

Some super-simple benchmarked GPU kernels
that nobody really cares about – yet are relevant
for my activity.

Norman Pellet

(and yes that's my title...)

Disclaimer

- I wanted to show you what I've been playing with recently. But sadly I'm not allowed to share any code. So I took this opportunity to write some independent code that I will be able to re-use later. So my idea was to do some benchmarking for some of the code I should have been optimizing anyway. I simplified the problems to their bare minimum to make the understandable for this presentation and wrote different implementations.
- I had little time to do that on my free time. So my apologies for the partial lack of clarity and semantics.
- All of the code is mine. The section on parallel reduction (which I might include by Sunday) is inspired by Mark Harris's work (from Nvidia).
- I hope it won't be too boring, too simple or too naive.

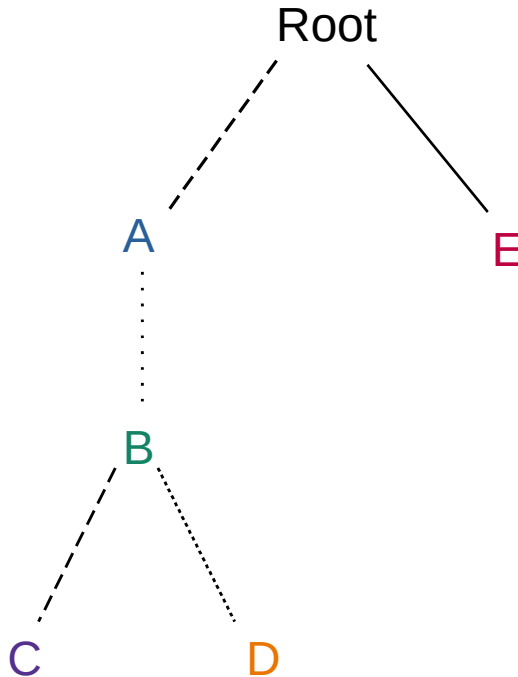
Motivation

- **4m²** of surface structuring with **1μm** resolution. File size: **3.6To** (8-bit gray, uncompressed).
- Impossible to store in RAM, possible but not practical to swap
- Potential solution: on-the-fly calculation and streamed directly into the machine controller
- Target calculation rate: 10Mo/s bandwidth

The challenges

- 1.Design a modular structure that can generate any design / pattern according to the client's idea
- 2.Keeping up with the necessary bandwidth

Tree representation



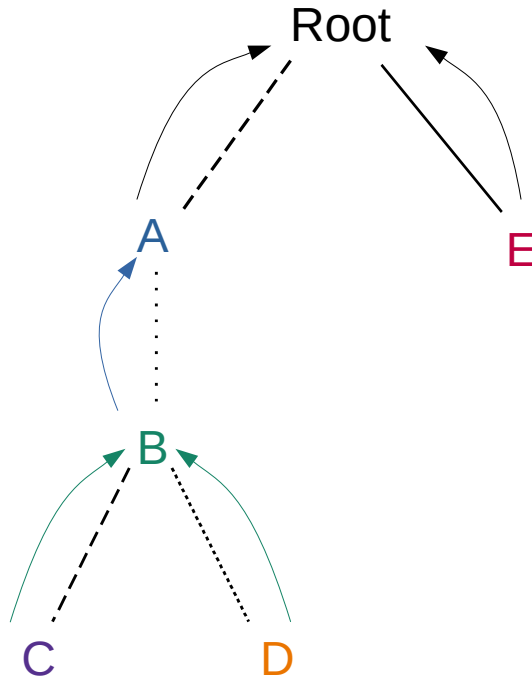
Each node have specific purpose, that use the data in their child node to compute a result.

The root node contains the output data

Each node is linked by a trait (or a flavor) which describes the function of the child node

`decltype(node)` defines a reduction strategy for each applicable trait.

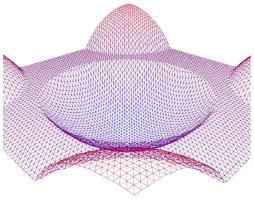
Tree reduction



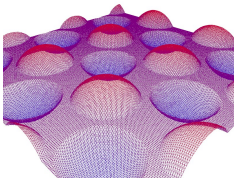
1. Reduce **C** into **B** according to trait **-----**
2. Reduce **D** into **B** according to trait **.....**
3. Reduce **B** into **A** according to trait **.....**
4. Reduce **A** into Root according to trait **-----**
5. Reduce **E** into Root according to trait **——**

Tree reduction

Root
Math2D



Root
Stitching
Math2D



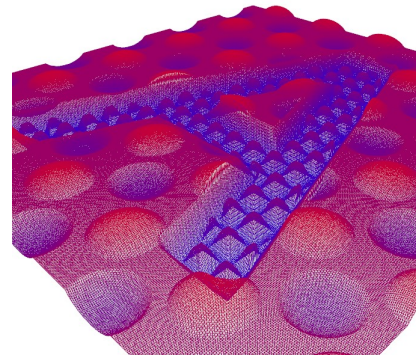
Root
Union
Stitching
Math2D

Profile
Tiff
A

mask

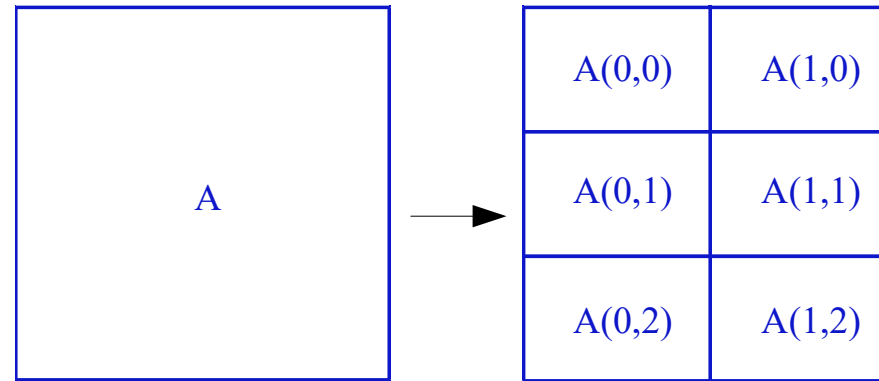
data

Stitching
Math2D



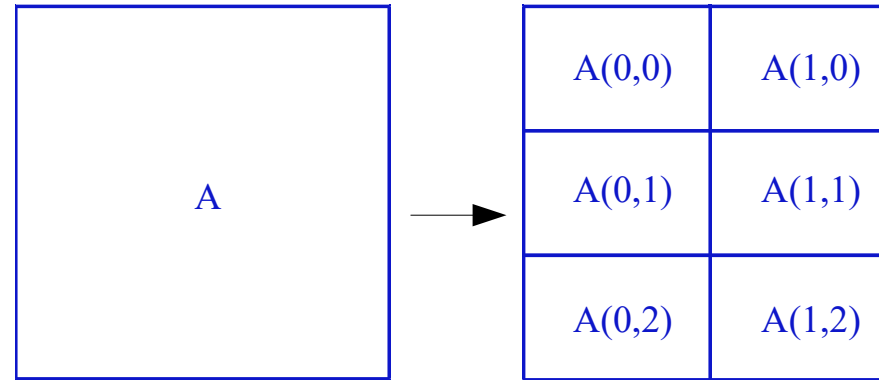
Tile system

- Memory constraints
- Each node has an independent width, height and offset
- Each node is therefore split into tiles



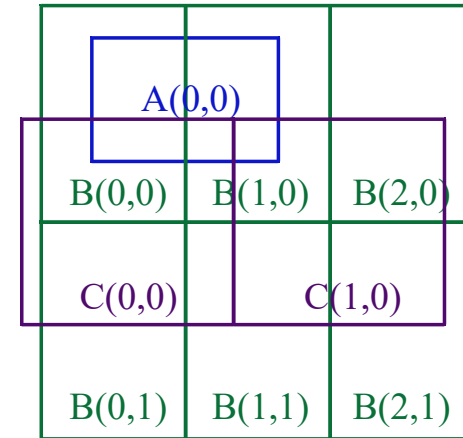
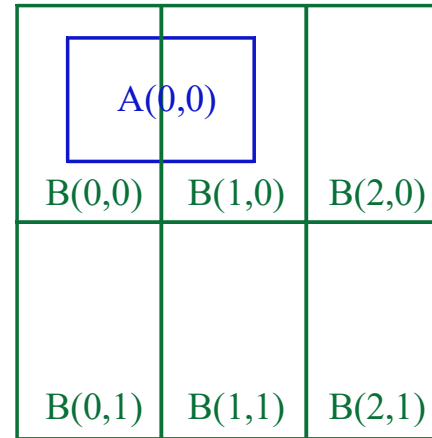
Reduction in a tile system

- Memory constraints
- Each node has an independent width, height and offset
- Each node is split into **tiles**
- Each tile can be computed separately



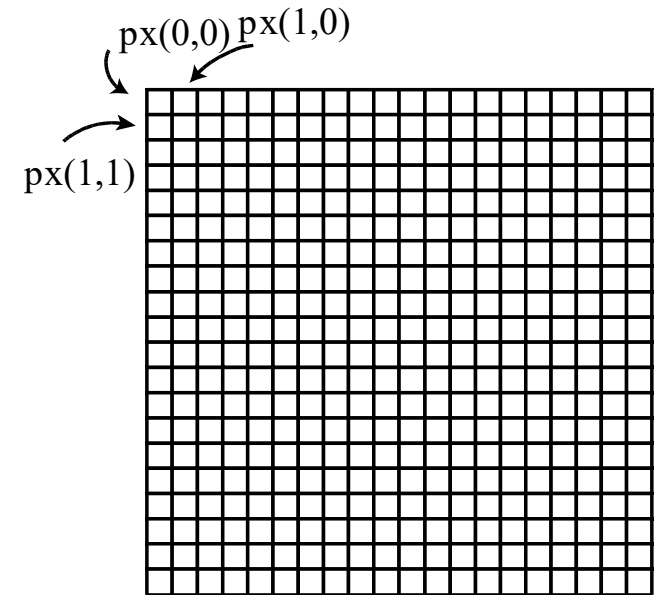
Reduction in a tile system

- Tile intersection calculation.
- Ex: to calculate $A(0,0)$ we need to calculate $B(0,0)$ and $B(1,0)$
- Ex: to calculate $B(0,0)$ we need to calculate $C(0,0)$.
To calculate $B(1,0)$ we need to calculate $C(0,0)$ and $C(1,0)$
- Here, $C(0,0)$ can be re-used.
Tiles can therefore be stored in an LRU system
- System memory usage is predictable



Calculation strategies

- The complete tree reduction must perform at 10Mo/s.
- The reductions must be considered sequentially
- But each computation is suitable for parallel computation
- Each branch can be computed concurrently
- Here comes in CUDA (OpenCL or potentially OpenGL 4+ with compute shaders could also work)
- I will show how CUDA could be useful for solving the data engraving file calculation bottlenecks and how it can speed up simple matrix calculations.



Each pixel can be calculated in parallel thanks to CUDA

Benchmarking conditions

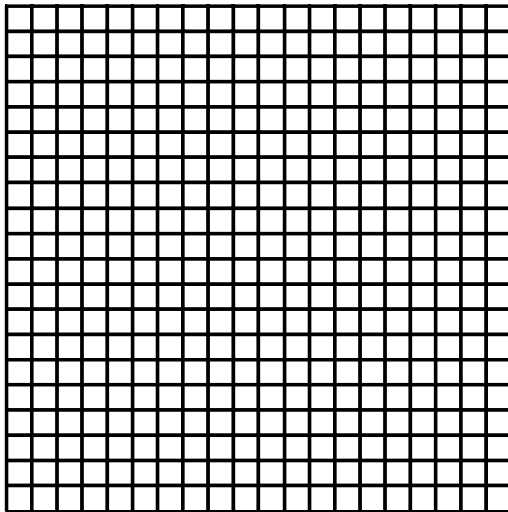
- CPU: Intel i9-9900k (single thread used)
- GPU: RTX 2070
- RAM: 32Gb
- Windows 10 Pro
- MSVC + CUDA 10.2

Simple stitching

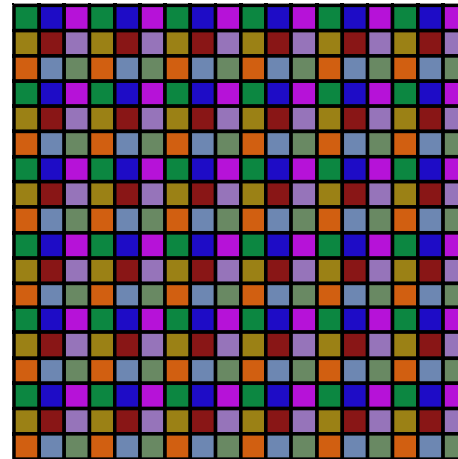
Child tile
(trait = pattern)



Parent tile



Reduction



(N.B.: In practice, the matrix might already be masked, so that the stitching only occurs in the necessary areas)

Let's define a target (the big empty matrix) and a repeater (the small coloured matrix)
(Here, `SIZE_REPEATER` and `SIZE_GLOBAL` are compile-time constant, but in practice they are not.
It's just to make it easier for the test and also (mostly) because it looks more fancy)

```
typedTile<float, GPUMatrixAllocator<float>> repeater{ SIZE_REPEATER, SIZE_REPEATER };  
typedTile<float, GPUMatrixAllocator<float>> target{ SIZE_GLOBAL, SIZE_GLOBAL };
```

Case 1.1 — Host code, basic

```
for (uint32_t x = 0; x < SIZE_GLOBAL; x++) {  
    for (uint32_t y = 0; y < SIZE_GLOBAL; y++) {  
        target.loadPointXY_assert(x, y, repeater.getPointXY(x % SIZE_REPEATER, y % SIZE_REPEATER));  
    }  
}  
  
void loadPointXY_assert(int32_t x, int32_t y, T value) {  
    assert(x >= 0);  
    assert(x < width);  
    assert(y >= 0);  
    assert(y < height);  
  
    loadPoint(calcIndex(x, y), value);  
}  
  
void inline loadPoint(const size_t position, const T val) {  
    assert(position < calcMaxIndex());  
    hostData[position] = val;  
}
```

Limitations:

- Modulo is slow (actually the result is skewed by the fact that `SIZE_REPEATER` is a compile-time constant, which enables compiler optimization. I noticed that in practice it makes little of a difference as it's not the time limiting step)
- For each point, the loading index in target must be recalculated ($\text{index} = y * \text{width} + x$)

Case 1 - Results

Host (basic)	870'000 μ s / iteration	

Conditions:

SIZE_GLOBAL = 10240

SIZE_REPEATER = 100

Case 1.2 — Host code, indexing

```
size_t index = 0;
for (uint32_t y = 0; y < SIZE_GLOBAL; y++) {
    for (uint32_t x = 0; x < SIZE_GLOBAL; x++) {
        target.loadPoint(index, repeater.getPointXY(x % SIZE_REPEATER, y % SIZE_REPEATER));
        index++;
    }
}
```

Limitations:

- Saves a lot of multiplications and additions
- Assumes that we know how to storage is arranged.
- Modulo is STILL slow

Case 1 - Results

Host (basic)	870'000 μ s / iteration	
Host (indexed)	114'000 μs / iteration	7.5x faster
Host (ptr increment)	99'600 μs / iteration	8.7x faster

Conditions:

SIZE_GLOBAL = 10240

SIZE_REPEATER = 100

Case 1.3 — Host code, loop over repeater

```
size_t repeaterIndex = 0;
for (uint32_t x = 0; x < SIZE_REPEATER; x++) {
    for (uint32_t y = 0; y < SIZE_REPEATER; y++) {
        const float repeaterValue = repeater.getPoint(repeaterIndex);

        for (uint32_t xB = x; xB < SIZE_GLOBAL; xB += SIZE_REPEATER) {
            for (uint32_t yB = y; yB < SIZE_GLOBAL; yB += SIZE_REPEATER) {
                target.loadPointXY(xB, yB, repeaterValue);
            }
        }
        repeaterIndex++;
    }
}
```

Limitations:

- We cannot cache the target index anymore
- But we gain the modulo
- No more sequential memory access over target matrix

Case 1 - Results

Host (basic)	870'000 μ s / iteration	
Host (indexed)	114'000 μ s / iteration	7.5x faster
Host (ptr increment)	99'600 μ s / iteration	8.7x faster
Host (repeater loop)	912'000 μs / iteration	slower

Conditions:

SIZE_GLOBAL = 10240

SIZE_REPEATER = 100

Case 1.4 – Host code, indexing

```
template<typename T>
__global__ void kernelRepeaterModulo(
    T* target,
    const uint32_t width,
    const uint32_t height,
    const T* repeater,
    const uint32_t repeaterWidth,
    const uint32_t repeaterHeight ) {

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

    if (col >= width || row >= height)
        return;

    const size_t targetIndex = getMatrixIndex(col, row, width);
    const uint32_t colTarget = col % repeaterWidth;
    const uint32_t rowTarget = row % repeaterHeight;

    target[targetIndex] = repeater[getMatrixIndex(colTarget, rowTarget, repeaterWidth)];
}
```

Calculate (x,y) from the CUDA grid

Minor but unavoidable warp divergence

Cannot cache

Cannot avoid modulo (not a compile constant)

Is blockDim.x == 32 (= warp size) then
the coalescing of gmem makes it such
that only 1 load is necessary (max 2, as
the matrix is not strided in this case)

Case 1 - Results

Host (basic)	870'000 μ s / iteration	
Host (indexed)	114'000 μ s / iteration	7.5x faster
Host (ptr increment)	99'600 μ s / iteration	8.7x faster
Host (repeater loop)	912'000 μ s / iteration	slower
GPU – with D-H and H-D copy	34'000 μs / iteration	25x faster
GPU – without copy	2'030 μs / iteration	430x faster

Conditions:

SIZE_GLOBAL = 10240

SIZE_REPEATER = 100

Case 1.5 — The mistake of shared mem

```
template<typename T, uint32_t SIZE_REPEATER>
__global__ void kernelRepeaterModuloShared(
    T* target,
    const uint32_t width,
    const uint32_t height,

    const T* repeater,
    const uint32_t repeaterWidth,
    const uint32_t repeaterHeight
) {
    __shared__ T s[SIZE_REPEATER * SIZE_REPEATER];

    if (threadIdx.x == 0 && threadIdx.y == 0) {
        memcpy(s, repeater, sizeof(T) * SIZE_REPEATER * SIZE_REPEATER);
    }
    __syncthreads();

    // Use s instead of repeater
    // ...
}
```

Copy repeater in statically assigned shared memory
Templated methods are impractical with dynamically allocated shared mem

Why is it plain stupid ?

Shared memory is block-wide (not grid-wide), so it is used by max 1024 threads.

With a repeater that is 100x100 (10'000 points), we will have 10'000 gmem loads into smem, and 1024 loads from smem. This is a major deoptimisation is $\text{SIZE_REPEATER}^2 > 1024$.

(I admit I realized it after trying it out...)

Case 1 - Results

Host (basic)	870'000 μ s / iteration	
Host (indexed)	114'000 μ s / iteration	7.5x faster
Host (ptr increment)	99'600 μ s / iteration	8.7x faster
Host (repeater loop)	912'000 μ s / iteration	slower
GPU – with D-H and H-D copy	34'000 μ s / iteration	25x faster
GPU – without copy	2'030 μ s / iteration	430x faster
GPU – smem load	816'000 μs / iteration	

Conditions:

SIZE_GLOBAL = 10240

SIZE_REPEATER = 100

Case 1.6 — Grid over repeater

```
template<typename T>
__global__ void kernelRepeaterRepGrid(
    T* target,
    const uint32_t width,
    const uint32_t height,

    const T* repeater,
    const uint32_t repeaterWidth,
    const uint32_t repeaterHeight
) {
    const uint32_t col = blockIdx.x * blockDim.x + threadIdx.x;
    const uint32_t row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col >= repeaterWidth || row >= repeaterHeight) {
        return;
    }

    const T repeaterValue = repeater[getMatrixIndex(col, row, repeaterWidth)];

    for (int y = row; y < height; y += repeaterHeight) {
        size_t index = getMatrixIndex(col, y, width);
        for (int x = col; x < width; x += repeaterWidth) {
            target[index] = repeaterValue;
            index += repeaterWidth;
        }
    }
}
```

Grid is adjusted to the size of the repe

Heavier load on the thread because of
the loop
No modulo but **scattered gmem access**

Speed gain will be very dependent on the repeater size and somewhat unpredictable

Case 1.6 — Grid over repeater

```
if (col >= repeaterWidth || row >= repeaterHeight) {  
    return;  
}
```

Given `repeaterWidth = 45`, `repeaterHeight = 45`

1 st block (0-31, 0-31)	Block fully used
2 nd block (0-31, 32-63)	Block 40% used
3 rd block (32-63, 0-31)	Block 40% used
4 th block (32-63, 32-63)	Block 16% used

In total: ~ 50% of the threads are unused.

Can be optimized by calculating the thread block size cleverly. However, 32x32 is optimal because it optimizes global memory loads transactions.

In the last three blocks, warp divergence is created (not all 32 threads in the warp follow the same path).

Also, remember, to maximize occupancy, we need `#total warps >> #of SM (36 for RTX 2070)`

Case 1 - Results

Host (basic)	870'000 μ s / iteration	
Host (indexed)	114'000 μ s / iteration	7.5x faster
Host (ptr increment)	99'600 μ s / iteration	8.7x faster
Host (repeater loop)	912'000 μ s / iteration	slower
GPU – with D-H and H-D copy	34'000 μ s / iteration	25x faster
GPU – without copy	2'030 μ s / iteration	430x faster
GPU – smem load	816'000 μ s / iteration	
GPU – repeater grid	3'200 μs / iteration	270x faster

Conditions:

SIZE_GLOBAL = 10240

SIZE_REPEATER = 100

SIZE_REPEATER = 128

GPU: 2.3ms GPU – rep grid : 1.4 ms

SIZE_REPEATER = 64

GPU: 2.3ms GPU – rep grid : 1.8 ms

SIZE_REPEATER = 32

GPU: 2.5ms GPU – rep grid: 7.3ms

SIZE_REPEATER = 16

GPU: 2.5ms GPU – rep grid: 10.1ms

General optimization guidelines

- Limit PCIe traffic. Limit global memory access
- Maximize utilization by:
 - Increasing thread-level parallelism (more threads “hide” away memory latency)
 - Decreasing latency (minimize global memory calls, use coalescing)
- Use little thread fencing (`__syncthreads` or `atomics`)

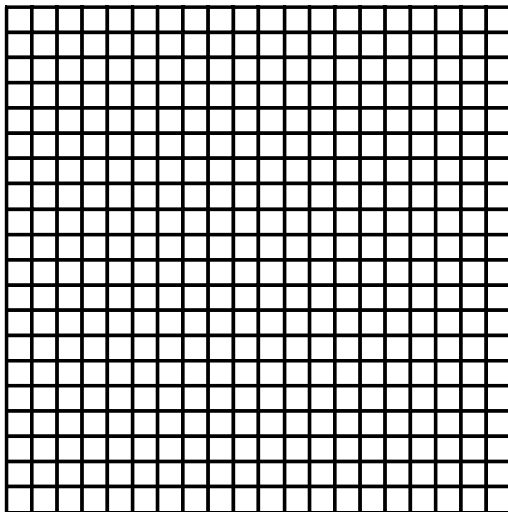
Parallelism required = Latency x Throughput

More complex stitching

Child tile
(trait = pattern)

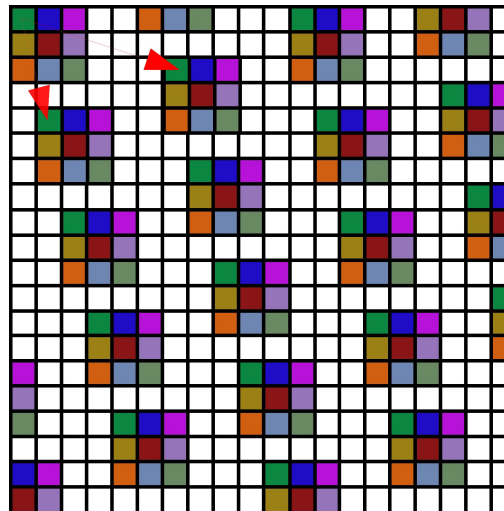


Target tile



Reduction

Repetition vector u, v



Here we have two possibilities:

Loop over the parent tile, or loop over all the possible (u,v) combinations

Using the same definitions:

```
typedTile<float, GPUMatrixAllocator<float>> repeater{ SIZE_REPEATER, SIZE_REPEATER };  
typedTile<float, GPUMatrixAllocator<float>> target{ SIZE_GLOBAL, SIZE_GLOBAL };
```

Case 2.1 — Loop over target (blending)

```
size_t index = 0; // Assumes we know how the data is placed
for (uint32_t y = 0; y < SIZE_GLOBAL; y++) {
    for (uint32_t x = 0; x < SIZE_GLOBAL; x++) {
        // Dot product over the normalized repetition vector
        const int32_t uComp = (int32_t)(dot(uNorm, 1, x, y));
        const int32_t vComp = (int32_t)(dot(vNorm, 1, x, y));

        // Accumulation
        float val = 0;
        // Number of samples for blending
        int16_t num = 0;

        // There might be more than one (u,v) to check for this position
        for (int32_t uR = uMin; uR <= uMax; uR++) {
            for (int32_t vR = vMin; vR <= vMax; vR++) {
                const int32_t startX = (uComp + uR) * u.x + (vComp + vR) * v.x;
                const int32_t xRep = x - startX;
                const int32_t startY = (uComp + uR) * u.y + (vComp + vR) * v.y;
                const int32_t yRep = y - startY;

                if (xRep < 0 || yRep < 0 || xRep >= SIZE_REPEATER || yRep >= SIZE_REPEATER) {
                    continue;
                }
                float locval = repeater.getPointXY(xRep, yRep);
                if (locval != 0) {
                    val += locval;
                    num++;
                }
            }
        }
        if (num > 0) {
            target.loadPoint(index, val / num);
        }
        index++;
    }
}
```

Loop over the whole frame

Find **u,v** for this position

Iterate over all possible overlaps

Boundary check

Calculate the repeater coordinates

Set global memory

Case 2.2 — Loop over (u,v), no blending

```
updateBoundaries(0, 0);  
updateBoundaries(SIZE_GLOBAL, 0);  
updateBoundaries(0, SIZE_GLOBAL);  
updateBoundaries(SIZE_GLOBAL, SIZE_GLOBAL);
```

Find min & max
boundaries

```
for (int32_t u = uMin; u <= uMax; u++) {  
    for (int32_t v = vMin; v <= vMax; v++) {  
        const int32_t startX = u * u.x + v * v.x;  
        const int32_t startY = u * u.y + v * v.y;
```

Find starting positions

```
        size_t repeaterIndex = 0;  
        for (uint32_t x = 0; x < SIZE_REPEATER; x++) {  
            for (uint32_t y = 0; y < SIZE_REPEATER; y++) {  
                if (startX + x < 0 || startY + y < 0 || startX + x >= SIZE_GLOBAL || startY + y >= SIZE_GLOBAL) {  
                    repeaterIndex++;  
                    continue;  
                }
```

Iterate over repeater

```
                const size_t index = target.calcIndex(startX + x, startY + y);  
                target.loadPoint(index, repeater.getPoint(repeaterIndex));  
                repeaterIndex++;  
            }  
        }  
    }  
}
```

Set the value to global
mem

Case 2.3 — GPU loop over target, no blending

```
void __global__ kernelRepeaterNonSquare(
    T* target,
    const uint32_t width, const uint32_t height,
    const T* repeater,
    const uint32_t repeaterWidth, const uint32_t repeaterHeight,
    const int2 u, const int2 v,
    const float2 uNorm, const float2 vNorm,
    int32_t uMin, int32_t uMax, int32_t vMin, int32_t vMax
) {
    const size_t x = blockIdx.x * blockDim.x + threadIdx.x;
    const size_t y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) { return; }
    const int32_t uComp = (int32_t)(dot(uNorm, 1, x, y)); // Integer clamp
    const int32_t vComp = (int32_t)(dot(vNorm, 1, x, y));
    float val = 0;
    int16_t num = 0;
    // If the repeater size is larger than the repetition vector, there is some blending
    for (int32_t uR = uMin; uR <= uMax; uR++) {
        for (int32_t vR = vMin; vR <= vMax; vR++) {
            const int32_t startX = (uComp + uR) * u.x + (vComp + vR) * v.x;
            const int32_t xRep = x - startX;
            const int32_t startY = (uComp + uR) * u.y + (vComp + vR) * v.y;
            const int32_t yRep = y - startY;
            // Boundary check
            if (xRep < 0 || yRep < 0 || xRep >= SIZE_REPEATER || yRep >= SIZE_REPEATER) {
                continue;
            }

            const float locval = repeater[getMatrixIndex(xRep, yRep, SIZE_REPEATER)];
            if (locval != 0) {
                val += locval;
                num++;
            }
        }
    }
    if (num > 0) {
        target[getMatrixIndex(x, y, width)] = val / num;
    }
}
```

Find min & max
boundaries

Find starting positions

Iterate over repeater

Set the value to global
mem

Case 2.4 — GPU loop over UV, no blending

```
void __global__ kernelRepeaterUVLoop(  
    T* target,  
    const uint32_t width, const uint32_t height,  
  
    const T* repeater,  
    const uint32_t repeaterWidth, const uint32_t repeaterHeight,  
  
    const int2 u, const int2 v,  
    const int32_t minU, const int32_t minV  
) {
```

```
    const int32_t gridU = blockIdx.x * blockDim.x + threadIdx.x + minU;  
    const int32_t gridV = blockIdx.y * blockDim.y + threadIdx.y + minV;
```

Grid is now over (u,v)

```
    const int32_t startX = gridU * u.x + gridV * v.x;  
    const int32_t startY = gridV * u.y + gridV * v.y;
```

(Look for the starting point)

```
    size_t indexRepeater = 0;
```

```
    for (uint32_t y = 0; y < SIZE_REPEATER; y++) {  
        for (uint32_t x = 0; x < SIZE_REPEATER; x++) {  
            float repValue = repeater[indexRepeater];  
            indexRepeater++;
```

Loop over repeater

```
            if (startX + x < 0 || startY + y < 0 || startX + x >= SIZE_GLOBAL || startY + y >= SIZE_GLOBAL) {  
                continue;  
            }  
        }  
    }
```

```
    const size_t index = getMatrixIndex(startX + x, startY + y, SIZE_GLOBAL);  
    if (repValue != 0) {  
        target[index] = repValue;  
    }  
}
```

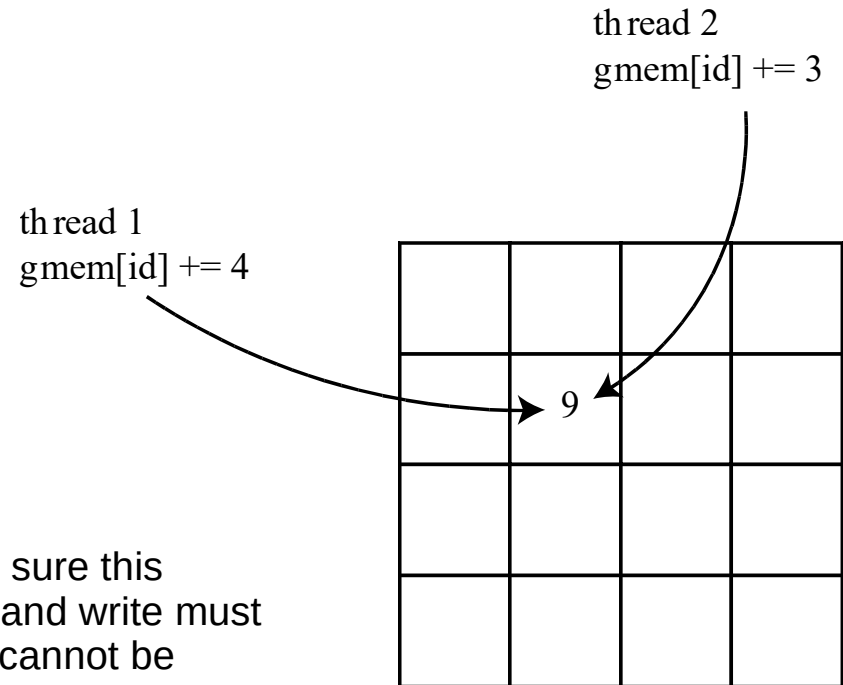
Set the repeater value the proper position

```
    }  
}
```


Atomic operations - reminder

- 1st possible output
 - Thread 1 read (9)
 - Thread 1 write (13)
 - Thread 2 read (13)
 - Thread 2 write (16)
- 2nd possible output
 - Thread 2 read (9)
 - Thread 2 write (12)
 - Thread 1 read (12)
 - Thread 1 write (16)
- 3rd possible output
 - Thread 1 read (9)
 - Thread 2 read (9)
 - Thread 2 write (12)
 - Thread 1 write (13)
- 4th possible output
 - ... you see the point

Atomic functions make sure this doesn't happen. Read and write must occur "atomically", i.e. cannot be separated



Case 2.4 – modified

You might have noticed this caveat:

```
target[index] = repValue;
```

- 1) Multiple threads potentially write at the same global memory location.
- 2) Introduces unpredictability.
- 3) In the general case includes blending and we should account for that

Replace

```
target[index] = repValue;          //PTX: ST.E.SYS [R6], R25    (Corresponds to global storage)
```

By the atomic version

```
atomicExch(target + index, repValue);    // PTX: CALL.ABS.NOINC 0x0  (Calls something on the stack...)  
// Or, if we want blending  
// atomicAdd(target + index, repValue);
```

And what a surprise, execution time goes from 9.8ms down to 6.3ms.
(note how long the kernel spends writing memory, at least 1/3 of the total time)

How come is an atomic function (which should have thread fences) faster than a global store ? PTX doesn't indicate mu

Atomic function: 40 register used, corresponds to 32 warp / SM

Direct save: 40 register used, corresponds to 32 warp / SM

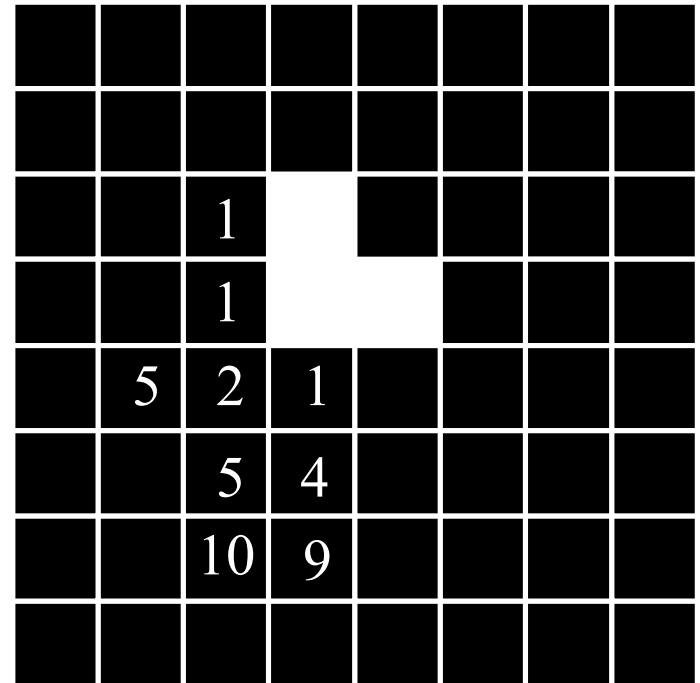
Occupancy is the same, so it cannot explain the difference in timing

The official explanation from Nvidia is that “atomics are performed directly in the memory controller”

Not sure I'm convinced. To be continued...

Case 3: distance calculation

- Given a binary matrix, we want to fill target calculate a profile $z = f(d)$ where d is the minimum distance to a white pixel.
- $f(z)$ can be complex, non-monotonic, and may not be evaluated at compile-time.
- Intermediate mapping using $a := dx^2 + dy^2$, and then use $z = f(a)$.
- Two strategies
 - Look for white pixel, then apply a kernel. Requires atomics (multiple kernels write in the same gmem location)
 - Look for black pixel, then look for the closest white pixel. No need for a kernel.



Case 3.1 — Find white pixels

```
__global__ void kernelDistance(  
    const T* source,  
    const uint32_t width,  
    const uint32_t height,  
    U* distance,  
    const int32_t distanceMax  
) {  
    const int32_t col = blockIdx.x * blockDim.x + threadIdx.x;  
    const int32_t row = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if (col >= width || row >= height) {  
        return;  
    }  
  
    const size_t index = getMatrixIndex<size_t>(col, row, width);  
  
    if (source[index] == 0) {  
        return;  
    }  
  
    for (int32_t y = -distanceMax; y < distanceMax; y++) {  
        for (int32_t x = -distanceMax; x < distanceMax; x++) {  
            const int32_t colDist = col + x;  
            const int32_t rowDist = row + y;  
  
            if (colDist < 0 || rowDist < 0 || colDist >= width || rowDist >= height) {  
                continue;  
            }  
            const size_t indexDistance = getMatrixIndex<size_t>(colDist, rowDist, width);  
            atomicMin((U*)(distance + indexDistance), (U)(x * x + y * y));  
        }  
    }  
}
```

Look exclusively for not(0x00)

Iterate over the square

Update the new distance for
that element in the square

Case 3 - results

White (unoptimized)	251'000 μ s / iteration	
White (block spread)	246'000 μ s / iteration	About the same

Conditions:

DISTANCE_MAX 15

Case 3.1 bis — Add pre-atomic distance check

Replace

```
atomicMin((U*)(distance + indexDistance), (U)(x * x + y * y));    // Runs in 230ms
```

by

```
const U distVal = (U)(x * x + y * y);                               // Runs in 130ms
if (distance[indexDistance] < distVal) {
    continue;
}
atomicMin((U*)(distance + indexDistance), distVal);
```

No need to perform the atomic minimum if the value that is read is already lower.
One additional global memory read to potentially save an atomic operation.

Gain / loss will depend on how many atomics are saved, and thus on the input mask.

Case 3 - results

White (unoptimized)	251'000 μ s / iteration	
White (block spread)	246'000 μ s / iteration	About the same
White (distance check)	125'000 μs / iteration	2x faster

Conditions:

DISTANCE_MAX 15

Case 3.1 tris — Add neighbour check

If one white pixel is already surrounded by 4 white pixels, its net effect is nil. Save the double-loop

```
// If we're inside the boundaries and surrounded by 4 non-black pixels, then no need to execute the matrix patching
if (col > 0 && col < width - 1 && row > 0 && row < height - 1 &&
    source[index - 1] > 0 && source[index + 1] > 0 && source[index - width] > 0 && source[index + width] > 0) {
    distance[index] = 0;
    return;
}
```

Computation time from 240ms down to 65ms (factor 4 !)

Case 3 - results

White (unoptimized)	251'000 μ s / iteration	
White (block spread)	246'000 μ s / iteration	About the same
White (distance check)	125'000 μ s / iteration	2x faster
White (neighbour check)	49'000 μs / iteration	5x faster

Conditions:

DISTANCE_MAX 15

Case 3.3 tert — improved neighbor check

If for a pixel at (x,y) is white, and pixel at (x,y-1) is also white, there's no need to check any pixel whose coordinate is $y < y$.

Therefore we can use:

```
int32_t xFrom = -distanceMax, xTo = distanceMax, yFrom = -distanceMax, yTo = distanceMax;
if (col == 0 || source[index - 1] != 0x00) xFrom = 0;
if (col == width - 1 || source[index + 1] != 0x00) xTo = 0;
if (row == 0 || source[index - width] != 0x00) yFrom = 0;
if (row == height - 1 || source[index + width] != 0x00) yTo = 0;

for (int32_t y = yFrom; y <= yTo; y++) {
    for (int32_t x = xFrom; x <= xTo; x++) {
        //...
        // Rest of the kernel
    }
}
```

Which restricts the boundaries of the loop when neighboring pixels are found.

Case 3 - results

White (unoptimized)	251'000 μ s / iteration	
White (block spread)	246'000 μ s / iteration	About the same
White (distance check)	125'000 μ s / iteration	2x faster
White (neighbour check)	49'000 μ s / iteration	5x faster
White (improved neighbour check)	9'800 μs / iteration	26x faster

Conditions:

DISTANCE_MAX 15

Case 3.1 — Find black pixels

```
template<typename T, typename U>
__global__ void kernelDistanceBlack(
    const T* source,
    const uint32_t width,
    const uint32_t height,
    U* distance,
    const int32_t distanceMax
) {
    const int32_t col = blockIdx.x * blockDim.x + threadIdx.x;
    const int32_t row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col >= width || row >= height) return;

    const size_t index = getMatrixIndex<size_t>(col, row, width);
    if (source[index] != 0) {
        distance[index] = 0;
        return;
    }

    U minDistance = distanceMax * distanceMax;

    for (int32_t y = -distanceMax; y < distanceMax; y++) {
        for (int32_t x = -distanceMax; x < distanceMax; x++) {
            const U dist = (U)(x * x + y * y);
            if (dist > minDistance) continue;

            const int32_t col2 = col + x;
            const int32_t row2 = row + y;

            if (col2 < 0 || row2 < 0 || col2 >= width || row2 >= height) continue;
            if (source[getMatrixIndex<size_t>(col2, row2, width)] != 0)
                minDistance = min(minDistance, dist);
        }
    }

    distance[index] = minDistance;
}
```

Usually boundary checks

Look exclusively for 0x00

Prepare min distance

Iterate over the square

Bypass if distance is already smaller

Frame check

Calculate the new distance

Case 3 - results

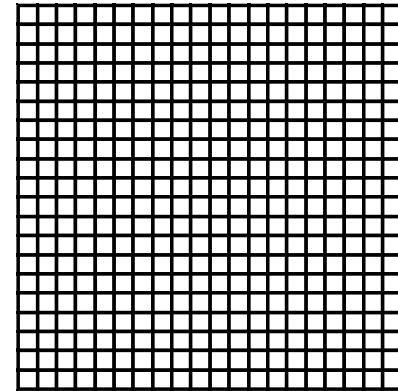
White (unoptimized)	251'000 μ s / iteration	
White (block spread)	246'000 μ s / iteration	About the same
White (distance check)	125'000 μ s / iteration	2x faster
White (neighbour check)	49'000 μ s / iteration	5x faster
White (improved neighbour check)	9'800 μ s / iteration	26x faster
Black	90'000 μs / iteration	

Conditions:

DISTANCE_MAX 15

Parallel reduction

- Matrix reduction to a single value
 - e.g. surface or volume of a hole
 - e.g. maximum height
 - e.g. projected surface
 - ...

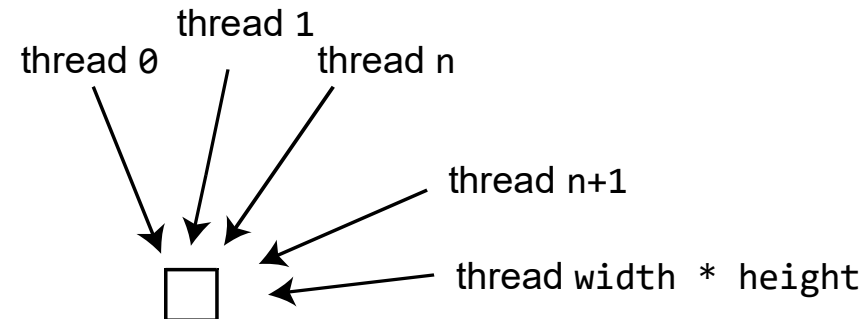


- As an example, we'll calculate the integral over the matrix (sum all elements)

The naive approach

- Create a global memory of size 1 and increment it with `atomicAdd` from global memory

```
__global__ void kernelSumAll(  
    const T* source,  
    const uint32_t w,  
    const uint32_t h,  
    T* sum  
) {  
    const int32_t col = blockIdx.x * blockDim.x + threadIdx.x;  
    const int32_t row = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if (col >= w || row >= h) {  
        return;  
    }  
  
    const size_t index = getMatrixIndex<size_t>(col, row, w);  
    atomicAdd((T*)sum, (T)source[index]);  
}
```



- + Coalesced global memory access
- + Low register number (more threads per block)
- Serialization of atomics in global memory
- Low memory / thread ratio (which means memory latency may be limiting)

The less naive approach

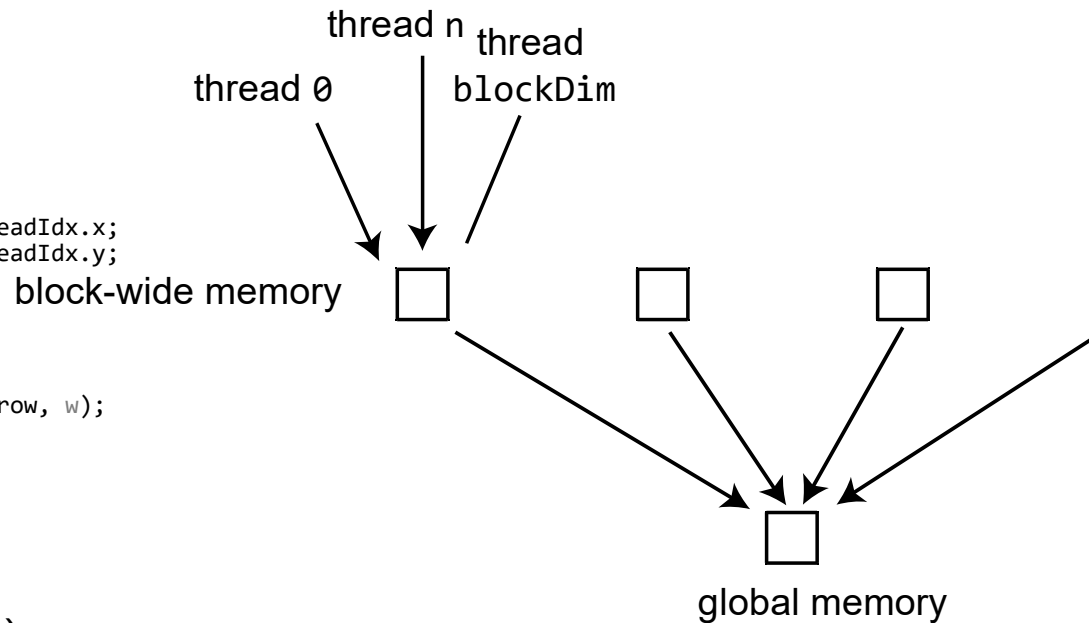
- Use privatization to reduce atomic serialization

```
template<typename T>
__global__ void kernelSumAll_shared(
    const T* source,
    const uint32_t w,
    const uint32_t h,
    T* sum
) {
    __shared__ T sSum;
    if (threadIdx.x == 0 && threadIdx.y == 0) {
        sSum = 0;
    }
    __syncthreads();

    const int32_t col = blockIdx.x * blockDim.x + threadIdx.x;
    const int32_t row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col >= w || row >= h) {
        return;
    }

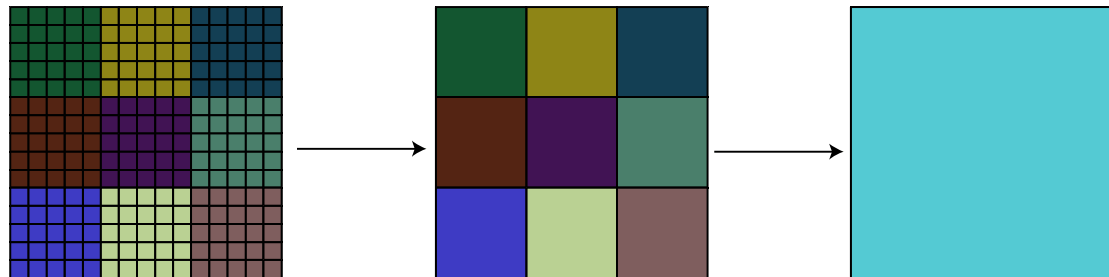
    const size_t index = getMatrixIndex<size_t>(col, row, w);
    atomicAdd_block(&sSum, source[index]);
    __syncthreads();
    if (threadIdx.x == 0 && threadIdx.y == 0)
        atomicAdd(sum, sSum);
}
```



- + Coalesced global memory access
- + Low register number (more threads per block)
- + Reduced serialization of atomics
- Low memory / thread ratio (which means memory latency may be limiting)
- Atomics in shared memory are software-implemented on older architectures

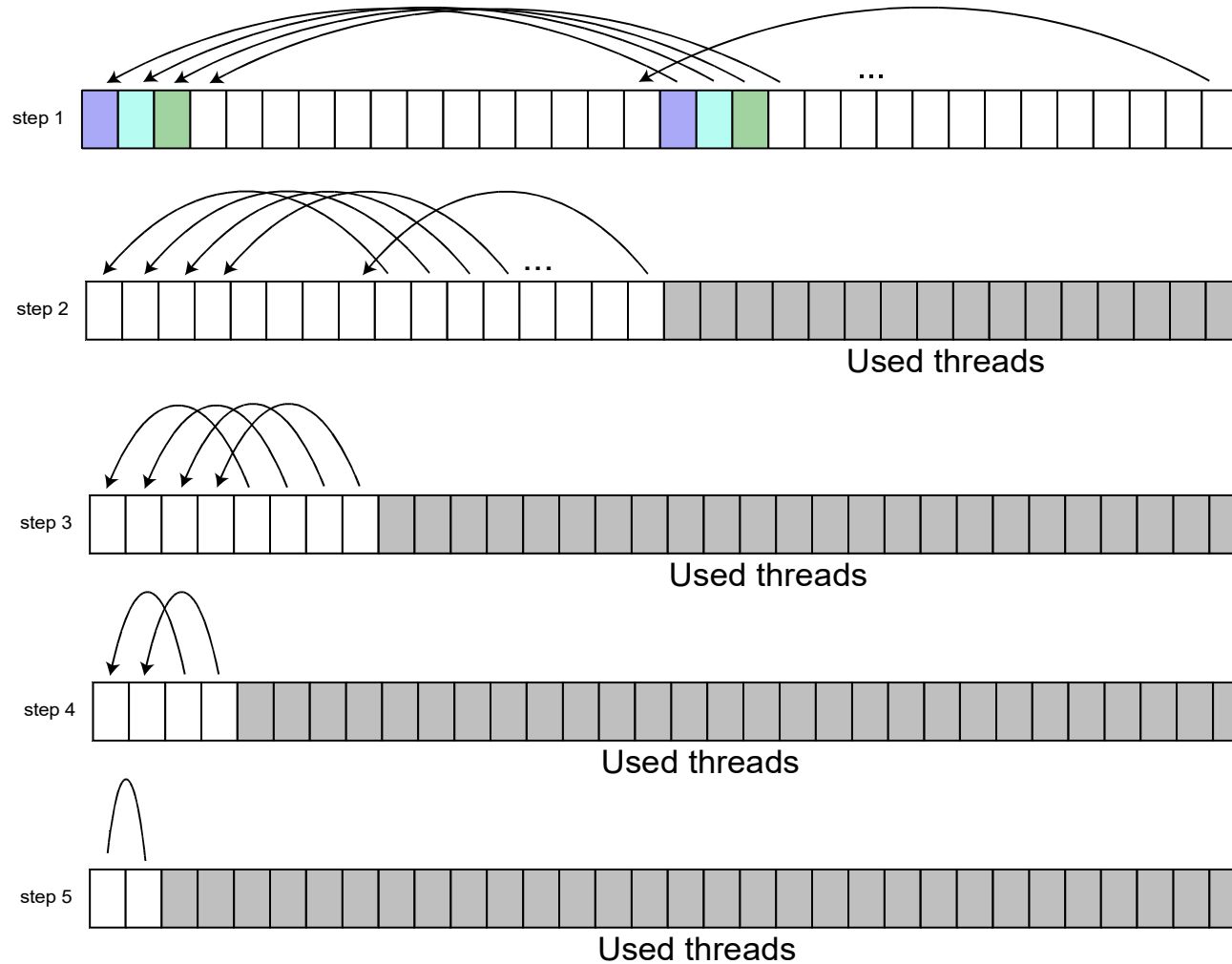
The clever (= best ?) approach

- Parallel reduction
- Each block of 1024 threads are reduced into a global memory location.
- Iteration until we get a matrix 1x1
- Question is: how to reduce each block ?



The basic approach

- Each thread reduces another memory location
- Iterate until the last reduction



The basic approach – step 1

- Each thread reduces another memory location
- Iterate until the last reduction

```
// Start loading shared memory with content of the thread block
extern __shared__ T sMem[];

// Reduced for the block data is offseted in the same shared memory blockLength
const uint32_t blockLength = blockDim.x * blockDim.y;

// Sequential IDs are important. also, x is major, because of how a warp is arranged
const uint32_t threadId = threadIdx.x + blockDim.x * threadIdx.y;

// Get matrix coordinates
const uint32_t col = blockIdx.x * blockDim.x + threadIdx.x;
const uint32_t row = blockIdx.y * blockDim.y + threadIdx.y;

if (col < width && row < height) { // Copy global memory into shared memory
    sMem[threadId] = input[getMatrixIndex<size_t>(col, row, width)];
}
else {
    sMem[threadId] = 0;
}

__syncthreads();
```

The basic approach – step 2

- Reduce memory offseted by the reduction iterator
- The last thread writes in global memory

```
for (unsigned int s = blockDim / 2; s >= 1; s >>= 1) {  
    if (threadId < s) {  
        sMem[threadId] = sMem[threadId] + sMem[threadId + s]; // Reduction step  
    }  
    __syncthreads();  
}  
  
// For the one thread into which the data has been reduced  
if (threadIdx.x == 0 && threadIdx.y == 0) {  
    // Note how we are filling a global matrix of size (gridDim.x, gridDim.y) at the position  
    (blockIdx.x, blockIdx.y)  
    reduced[blockIdx.y * gridDim.x + blockIdx.x] = sMem[0];  
}
```

Optimizing the parallel reduction

- When $s \leq 32$, there's only one active warp. We can use this to unroll the last 6 loops:

```
for (unsigned int s = blockLength / 2; s > 32; s >>= 1) {
    if (threadId < s) {
        sMem[threadId] = sMem[threadId] + sMem[threadId + s]; // Reduction step
    }
    __syncthreads();
}

// Applies to threads 0 to 31, i.e. the ones in the last warp.
if (threadId < 32) { // First warp only, but still needs to be synchronized
    if (blockLength >= 64) {
        sMem[threadId] += sMem[threadId + 32]; __syncwarp();
    }
    if (blockLength >= 32) {
        sMem[threadId] += sMem[threadId + 16]; __syncwarp();
    }
    if (blockLength >= 16) {
        sMem[threadId] += sMem[threadId + 8]; __syncwarp();
    }
    if (blockLength >= 8) {
        sMem[threadId] += sMem[threadId + 4]; __syncwarp();
    }
    if (blockLength >= 4) {
        sMem[threadId] += sMem[threadId + 2]; __syncwarp();
    }
    if (blockLength >= 2) {
        sMem[threadId] += sMem[threadId + 1];
    }
}
```

Optimizing the parallel reduction

- We can give more work to each thread by serially reducing some data.
- More serial work per thread can be used to hide memory latency
- Here, we serially reduce `loadNum - 1` additional rows
- Note that warp-wide, gmem access is still coalesced

```
if (col < width && row < height) { // Copy global memory into shared memory
    sMem[threadId] = input[getMatrixIndex<size_t>(col, row, width)];
    if constexpr (numLoads > 1) {
        loadIntoSMem<T, numLoads>(
            &(sMem[threadId]),
            input,
            col,
            row + blockDim.y,
            blockDim.y,
            width,
            height
        );
    }
}
```

```
template<typename T, uint32_t loadNum>
__device__ inline __forceinline__ void loadIntoSMem(
    volatile T* sMem,
    const T* input,
    const uint32_t col,
    const uint32_t row,
    const uint32_t strideY,
    const uint32_t width,
    const uint32_t height
) {
    if (row < height) {
        *sMem += input[getMatrixIndex<size_t>(col, row, width)];
    }

    if constexpr (loadNum > 2) {
        loadIntoSMem<T, loadNum - 1>(sMem, input, col, row + strideY,
        strideY, width, height);
    }
}
```

Parallel reduction - results

Global atomics	1'600 μ s / iteration	155 GB / s
Global atomics – sequential addressing	1'400 μ s / iteration	177 GB / s
Shared atomics	1'800 μ s / iteration	138 GB / s
Parallel reduction standard	4'980 μ s / iteration	50 GB / s
Parallel reduction, 2x load	5'000 μ s / iteration	49.7 GB / s
Parallel reduction, 2x load + unrolling	3'500 μ s / iteration	71 GB / s
Parallel reduction, 4x load + unrolling	3'700 μ s / iteration	67 GB / s

Dead time (no gmem load): ~1.2ms

Theoretical bandwidth:

256 bit mem interface, 1.4 GHz = **448 GB/s**

Possible enhancement, switch to 1D addressing

Conditions:

T = unsigned int