 elements.

Time increases down the page; `threadIdx` increases left to right.

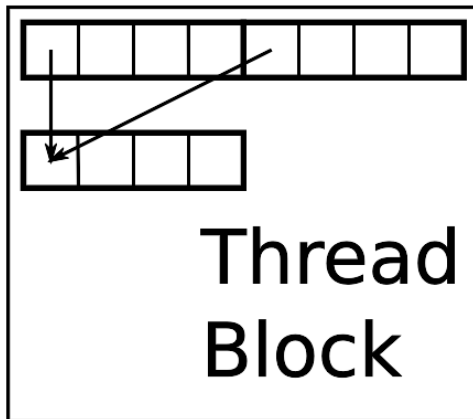




## Determine time-step ctd

Sketch diagram of reduction of 8 elements.

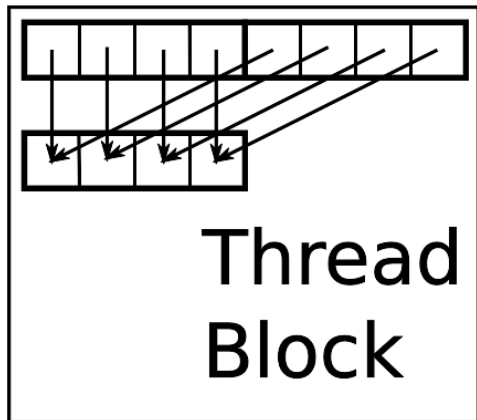
Time increases down the page; `threadIdx` increases left to right.



# Determine time-step ctd

Sketch diagram of reduction of 8 elements.

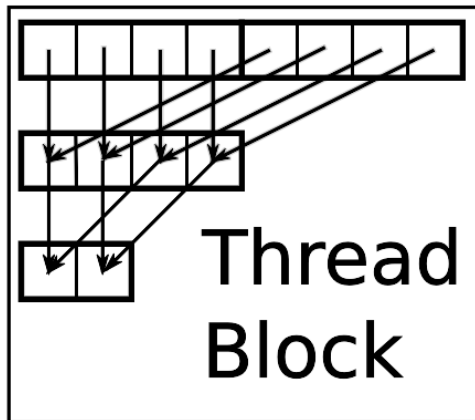
Time increases down the page; `threadIdx` increases left to right.



## Determine time-step ctd

Sketch diagram of reduction of 8 elements.

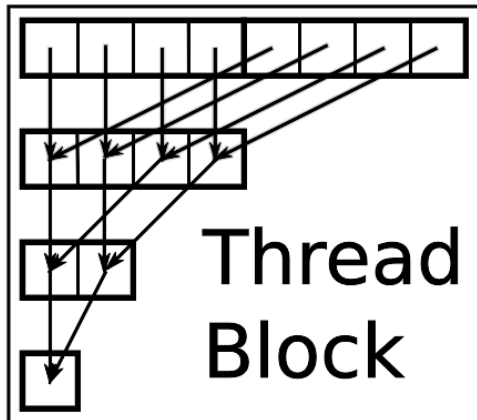
Time increases down the page; `threadIdx` increases left to right.



# Determine time-step ctd

Sketch diagram of reduction of 8 elements.

Time increases down the page; `threadIdx` increases left to right.



# Determine fluxes

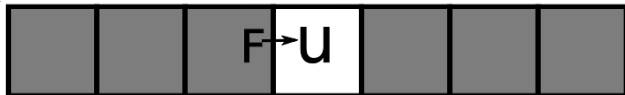
- On a CPU, we would allocate a new array of limited values, and calculate fluxes from these
- Could do the same on GPU. However for instructive purposes we do the whole calculation in one kernel except for final flux addition.



- Updating central cell

# Determine fluxes

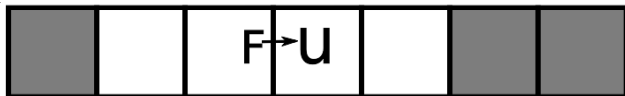
- On a CPU, we would allocate a new array of limited values, and calculate fluxes from these
- Could do the same on GPU. However for instructive purposes we do the whole calculation in one kernel except for final flux addition.



- Updating central cell
- Requires flux from left

# Determine fluxes

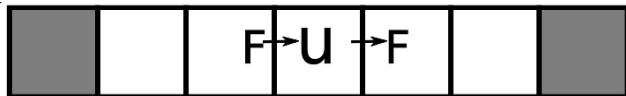
- On a CPU, we would allocate a new array of limited values, and calculate fluxes from these
- Could do the same on GPU. However for instructive purposes we do the whole calculation in one kernel except for final flux addition.



- Updating central cell
- Requires flux from left
- which requires slope-limited values from cells shown

# Determine fluxes

- On a CPU, we would allocate a new array of limited values, and calculate fluxes from these
- Could do the same on GPU. However for instructive purposes we do the whole calculation in one kernel except for final flux addition.

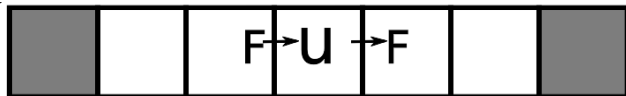


- Updating central cell
- Requires flux from left
- which requires slope-limited values from cells shown
- which needs data from cells shown



# Determine fluxes

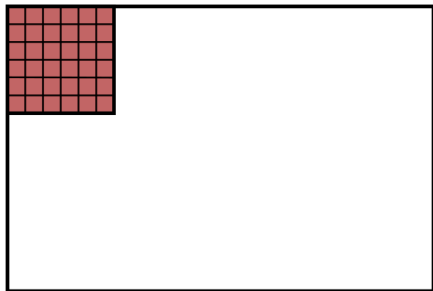
- On a CPU, we would allocate a new array of limited values, and calculate fluxes from these
- Could do the same on GPU. However for instructive purposes we do the whole calculation in one kernel except for final flux addition.



- Updating central cell
- Requires flux from left
- which requires slope-limited values from cells shown
- which needs data from cells shown
- Need cells shown overall to update single cell

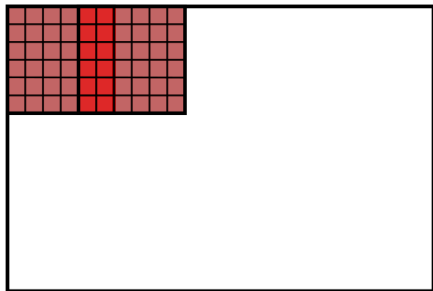
## Determine fluxes ctd.

- Want threads to share calculations of limited data
- Use shared memory to hold limited data for a block of cells
- Need overlapping blocks to hold required data in shared memory
- For calculating fluxes in x-direction ( $6 \times 6$  block):



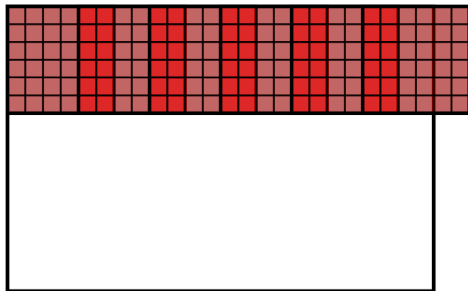
## Determine fluxes ctd.

- Want threads to share calculations of limited data
- Use shared memory to hold limited data for a block of cells
- Need overlapping blocks to hold required data in shared memory
- For calculating fluxes in x-direction ( $6 \times 6$  block):



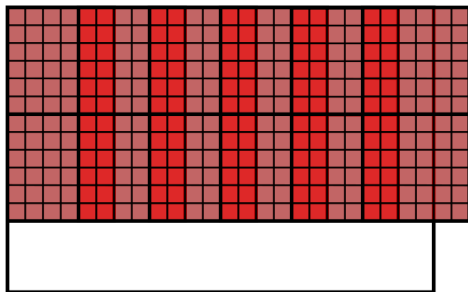
## Determine fluxes ctd.

- Want threads to share calculations of limited data
- Use shared memory to hold limited data for a block of cells
- Need overlapping blocks to hold required data in shared memory
- For calculating fluxes in x-direction ( $6 \times 6$  block):



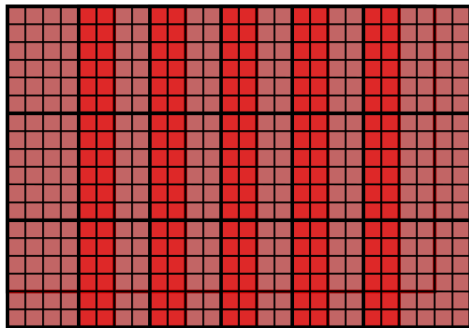
## Determine fluxes ctd.

- Want threads to share calculations of limited data
- Use shared memory to hold limited data for a block of cells
- Need overlapping blocks to hold required data in shared memory
- For calculating fluxes in x-direction ( $6 \times 6$  block):



## Determine fluxes ctd.

- Want threads to share calculations of limited data
- Use shared memory to hold limited data for a block of cells
- Need overlapping blocks to hold required data in shared memory
- For calculating fluxes in x-direction ( $6 \times 6$  block):



# Grid class

- Class for holding solution data on CPU/GPU
- Could do without it - but is instructive

grid.cu

```
struct Grid
{
...
    float* data;
    int xCells;
    int yCells;
    float xMin;
    float yMin;
    float xMax;
    float yMax;
    Processor proc;
};
```

## Grid class - ctd

```
// Create a grid with resolution xc*yc and covering
// [x0,y0] x [x1,y1]
Grid(int xc, int yc, float x0, float y0, float x1, float y1,
     Processor p)
{
    xCells = xc;
    yCells = yc;
    xMin = x0;
    xMax = x1;
    yMin = y0;
    yMax = y1;

    proc = p;

    switch(proc)
    {
    case CPU:
        data = new float[xc*yc*NUM_VARS];
        break;
    case GPU:
        cudaMalloc((void **)&data, xc*yc*NUM_VARS*sizeof(float));
        break;
    }
}
```



- Strided access, so that variable index varies the slowest
- Gives coalesced global memory access as required

```
// Access to data as r-value
__device__ __host__
float operator()(int i, int j, int v) const
{
    return data[i + j*xCells + v*xCells*yCells];
}

// Access to data as l-value
__device__ __host__
float& operator()(int i, int j, int v)
{
    return data[i + j*xCells + v*xCells*yCells];
}
```

# Grid class - thread block layout

- Using overlapping thread-blocks
- First one has width `blockSize.x`
- Rest cover an extra `blockSize.x - 2*overlap.x` cells each

Number of thread-blocks in  $x$  dimension given by:

$$1 + \left\lceil \frac{xCells - blockSize.x}{blockSize.x - 2 \times overlap.x} \right\rceil$$

See the earlier diagram to convince yourself of this.

# Considerations for shared-memory

- We want to use shared memory as much as possible
- Shared memory best accessed using stride 1 so need functions
  - dealing with arbitrary stride (either 1 or full volume)
  - and some that transform in place without using extra shared memory/registers

# Euler-specific functions

## idealGas.cu

```
__device__ void
primitiveToConservative(const
    float* prim, const int pStride,
    float* cons, const int cStride)
{
}
```

Calculate conserved variables from primitive variables with general strided vectors `prim` and `cons` where the adjacent elements of `prim` are separated by `pStride` in memory.

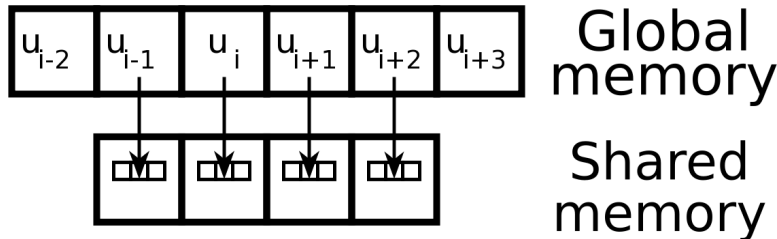
## idealGas.cu

```
if( g.contains(i, j) )
{
    primitiveToConservativeInPlace (&g(i, j, RHO),
    g.stride());
}
```

# Loading global data into shared memory

See `Examples/slic.cu` for full code.

- For each cell, need data from three cells to compute slope-limited values.

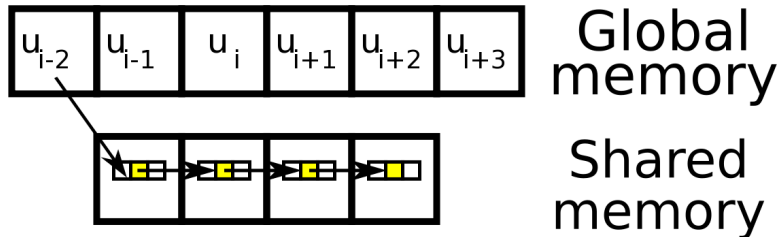


1) Load four cell-centred solution vectors into shared memory

# Loading global data into shared memory

See `Examples/slic.cu` for full code.

- For each cell, need data from three cells to compute slope-limited values.

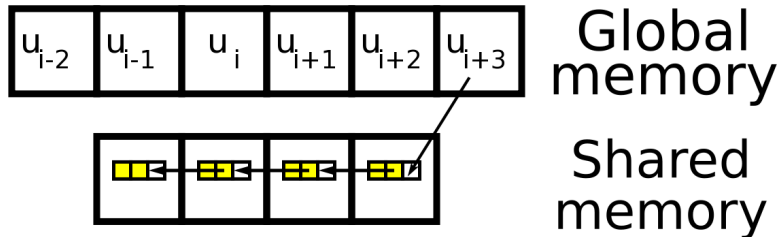


2) Load left data into shared memory

# Loading global data into shared memory

See `Examples/slic.cu` for full code.

- For each cell, need data from three cells to compute slope-limited values.

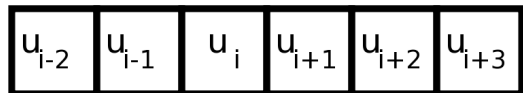


3) Load right data into shared memory

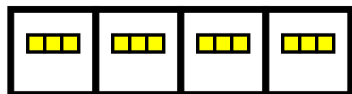
# Loading global data into shared memory

See `Examples/slic.cu` for full code.

- For each cell, need data from three cells to compute slope-limited values.



Global  
memory



Shared  
memory

4) Now have three vectors per thread in shared memory



# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$
Advance $\frac{1}{2}\Delta t$	$\bar{u}_{i-1}^L, u_{i-1}, \bar{u}_{i-1}^R$	$\bar{u}_i^L, u_i, \bar{u}_i^R$	$\bar{u}_{i+1}^L, u_{i+1}, \bar{u}_{i+1}^R$

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$
Advance $\frac{1}{2}\Delta t$	$\bar{u}_{i-1}^L, u_{i-1}, \bar{u}_{i-1}^R$	$\bar{u}_i^L, u_i, \bar{u}_i^R$	$\bar{u}_{i+1}^L, u_{i+1}, \bar{u}_{i+1}^R$
__syncthreads()			

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$
Advance $\frac{1}{2}\Delta t$	$\bar{u}_{i-1}^L, u_{i-1}, \bar{u}_{i-1}^R$	$\bar{u}_i^L, u_i, \bar{u}_i^R$	$\bar{u}_{i+1}^L, u_{i+1}, \bar{u}_{i+1}^R$
__syncthreads()			
Solve R.P.	$f(\bar{u}_{i-2}^R, \bar{u}_{i-1}^L)$	$f(\bar{u}_{i-1}^R, \bar{u}_i^L)$	$f(\bar{u}_i^R, \bar{u}_{i+1}^L)$

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$
Advance $\frac{1}{2}\Delta t$	$\bar{u}_{i-1}^L, u_{i-1}, \bar{u}_{i-1}^R$	$\bar{u}_i^L, u_i, \bar{u}_i^R$	$\bar{u}_{i+1}^L, u_{i+1}, \bar{u}_{i+1}^R$
__syncthreads()			
Solve R.P.	$f(\bar{u}_{i-2}^R, \bar{u}_{i-1}^L)$	$f(\bar{u}_{i-1}^R, \bar{u}_i^L)$	$f(\bar{u}_i^R, \bar{u}_{i+1}^L)$
__syncthreads()			

# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$
Advance $\frac{1}{2}\Delta t$	$\bar{u}_{i-1}^L, u_{i-1}, \bar{u}_{i-1}^R$	$\bar{u}_i^L, u_i, \bar{u}_i^R$	$\bar{u}_{i+1}^L, u_{i+1}, \bar{u}_{i+1}^R$
__syncthreads()			
Solve R.P.	$f(\bar{u}_{i-2}^R, \bar{u}_{i-1}^L)$	$f(\bar{u}_{i-1}^R, \bar{u}_i^L)$	$f(\bar{u}_i^R, \bar{u}_{i+1}^L)$
__syncthreads()			
Final Flux	$\Delta t \times (f_{i-2} - f_{i-1})$	$\Delta t \times (f_{i-1} - f_i)$	$\Delta t \times (f_i - f_{i+1})$



# Within shared memory

	Thread $i - 1$	Thread $i$	Thread $i + 1$
Initial	$u_{i-2}, u_{i-1}, u_i$	$u_{i-1}, u_i, u_{i+1}$	$u_i, u_{i+1}, u_{i+2}$
Reconstruct	$u_{i-1}^L, u_{i-1}, u_{i-1}^R$	$u_i^L, u_i, u_i^R$	$u_{i+1}^L, u_{i+1}, u_{i+1}^R$
Slope limit	$u_{i-1}^{L*}, u_{i-1}, u_{i-1}^{R*}$	$u_i^{L*}, u_i, u_i^{R*}$	$u_{i+1}^{L*}, u_{i+1}, u_{i+1}^{R*}$
Advance $\frac{1}{2}\Delta t$	$\bar{u}_{i-1}^L, u_{i-1}, \bar{u}_{i-1}^R$	$\bar{u}_i^L, u_i, \bar{u}_i^R$	$\bar{u}_{i+1}^L, u_{i+1}, \bar{u}_{i+1}^R$
__syncthreads()			
Solve R.P.	$f(\bar{u}_{i-2}^R, \bar{u}_{i-1}^L)$	$f(\bar{u}_{i-1}^R, \bar{u}_i^L)$	$f(\bar{u}_i^R, \bar{u}_{i+1}^L)$
__syncthreads()			
Final Flux	$\Delta t \times (f_{i-2} - f_{i-1})$	$\Delta t \times (f_{i-1} - f_i)$	$\Delta t \times (f_i - f_{i+1})$

- Functions such as `fluxInPlace(u)` mean that we only need 4 solution vectors per thread. But, for a  $32 \times 32$  block:
- Overall shared-memory:  $4 * \text{NUM\_VARS} * \text{blockDim.y} * \text{blockDim.x} * \text{sizeof(float)}$   
 $= 16,384$  bytes

- The Riemann solver is very similar to that in a CPU code
- It uses an iterative method to calculate the final pressure.
- We assume solution vectors are in shared memory and therefore strided
- Make frequent use of `prim_L[P * stride]` construct
- No real way to reduce branching due to iterations within Riemann solver
- Plausibly, neighbouring cells might need similar number of iterations in solver, so divergent branching is avoided.
- However, this is not true in general.

# Storing the final flux

## Adding the flux in the x-direction

```
__syncthreads();

//Only put flux into global memory if we're not on the ghost
  cells of the current thread block and not on ghost cells of
  whole grid
if(coord == 0 && 0 < threadIdx.x && threadIdx.x <
  blockDim.x-1 && fluxToCalculate)
{
  for(int v=0 ; v < NUM_VARS ; v++)
    flux(i,j,v) = dt/dCoords.x *
      (temp[1][v][threadIdx.y][threadIdx.x] -
       temp[1][v][threadIdx.y][threadIdx.x+1]);
}
```

# Adding the final flux

- Perform the addition in a separate kernel
- Now do not need overlapping thread-blocks
- Adding flux is completely local and correctly coalesced due to structure of arrays approach.

# Adding the final flux

```
template<int coord>
__global__ void addFlux_GPU(Grid u, Grid flux)
{
    const int i = blockIdx.x * blockDim.x +
        threadIdx.x; //x-index on main grid
    const int j = blockIdx.y * blockDim.y +
        threadIdx.y; //y-index on main grid

    const int i_left_limit = (coord == 0) ? 2 : 0;
    const int i_right_limit = (coord == 0) ? u.xCells - 2 :
        u.xCells;
    const int j_left_limit = (coord == 1) ? 2 : 0;
    const int j_right_limit = (coord == 1) ? u.yCells - 2 :
        u.yCells;

    if( i >= i_left_limit && j >=j_left_limit && i <
        i_right_limit && j < j_right_limit )
    {
        for(int v=0 ; v < NUM_VARS ; v++)
            u(i, j, v) += flux(i, j, v);
    }
}
```

# How fast?

We should now check how fast our implementation goes  
Benchmark with:

- $1000 \times 1000$  cells
- CFL 0.95
- 2D Riemann problem initial-data
- Adiabatic index  $\gamma = 1.4$
- van Leer limiter
- End time  $T=0.2s$

Result: 6.65s (on tycho - Tesla K20c)

# Templating

In expressions such as

```
const int dim_x = blockDim_x - ((coord == 0) ? 2 : 0);
const int dim_y = blockDim_y - ((coord == 1) ? 2 : 0);
const float* u_left = &u( i - ((coord==0) ? 1 : 0 ), j -
    ((coord==1) ? 1 : 0 ), RHO);
const float* u_right = &u( i + ((coord==0) ? 1 : 0 ), j +
    ((coord==1) ? 1 : 0 ), RHO);
```

divergent branching isn't a problem.

- However, some instructions are needed to branch on `coord`
- So get the compiler to evaluate branching

```
template<int blockDim_x, int blockDim_y, int coord>
__global__ void getSLICflux_GPU(Grid u, Grid flux, float dt)

    const int xSizeXsolve=8, ySizeXsolve=8; //Block size when
        solving for x-fluxes
    getSLICflux_GPU<xSizeXsolve, ySizeXsolve, 0><<<xSLICgrid,
        xSLICblocks>>>(grid, flux, dt);
```

This gives at least some speed-up over branching version:

Result: 5.80s (on poros - Quadro K620)

# What block-size should we use? (x-direction)

- When performing SLIC update, want to reduce no. of overlap cells
- For a grid of  $1000 \times 1000$  cells, solving for x-fluxes
- Tesla K20c has 48kB shared-memory per SM.

Block size	Overall time
$16 \times 16$	5.54s
$32 \times 8$	5.41s
$8 \times 32$	5.91s
$32 \times 16$	6.34s
$16 \times 32$	6.62s

- We modify `slic.cu`

- First 3 lines can have 3 thread blocks per multiprocessor.
- Last two lines are limited to 1 thread block per multiprocessor.
- Latency cannot be hidden as easily by swapping execution to different thread blocks.



# What block-size should we use? (y-direction)

- In  $y$  direction, different effects come into play
- For global memory access coalescence, want to read several adjacent cells in  $x$ -direction.
- So,  $8 \times 32$  is not the obvious answer
- For a grid of  $1000 \times 1000$  cells, solving for  $y$ -fluxes:

Block size	Overall time
$16 \times 16$	5.14s
$32 \times 8$	5.38s
$8 \times 32$	5.08s
$32 \times 16$	6.07s
$16 \times 32$	5.98s

- (with x-block  $32 \times 8$ )
- Actually, it turns out  $8 \times 32$  is the unobvious answer...
- This seems to be card dependent (earlier cards had different answers).

# Avoid flux-update kernel

Initially, performed update as:

- `calcFlux(u, flux, X_COORD)`  $\text{flux} = f^x(u)$
- `addFlux(u, flux)`  $u = u + f^x(u)$
- `calcFlux(u, flux, Y_COORD)`  $\text{flux} = f^y(u)$
- `addFlux(u, flux)`  $u = u + f^y(u)$

Instead, we can calculate updated solution directly in extra array:

- `advanceSoln(u, u_plus, X_COORD)`  $u^+ = u + f^x(u)$
- `advanceSoln(u_plus, u, Y_COORD)`  $u = u^+ + f^y(u^+)$

This gives an extra 8% speed-up

- Optimizing an algorithm for CUDA can be tricky.
- Requires some thought to make best use of shared memory and reduce arithmetic operations.
- In practice, this may not make a dramatic speed-up.
- We went from 6.65s to 5.08s (24% time-saving) but with quite a lot of effort.
- However, being aware of available hardware characteristics is important.