CUDA by examples: delay and sum

Piotr Jarosik, Marcin Lewandowski, Billy Yiu

IUS 2023 GPU short course, Canada, Montreal

Agenda

- Introduction
- Delay
- Sum
- Linear array
- Matrix array
- Row Column Array

Motivation

Why delay and sum?

- *GPU-based **ultrasound** signal processing*, **IUS** 2023
- fundamental part of ultrasound signal processing
- combination of two more general problems: interpolation and reduction
- good practical CUDA example

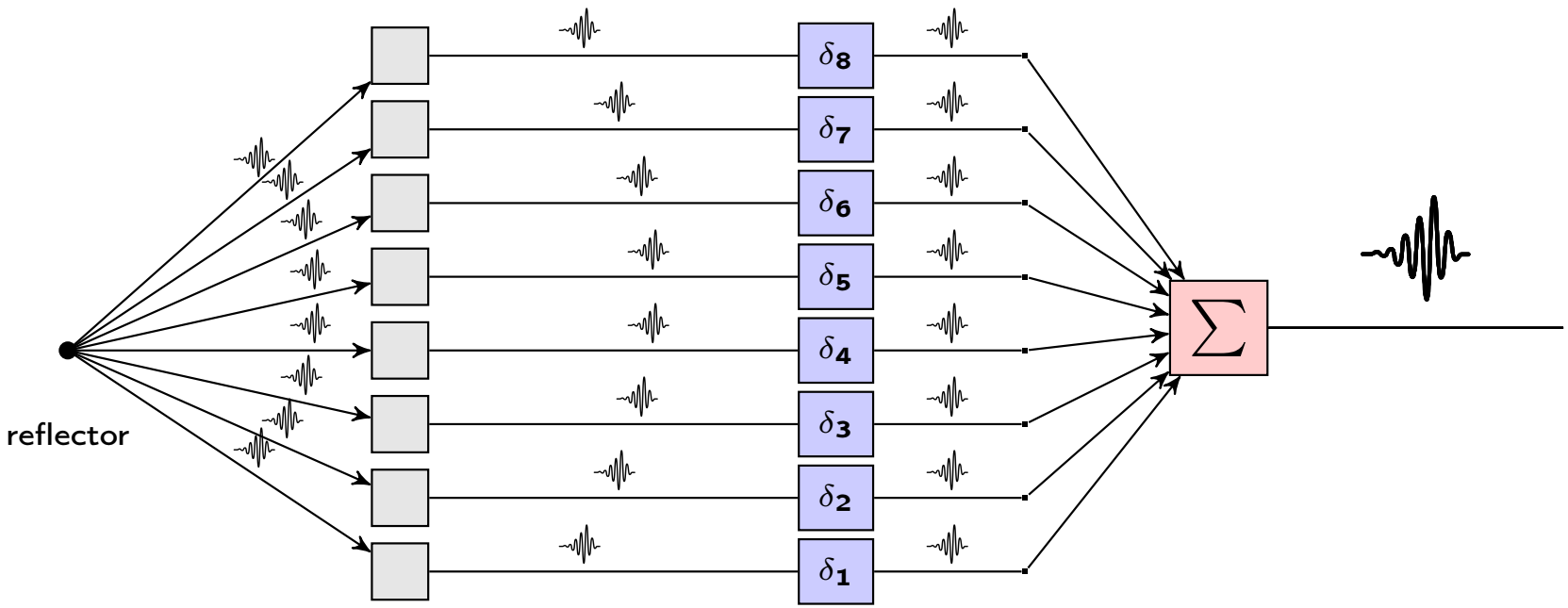
Objectives

Practical introduction to:

- CUDA kernel syntax (grids, blocks, global, device kernels...)
- CUDA memory model (global, shared, texture)
- CUDA core performance guidelines

For more examples, see also:
https://github.com/lab4us/gpu-short-course

Delay and sum



Delay and sum

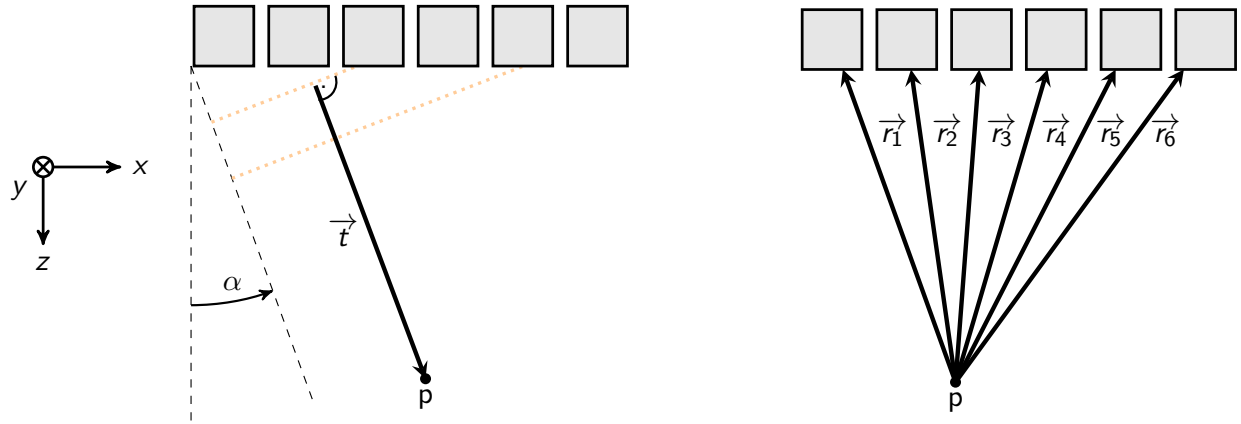
$$g(t,x,z) = \sum_{r=1}^{N_r} f(t,r,\delta(t,r,x,z)) \tag{1}$$

where:

- g: low resolution images (LRI)
- f: input signal (e.g. raw channel data)
- t: transmit index (e.g. plane wave transmit number)
- r: receiver index, N_r: total number of receivers
- x, z: output image x, z coordinates
- $\delta(t,r,x,z)$: beamforming delays

To get the final B-mode image: compound, envelope, logarithmic compression.

Plane wave imaging



$$\delta(t, r, x, z) = (z\cos(\alpha(t)) + x\sin(\alpha(t)) + \sqrt{(x - l(r))^2 + z^2})\frac{f_s}{c}$$

where:

- $l(r)$ is the position of probe element r (*azimuth*),
- $\alpha(t)$ is an angle in transmit t

Delay and sum decomposition

How we interpret delay and sum?

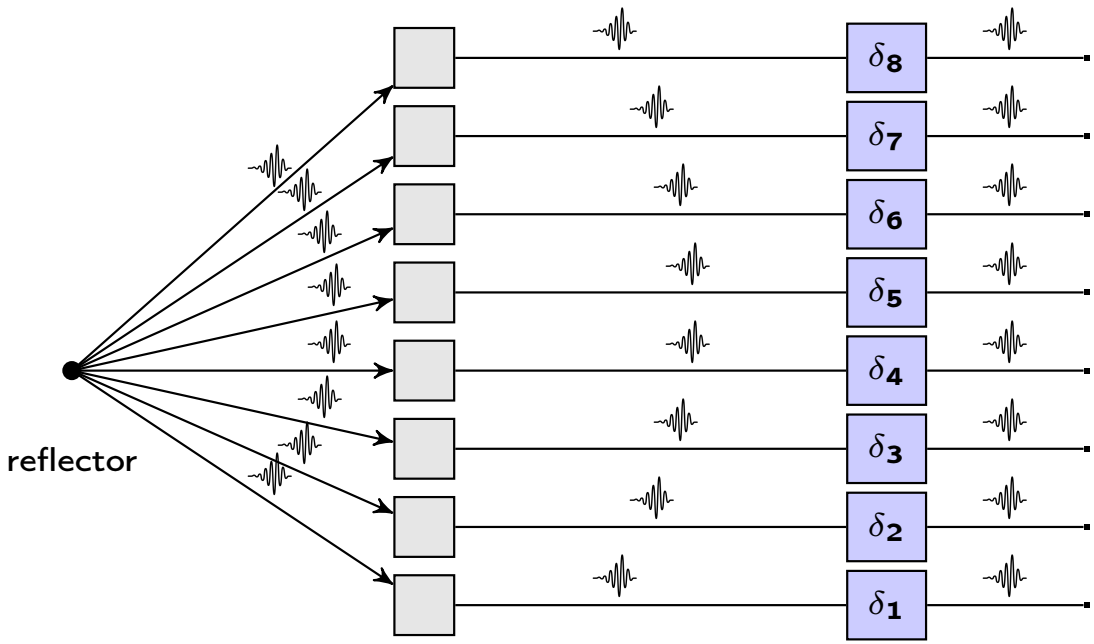
- delay $\equiv$ interpolation (signal *sampling* with delays)

$$g(t, r, x, z) \leftarrow f(t, r, \delta(t, r, x, z))$$

- sum $\equiv$ just sum :) (an example of data *reduction*)

$$h(t, x, z) \leftarrow \sum_{r=1}^{N_r} g(t, r, x, z)$$

Delay

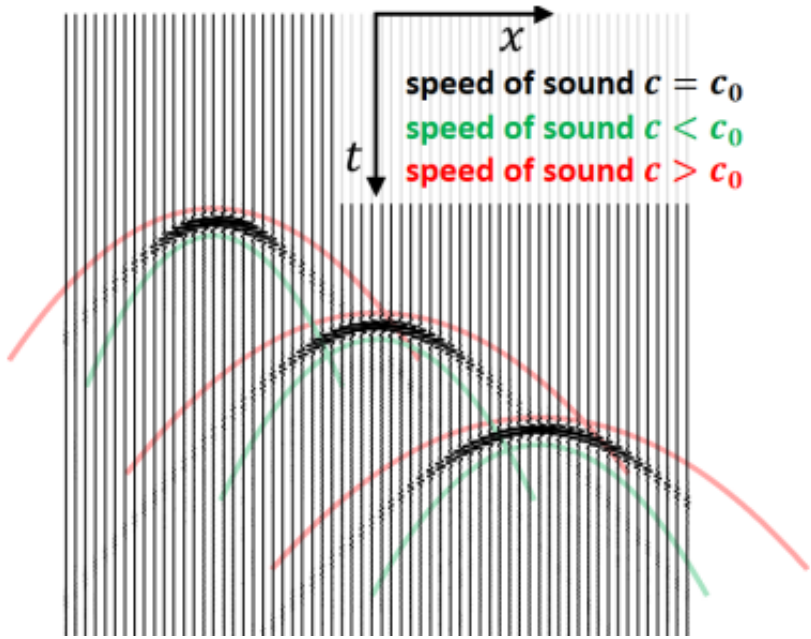


$$g(t, r, p) \leftarrow f(t, r, \delta(t, r, p))$$

where $p = (x, z)$.

Delay ≡ interpolation

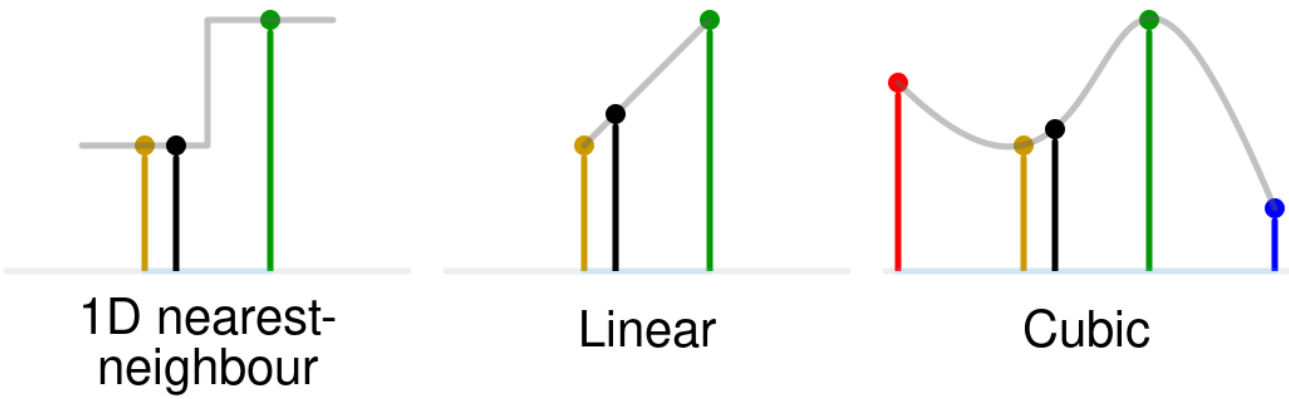
- "Delay" is the data sampling part of processing.
- The signal is sampled according to the beamforming delays δ.



source: So you think you can DAS? Vincent Perrot et al., 2021

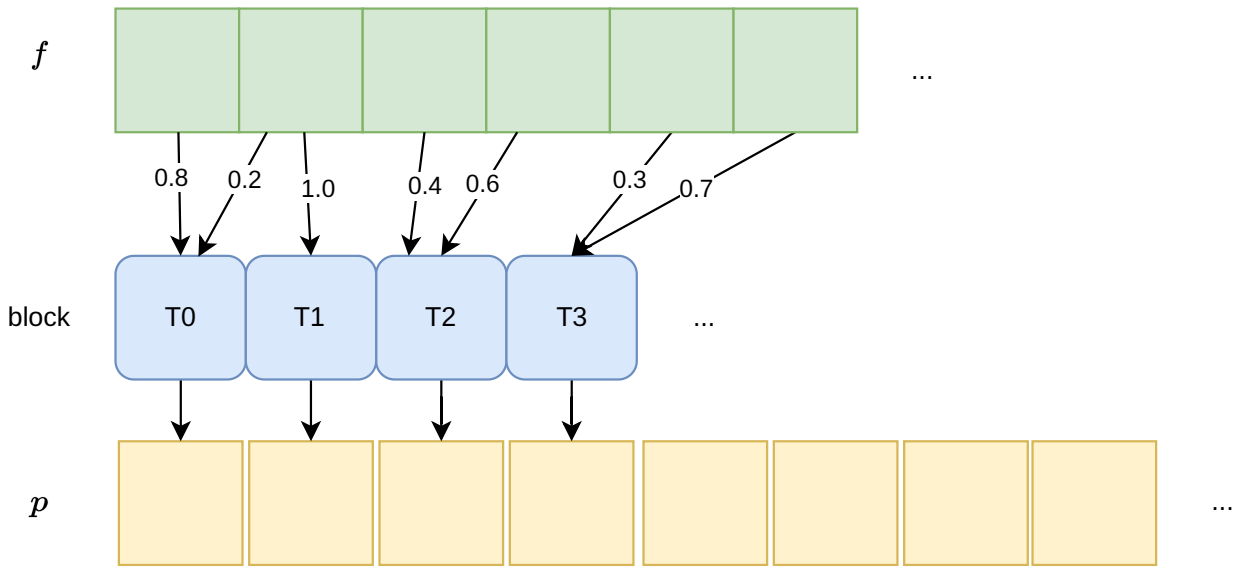
Delay ≡ interpolation

- $\delta :\rightarrow \mathrm{R}$, δ can be a floating-point value
- our raw channel data is a discretized signal (sampled with finite sampling frequency)
- we have to **interpolate** in case δ points somewhere between samples
  - it would be also good to **extrapolate** to reasonable value, e.g. 0



source: https://commons.wikimedia.org/

CUDA: interpolation kernel – block of threads



CUDA: 1D nearest-neighbor interpolation

```
__global__
void interpolate1d(
    float *output, float *input, float *points,
    size_t np, size_t ns
) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;

    if(x >= np) {
        return;
    }

    int point = int(roundf(points[x]));
    if(point < ns) {
        output[x] = input[point];
    }
    else {
        return 0.0f;
    }
}
```

CUDA: 1D nearest-neighbor interpolation

```
def interpolate1d(input_array, points):
    interpolate1d_kernel = load_cuda_kernel("1_1_interpolate1d.cc")
    n_samples, n_points = len(input_array), len(points)

    block = (256, )
    grid = (math.ceil(n_points/256), )

    output_array = cp.zeros((n_points, ), dtype=np.float32)
    interpolate1d_kernel(
        grid, block,
        args=(
            output_array,
            cp.asarray(input_array).astype(cp.float32),
            cp.asarray(points).astype(cp.float32),
            n_points, n_samples
        )
    )
    return output_array
```

CUDA: 1D linear interpolation

```
__global__
void interpolate1d(
    float *out, float *in, float *points,
    size_t np, size_t ns
) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    if(x >= np) {
        return;
    }

    float sample = points[x];
    float weight = modff(sample, &sample);

    int sample_nr = int(sample);
    if(sample_nr < ns-1) {
      return in[sample_nr]*(1-weight) + in[sample_nr+1]*weight;
    }
    else {
      return 0.0f;
    }

}
```

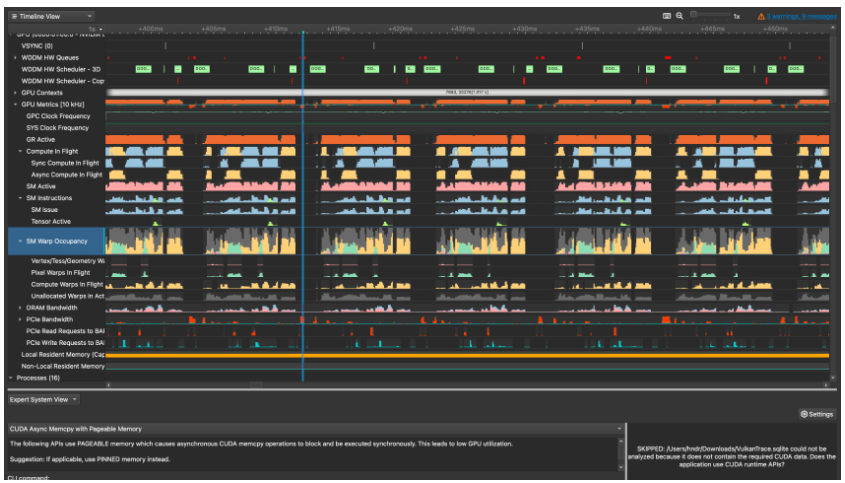
CUDA: 1D linear interpolation

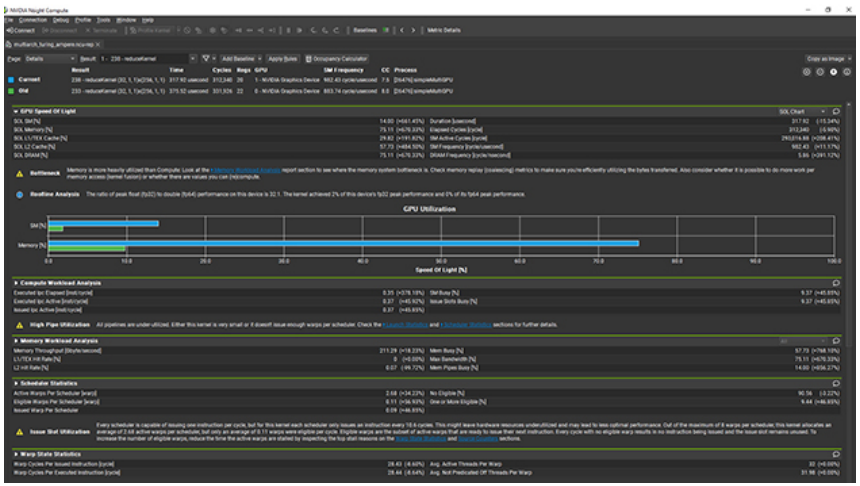
```
__forceinline__
__device__
float interp_linear(float *in, float sample, int ns) {
    float weight = modff(sample, &sample);
    int sample_nr = int(sample);
    if(sample_nr < ns-1) {
      return in[sample_nr]*(1-weight) + in[sample_nr+1]*weight;
    }
    else {
      return 0.0f;
    }
}

__global__
void interpolate1d(float *out, float *in, float *points, size_t np, size_t ns) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    if(x >= np) {
        return;
    }
    out[x] = interp_linear(in, points[x], ns);
}
```

Profiler

How can we measure kernel's performance?
We can analyse time and memory requirements using **code profiler**.
NVIDIA provides two profiling tools:

- Nsight Systems: overall performance of the CUDA streams
- Nsight Compute: detailed analysis of a CUDA kernel





CUDA: 1D linear interpolation

```
nsys profile --stats=true --trace cuda python 1_1_interpolate1d.py
```

NVIDIA Geforce 3700 Mobile NVIDIA GPU 3700 RTX
Results for 2^{15} samples:

```
...
Generating '/tmp/nsys-report-425c.qdstrm'
[1/6] [========================100%] report1.nsys-rep
[2/6] [========================100%] report1.sqlite
...
[4/6] Executing 'cuda_gpu_kern_sum' stats report
 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)            Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  --------------------------
     49,0          184 673         10  18 467,0  18 464,0    18 336    18 624        109,0  interpolate1d
```

CUDA: 1D linear interpolation for DAS

Now let's move to the *layered* version of our kernel.
We will use it in DAS implementation.
The *layered* kernel should perform 1D interpolation for each t, r and p in parallel.

$$g(t, r, p) \leftarrow f(t, r, \delta(t, r, p))$$

CUDA: 1D linear interpolation – DAS

Input array: $(N_t, N_r, N_s) = (3, 128, 8192)$
Output array: $(N_t, N_r, N_p) = (3, 128, 256 * 1024)$
Thread block: $(Rx, Tx, Points)$

CUDA: 2D linear interpolation – first attempt

```
__global__
void interpolate1d_layered(
    float *out, float *in, float *points,
    size_t nt, size_t ne, size_t ns, size_t np
) {
    int e = blockIdx.x*blockDim.x + threadIdx.x;
    int t = blockIdx.y*blockDim.y + threadIdx.y;
    int p = blockIdx.z*blockDim.z + threadIdx.z;

    if(p >= np || e >= ne || t >= nt) {
        return;
    }

    // input  [t, e, 0]
    size_t in_i = (ne*ns)*t +  (ns)*e;
    // output [t, e, p]
    size_t out_i = (ne*np)*t + (np)*e + p;

    out[out_i] = interp_linear(&in[in_i], points[out_i], ns);
}
```

CUDA: 2D linear interpolation – first-attempt

Step	Time (median, ms)
1D interpolation - first attempt	147

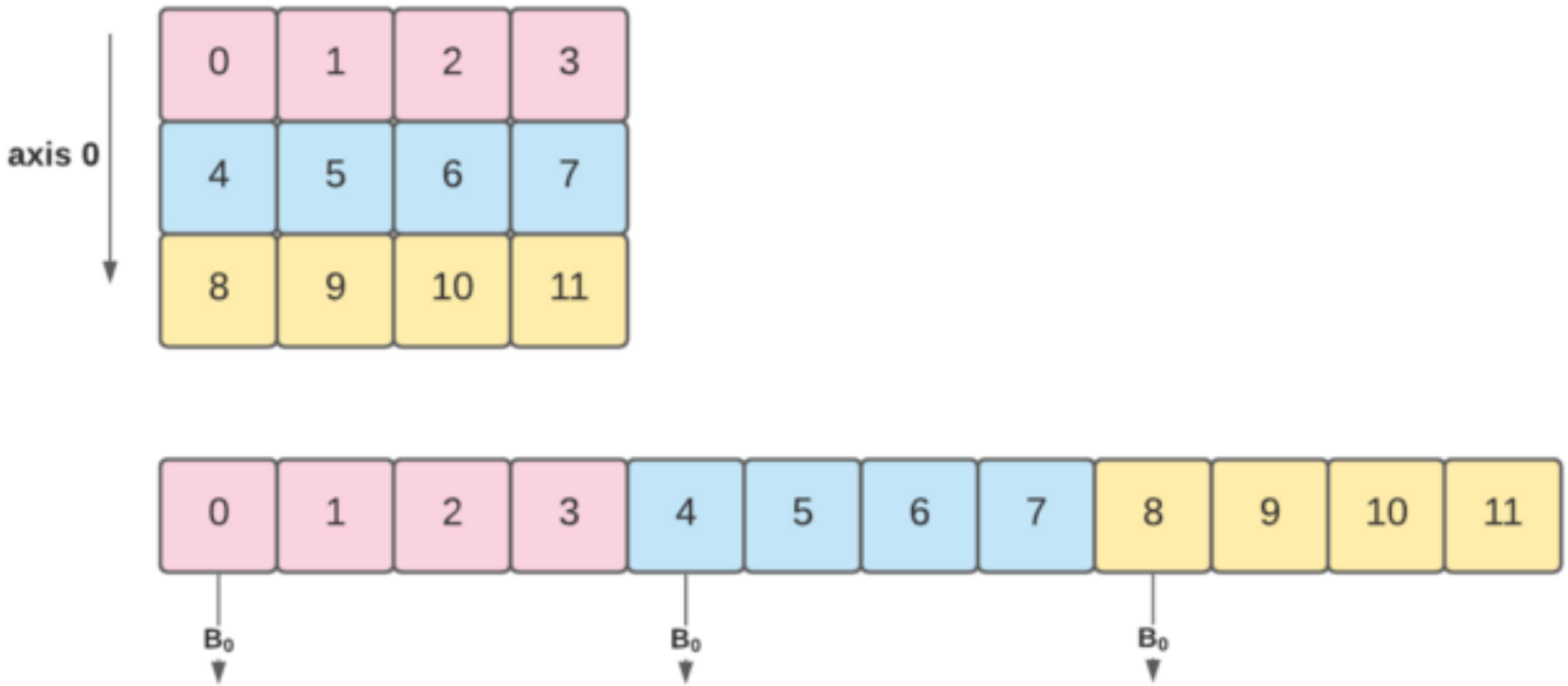
CUDA: 2D linear interpolation – first-attempt

Let's check global memory store efficiency:

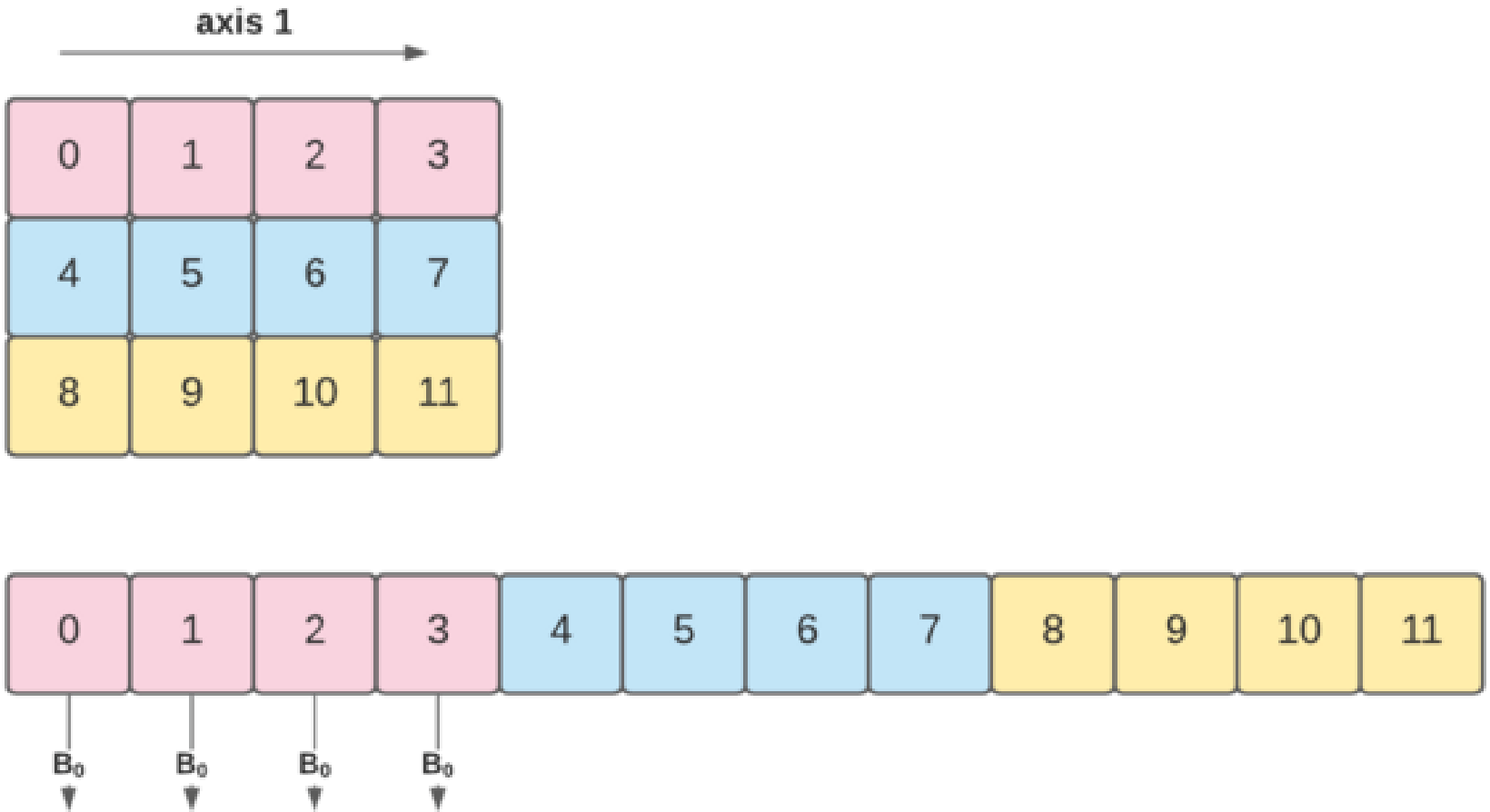
```
ncu
--target-processes all
--metrics
     smsp__sass_average_data_bytes_per_sector_mem_global_op_st
-f python 1_2_interpolate1d_layered_1st_attempt.py
```

```
interpolate1d_layered (16, 1, 32768)x(8, 8, 8), Context 1, Stream 7, Device 0, CC 8.6
Section: Command line profiler metrics
---------------------------------------------------------------- ----------- ------------
Metric Name                                                      Metric Unit Metric Value
---------------------------------------------------------------- ----------- ------------
smsp__sass_average_data_bytes_per_sector_mem_global_op_st.max_rate byte/sector           32
smsp__sass_average_data_bytes_per_sector_mem_global_op_st.pct                %        12,50
smsp__sass_average_data_bytes_per_sector_mem_global_op_st.ratio    byte/sector            4
---------------------------------------------------------------- ----------- ------------
```

Memory coalescing



Memory coalescing



CUDA: 1D linear interpolation – write coalescing

From:

```
int e = blockIdx.x*blockDim.x + threadIdx.x;
int t = blockIdx.y*blockDim.y + threadIdx.y;
int p = blockIdx.z*blockDim.z + threadIdx.z;
```

To:

```
int p = blockIdx.x*blockDim.x + threadIdx.x;
int e = blockIdx.y*blockDim.y + threadIdx.y;
int t = blockIdx.z*blockDim.z + threadIdx.z;
```

And Thread block: $(Points, Rx, Tx)$

CUDA: 2D linear interpolation – write-coalescing

Step	Time (median, ms)
1D interpolation - first attempt	147
1D interpolation - write coalescing	51

CUDA: 2D linear interpolation – write-coalescing

```
ncu
--target-processes all
--metrics
     smsp__sass_average_data_bytes_per_sector_mem_global_op_st
-f python 1_3_interpolate1d_layered_improved.py
```

```
interpolate1d_layered (32768, 16, 1)x(8, 8, 8), Context 1, Stream 7, Device 0, CC 8.6
Section: Command line profiler metrics
---------------------------------------------------------------------- ----------- ------------
Metric Name                                                            Metric Unit Metric Value
---------------------------------------------------------------------- ----------- ------------
smsp__sass_average_data_bytes_per_sector_mem_global_op_st.max_rate byte/sector           32
smsp__sass_average_data_bytes_per_sector_mem_global_op_st.pct                 %          100
smsp__sass_average_data_bytes_per_sector_mem_global_op_st.ratio    byte/sector           32
---------------------------------------------------------------------- ----------- ------------
```

CUDA: 2D linear interpolation – compute-bound

For each output point interpolation we need to read one value from the `*delays` array.
Can we calculate delays on-the fly instead? Will the processing time be improved?

- the kernel becomes specific for some imaging approach
- however the kernel becomes *less memory-bound* and *more compute-bound*

CUDA: 1D linear interpolation – compute-bound

From:

```
float sample = points[out_i]
```

To:

```
int point_x = point/nz;
int point_z = point-point_x*nz;

float angle = alpha[tx];
float z = z_origin + point_z*dz;
float x = x_origin + point_x*dx;
float element_pos = probe_origin + element*pitch;

float tx_delay = (z*cos(angle) + x*sin(angle))/c;
float rx_delay = hypotf(z, (x-element_pos))/c;
float sample = (tx_delay+rx_delay)*fs;
```

CUDA: 2D linear interpolation – compute-bound

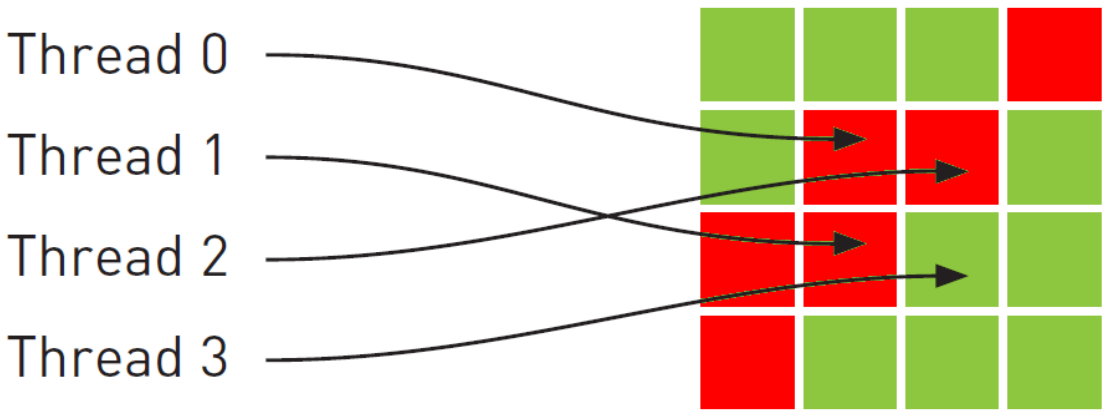
Step	Time (median, ms)
1D interpolation - first attempt	147
1D interpolation - write coalescing	51
1D interpolation - compute-bound	28

Texture memory

Texture memory is a GPU-specific global memory cache.

- CUDA-available cache.
- Dedicated for global memory access patterns with *spatial proximity*.
- Provides some basic interpolation and extrapolation functionality.

Texture memory



Texture memory

CUDA texture memory functionality:

- interpolation:
  - nearest-neighbor
  - linear, bilinear and trilinear
- extrapolation: clamp (extrapolate with the values on the border), constant value, ...
- address normalization (to $[0, 1]$)
- ...

CUDA: linear interpolation – Texture memory

```
from cp.cuda.texture import *

ch = ChannelFormatDescriptor(32, 0, 0, 0, cudaChannelFormatKindFloat)
array = CUDAarray(
    ch,
    width=n_samples,
    height=n_elements,
    depth=n_tx,
    flags=1, # CudaArrayLayered

)
res = ResourceDescriptor(cudaResourceTypeArray, cuArr=array)
tex = TextureDescriptor(
    (cudaAddressModeBorder, cudaAddressModeBorder),
    cudaFilterModeLinear,
    cudaReadModeElementType,
    borderColors=(0.0, 0.0, 0.0, 0.0)
)
texobj = TextureObject(res, tex)
# ... and intead of input array:
kernel(grid, block, args=(out, texobj, ...))
```

CUDA: linear interpolation – Texture memory

From:

```
out[out_i] = interp_linear(&in[in_i], points[out_i], ns);
```

To:

```
out[out_i] = tex2DLayered<float>(input, sample+0.5, element+0.5, t);
```

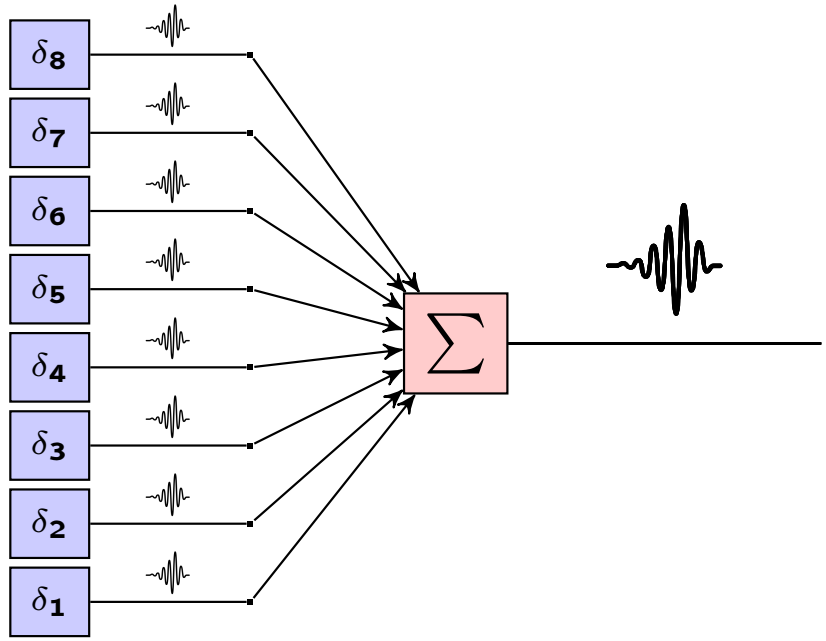
CUDA: 2D linear interpolation with Texture Memory

Step	**Time (median, ms)**
1D interpolation - first attempt	147
1D interpolation - write coalescing	51
1D interpolation - compute-bound	28
1D interpolation - texture memory	23

CUDA: 2D linear interpolation with Texture Memory

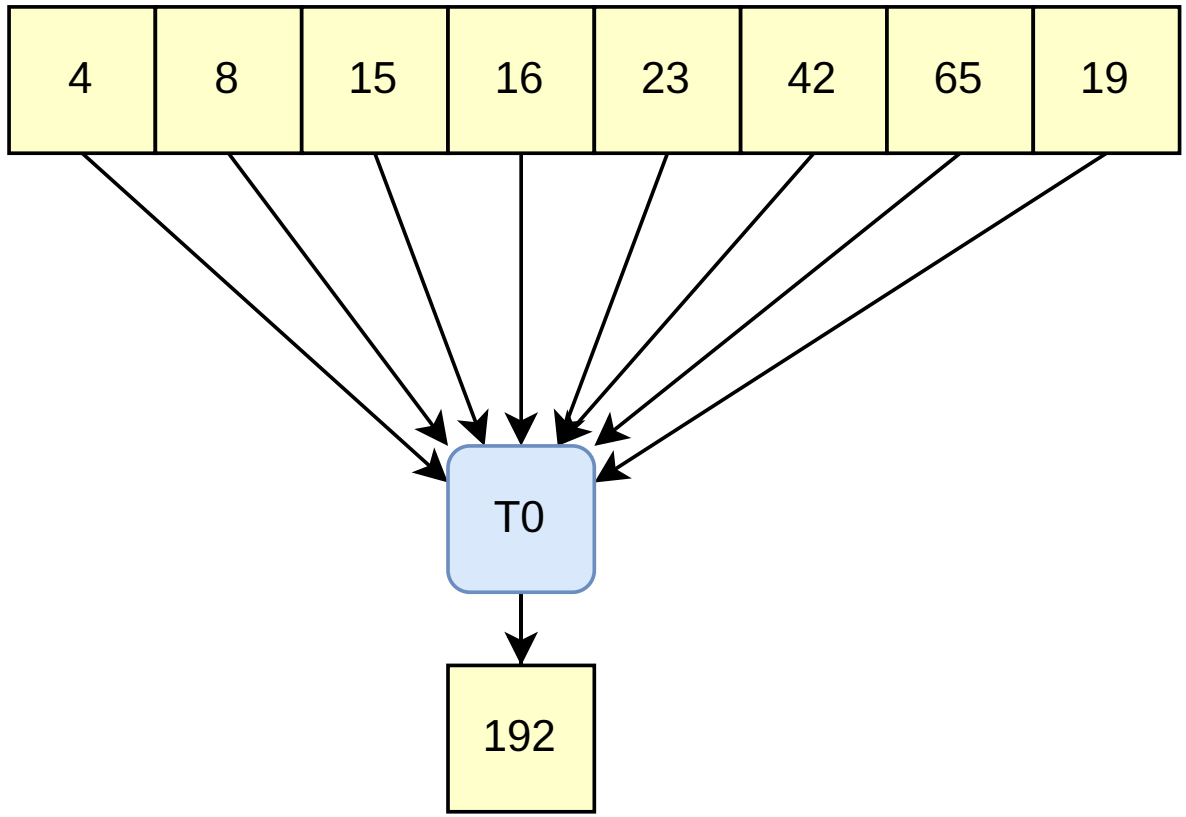
```
__global__
void interpolate1d_layered(
    float *output, cudaTextureObject_t input, const float *alpha,
    const size_t n_tx, const size_t n_elements, const size_t n_samples,
    const size_t n_points,
    const float c, const float fs,
    const float pitch, const float probe_origin,
    const float dx,    const float nx, const float x_origin,
    const float dz,    const float nz, const float z_origin) {

    int point = blockIdx.x*blockDim.x + threadIdx.x;
    int element = blockIdx.y*blockDim.y + threadIdx.y;
    int tx = blockIdx.z*blockDim.z + threadIdx.z;

    if(point >= n_points || element >= n_elements || tx >= n_tx) {
        return;
    }
    int point_x = point/nz;
    int point_z = point-point_x*nz;

    float angle = alpha[tx];
    float z = z_origin + point_z*dz;
    float x = x_origin + point_x*dx;
    float element_pos = probe_origin + element*pitch;

    float tx_delay = (z*cos(angle) + x*sin(angle))/c;
    float rx_delay = hypotf(z, (x-element_pos))/c;
    float sample = (tx_delay+rx_delay)*fs;


    size_t output_idx = (n_elements*n_points)*tx + (n_points)*element + point;
    output[output_idx] = tex2DLayered<float>(input, sample+0.5, element+0.5, tx);
}
```

Sum



$$h(t,p) \leftarrow \sum_{r=1}^{R} g(t,r,p)$$

Sequential sum – 1D



Sequential sum - 1D

```
__global__
void sum_sequential(float* output, float *input, size_t ne) {
    float result = 0.0f;
    for (int e = 0; e < ne; ++e) {
        result += input[e];
    }
    output[0] = result;
}
```

Sequential sum - 1D

Block and grid: $(1,)$
Vector of 1024 elements.

Step	**Time (median, μs)**
Sequential sum	340

Sequential sum - 1D

- $O(n)$
- optimal for a single-threaded implementation
- but only a single thread is used, and GPU have them a lot more

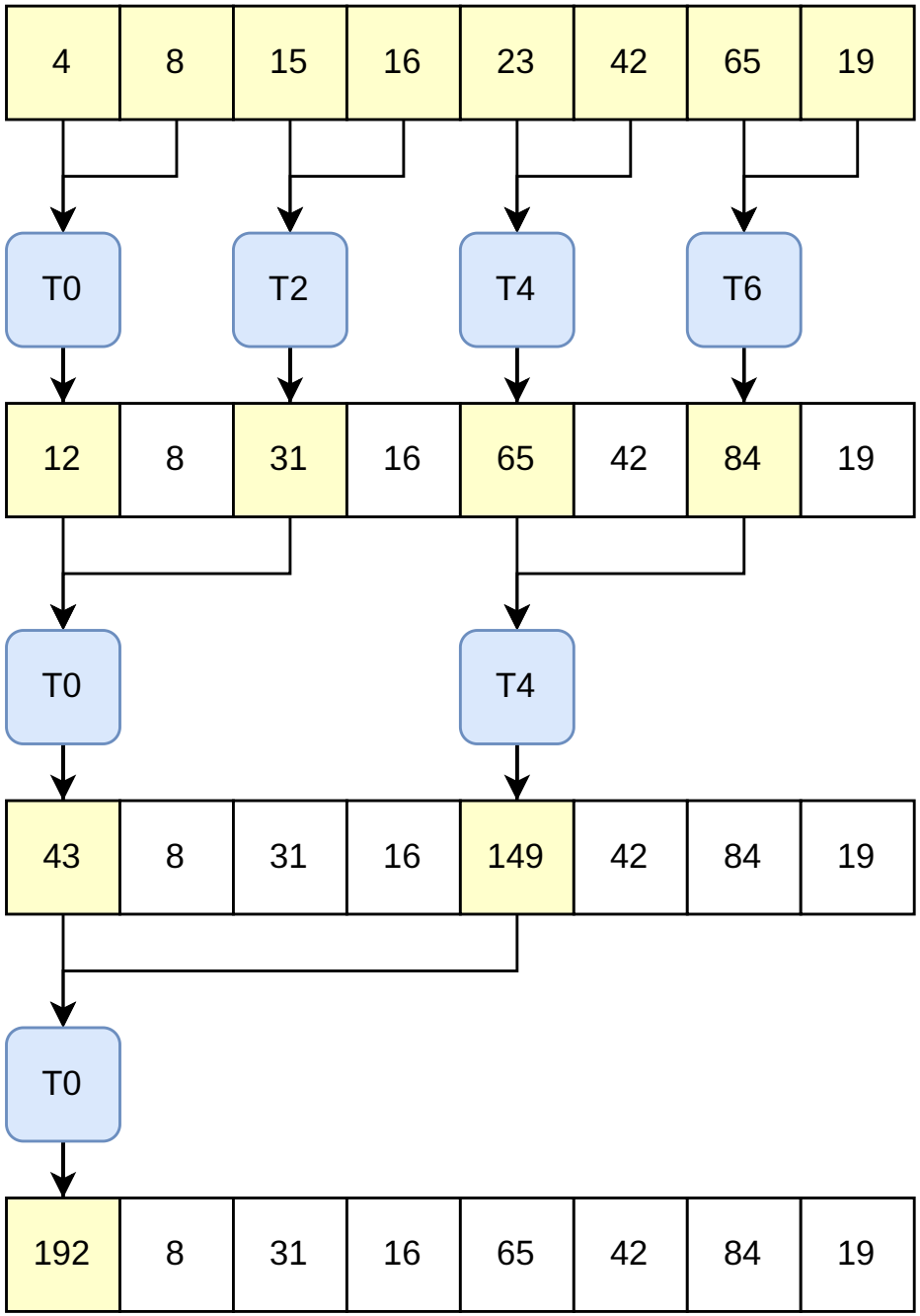
Parallel sum - 1D

We can further reduce the computational complexity to $O(lgn)$ for the multi-core CPU/GPUs:

- threads with id divisible by 2: $h[id] += h[id+1]$ then,
- threads with id divisible by 4: $h[id] += h[id+2]$ then,
- ...
- threads with id divisible by $2^{lg(n)}$: $h[id] += h[id+2^{n-1}]$.

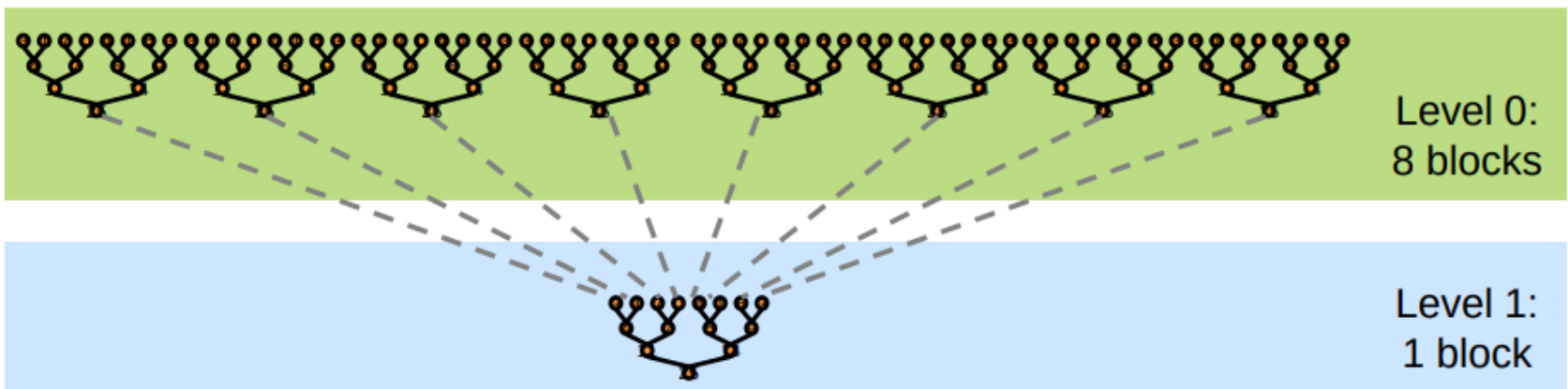
This is Brent's algorithm, usually visualized as a reduction tree.

Parallel sum - 1D



Parallel sum - 1D

- The reduction can be performed by a single block of threads, using shared memory.
- However the max number of threads ≤ 1024.
- How can we reduce larger arrays? Apply the reduction tree recursively.



source: Optimizing Parallel Reduction in CUDA, Mark Harris

Parallel sum - 1D

```
__global__
void sum_parallel(float *out, float *in) {
    extern __shared__ float shared_memory[];
    int thread = threadIdx.x;
    int element = thread;

    shared_memory[thread] = in[element];
    __syncthreads();
    for(int s = 1; s < blockDim.x; s *= 2) {
        if (thread % (2*s) == 0) {
            shared_memory[thread] += shared_memory[thread + s];
        }
        __syncthreads();
    }
    if (thread == 0) {
        out[nz*x+z] = shared_memory[0];
    }
}
```

Parallel sum - 1D

Block and grid: $(1024,)$
Vector of 1024 elements.

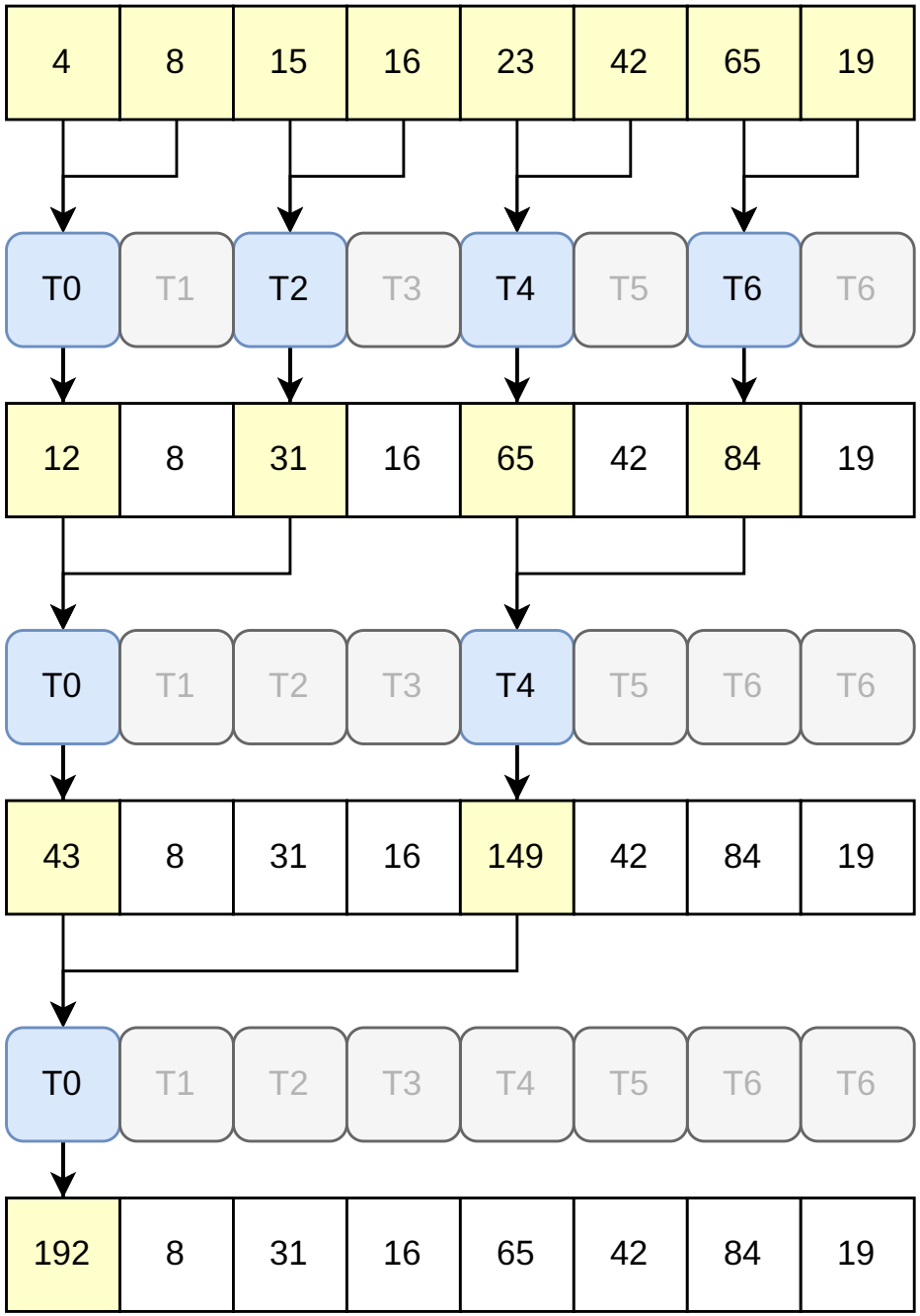
Step	**Time (median, μs)**
Sequential sum	340
Parallel sum	37

Parallel sum - 1D – branch divergence

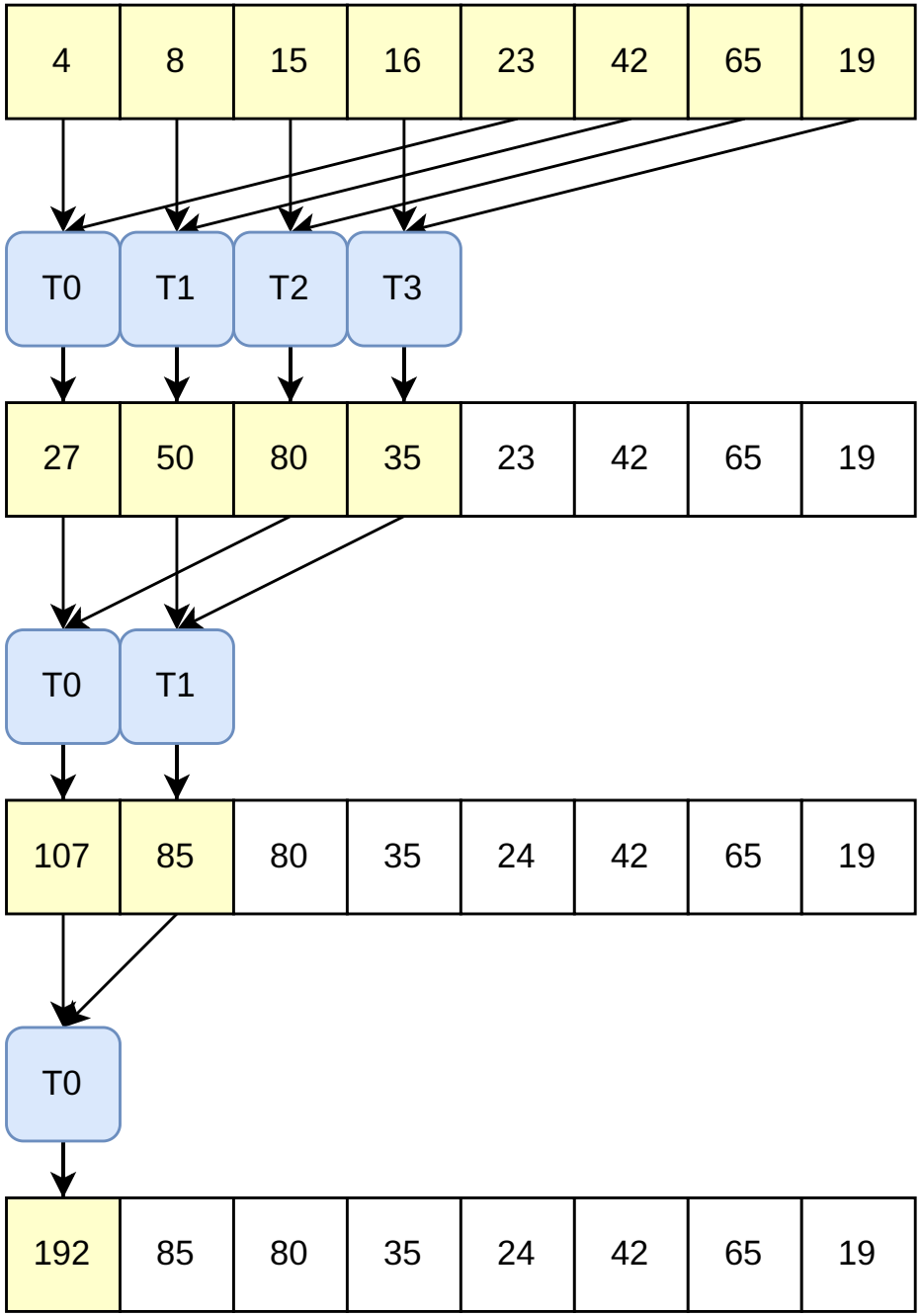
Every second thread in the warp (32 threads) is inactive.

```
    for(int s = 1; s < blockDim.x; s *= 2) {
        if (thread % (2*s) == 0) {
            shared_memory[thread] += shared_memory[thread+s];
        }
        __syncthreads();
    }
}
```

Parallel sum - 1D – branch divergence



Parallel sum - 1D – non-divergent



Parallel sum - 1D – non-divergent

From:

```
for(int s = 1; s < blockDim.x; s *= 2) {
    if (thread % (2*s) == 0) {
        shared_memory[thread] += shared_memory[thread+s];
    }
    __syncthreads();
}
```

To:

```
for (int s = blockDim.x/2; s > 0; s >>= 1) {
    if (thread < s) {
        shared_memory[thread] += shared_memory[thread+s];
    }
    __syncthreads();
}
```

Parallel sum – 1D – non-divergent

Step	**Time (median, μs)**
Sequential sum	340
Parallel sum	37
Parallel sum – non-divergent	23

Parallel sum – 1D – non-divergent

Only half of the threads is active on the first loop (and the other half is just inactive).

- Halve the number of threads/blocks.
- Replace single element load with two loads and a single addition.

Parallel sum – 1D – non-divergent

From:

```
shared_memory[thread] = in[element];
```

To:

```
shared_memory[thread] = in[element]+in[element+blockDim.x];
```

Parallel sum – 1D – first add

Block and grid: (1024,)
Vector of 1024 elements.

Step	**Time (median, μs)**
Sequential sum	340
Parallel sum	37
Parallel sum – non-divergent	22
Parallel sum – first add	16

Parallel sum – 1D – unroll last warp

- when $s \leq 32$ only one warp in the block is active
- SIMD
  - we don't need `syncthreads`
  - we can skip check if thread id is $<$ s (all threads within warp are used anyway)

Parallel sum – 1D – unroll last warp

From:

```
for (int s = blockDim.x/2; s > 0; s >>= 1) {
    if(thread < s) {
        shared_memory[thread] += shared_memory[thread+s];
    }
    __syncthreads();
}
```

To:

```
for (int s = blockDim.x/2; s > 32; s >>= 1) {
    if(thread < s) {
        shared_memory[thread] += shared_memory[thread+s];
    }
    __syncthreads();
}
if(thread < 32) {
    volatile float* sm = shared_memory;
    sm[thread] += sm[thread + 32];
    sm[thread] += sm[thread + 16];
    sm[thread] += sm[thread + 8];
    sm[thread] += sm[thread + 4];
    sm[thread] += sm[thread + 2];
    sm[thread] += sm[thread + 1];
}
```

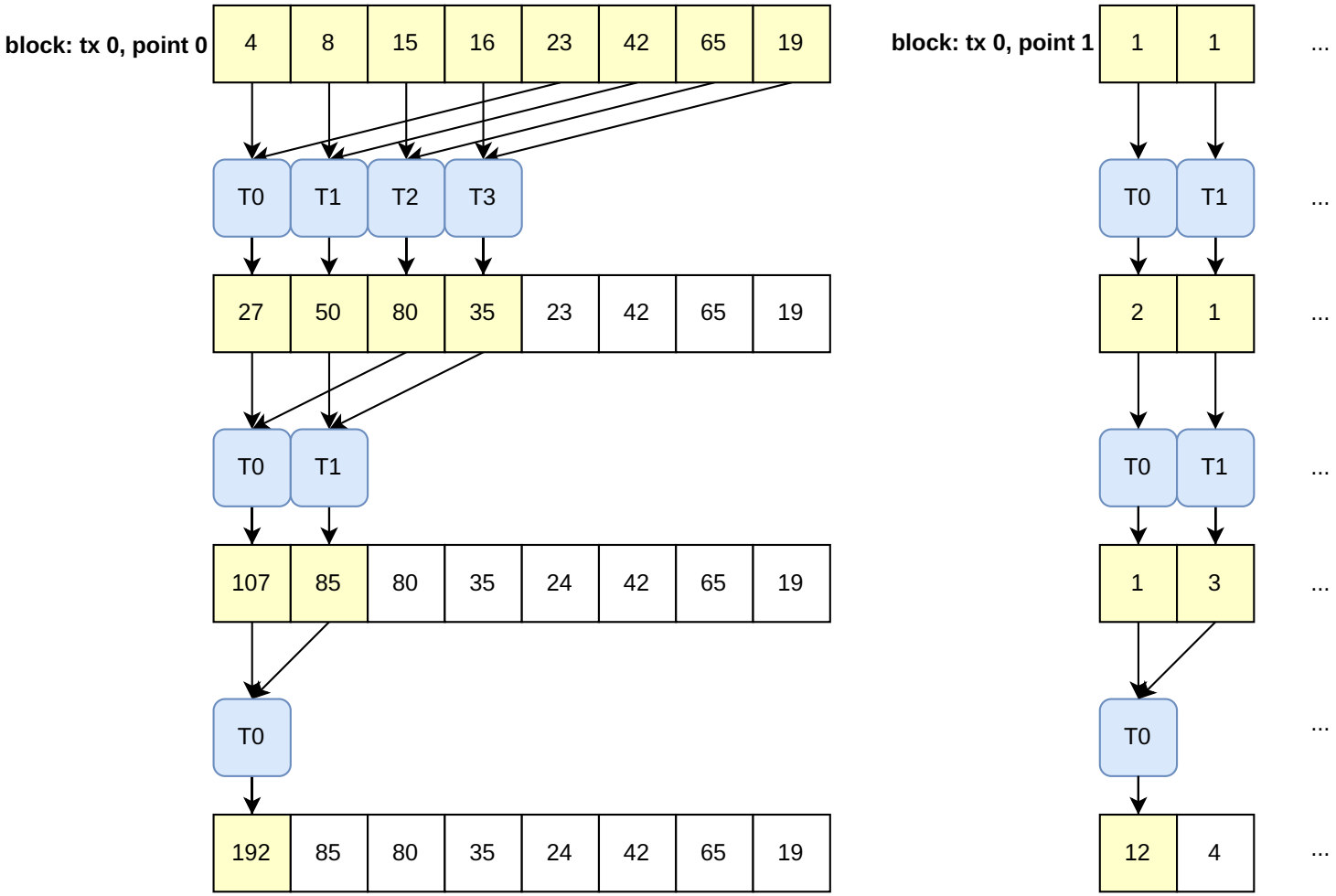
Parallel sum – 1D – unroll last warp

Block and grid: (1024,)
Vector of 1024 elements.

Step	**Time (median, μs)**
Sequential sum	340
Parallel sum	37
Parallel sum – non-divergent	22
Parallel sum – first add	16
Parallel sum – unroll last warp	12

Parallel sum – DAS

How can we assign reduction for each output image point?



Parallel sum – DAS

```
__global__
void sum_parallel4(float *out, float *in,
                   size_t nx, size_t nz, size_t n_elements)
{
    extern __shared__ float shared_memory[];
    int thread = threadIdx.x;
    int z = blockIdx.y;
    int x = blockIdx.z;
    int el = thread;
    if (z >= nz || x >= nx) {return;}

    int off = nz*n_elements*x + n_elements*z;
    shared_memory[thread] = in[off+el] + in[off+el+blockDim.x];
    __syncthreads();

    for (int s = blockDim.x/2; s > 32; s >>= 1) {
        if (thread < s) {
            shared_memory[thread] += shared_memory[thread+s];
        }
        __syncthreads();
    }
    if(thread < 32) {
        volatile float* warp_shm = shared_memory;
        warp_shm[thread] += warp_shm[thread + 32];
        warp_shm[thread] += warp_shm[thread + 16];
        warp_shm[thread] += warp_shm[thread + 8];
        warp_shm[thread] += warp_shm[thread + 4];
        warp_shm[thread] += warp_shm[thread + 2];
        warp_shm[thread] += warp_shm[thread + 1];
    }

    if (thread == 0) {
        out[nz*x+z] = shared_memory[0];
    }
}
```

Parallel sum – DAS

Array $(256, 1024, 128)$.

Step	**Time (median, ms)**
Sequential sum	7.87
Parallel sum	11.38
Parallel sum – non-divergent	7.82
Parallel sum – conflict free	7.71
Parallel sum – first add	7.63
Parallel sum – unroll last warp	7.62

Sum – Conclusions

Where is the bottleneck?
We have $O(lgn)$, however the constant factor may be quite high:

- synchronization barrier,
- occupancy (with every loop iteration less and less threads are active),
- instruction overhead (see Mark Harris presentation on it).

And the axis we are reducing along has only 128 elements.
What options do we have?

- Use sequential sum: it's probably good enough for < 1024 elements.
- Combine sequential with parallel sum

In the further examples we will use sequential sum, for the sake of simplicity.

Combining delay and sum: stream of 2 kernels

Let's combine the kernels to get the reconstructed ultrasound images.

```
with stream:
    h = interpolate1d_kernel(block1, grid1, params=(f, # ...
    g = sequential_sum(block2, grid2, params=(h, #...
```

Combining delay and sum: stream of 2 kernels

Step	**Time (median, ms)**
Delay and sum: 2 kernels	27+25 = 52

Frame rate < 20 FPS.

Not impressive...

Combining delay and sum: stream of 2 kernels

Where is the bottleneck?

$$g(t, r, x, z) \leftarrow f(t, r, \delta(t, r, x, z))$$

$$h(t, x, z) \leftarrow \sum_{r=1}^{R} g(t, r, x, z)$$

- interp1d writes h to global memory,
- sum reads h from global memory,
- h has $N_t N_p N_r$ values,
- reading/writing from/to global memory is expensive.

Do we really need g?
Answer: usually no, and we can combine delay and sum, e.g. implement kernel $h(f)$.

Combining delay and sum: single kernel

```
__global__
void beamform(float *out, const float *in, const float *angles,
              const float pitch, const float probe_origin, /*...*/) {
    int z = blockIdx.x*blockDim.x + threadIdx.x;
    int x = blockIdx.y*blockDim.y + threadIdx.y;
    int tx = blockIdx.z*blockDim.z + threadIdx.z;

    float rx_distance, sample_number, element_pos;
    int offset;

    if (z >= nz || x >= nx || tx >= n_angles) {
        return;
    }

    float pixel_x = x_origin + dx*x; // [m]
    float pixel_z = z_origin + dz*z; // [m]
    float angle = angles[tx];
    int tx_offset = (n_elements*n_samples)*tx;

    float tx_distance = pixel_z*cosf(angle) + pixel_x*sinf(angle);

    out[(nz*nx)*tx + (nz)*x + z] = 0.0f;
    for (int element = 0; element < n_elements; ++element) {
        offset = tx_offset + (n_samples)*element;

        element_pos = probe_orig + element*pitch;
        rx_distance = hypotf(pixel_x - element_pos, pixel_z);
        delay = (tx_distance + rx_distance)/speed_of_sound*sampling_frequency;

        float value = interp_linear(&in[offset], delay, n_samples);
        out[(nz*nx)*tx + (nz)*x + z] += value;
    }
}
```

Combining delay and sum: single kernel

Step	Time (median, ms)
Delay and sum: 2 kernels	27+25 = 52
Delay and sum: 1 kernel	8

Conclusions and further improvements

Conclusion: optimizing the memory access pattern is usually critical for performance.
Further improvements:

- use IQ data (and increase the decimation factor)
- combine sequential with parallel reduction
- use intrinsic functions (e.g. `sincos` to get sin and cos)
- move transmit parameters (e.g. transmit angle) to constant memory (in case $\gg 3$)

Delay and sum for matrix array

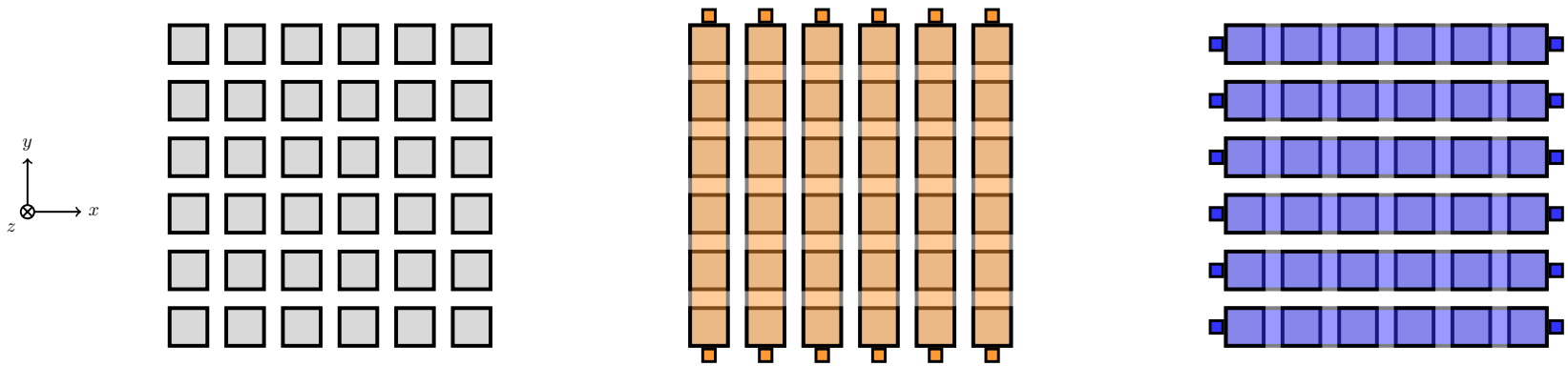
$$g(t,x,y,z) = \sum_{r_y=1}^{N_{r_y}} \sum_{r_x=1}^{N_{r_x}} f(t, r_x, r_y, \delta(t, r_x, r_y, x, y, z)) \qquad (2)$$

Where *delta* depends on a pair of angles α, γ or OXZ, OYZ in case of PW imaging.
Some practical notes:

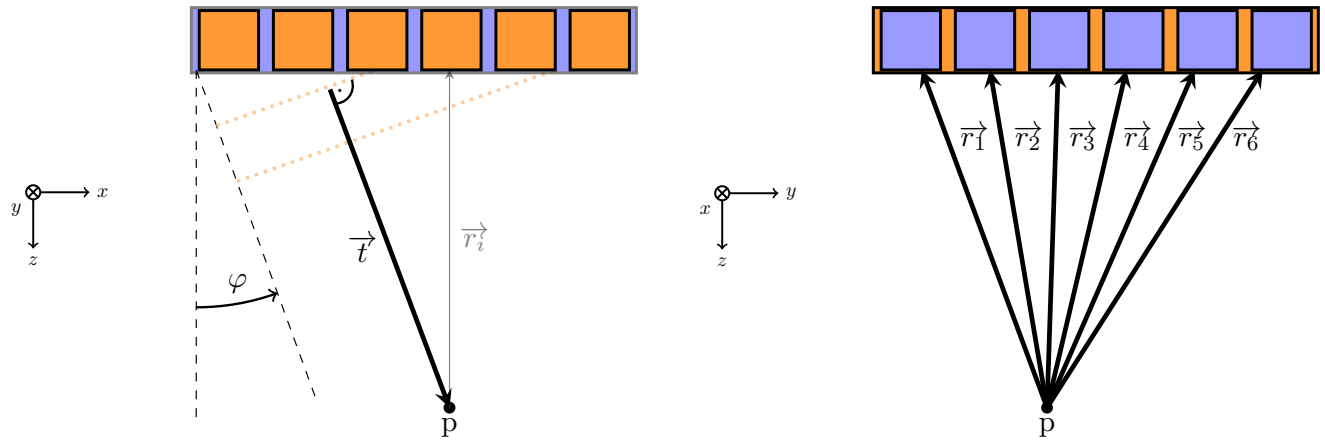
- Delays becomes $\delta(t, r_x, r_y, x, y, z)$ and can be large for extensive volumes – highly recommended to **compute on the fly** to save GPU RAM and reduce memory W/R.
- The LRIs become $g(t, x, y, z)$ and can be relatively large for extensive volumes – consider summing along TX and RX in a single kernel.
- δ becomes more complex due to 2D angle space – highly recommended to optimize with CUDA intrinsics.

Delay and sum for RCA

RCA – Row-column Arrays



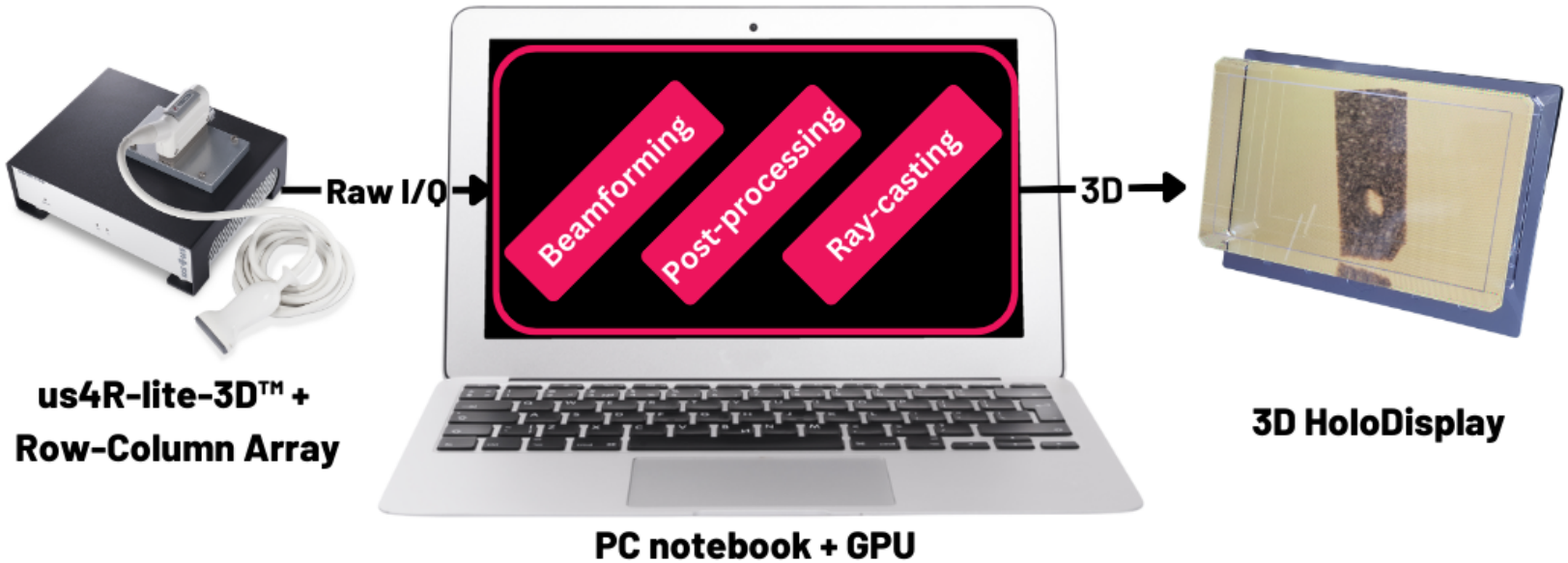
Delay and sum for RCA



$$\| \vec{t} \|_2 = z\cos(\varphi) + x\sin(\varphi), \| \vec{r_i} \|_2 = \sqrt{z^2 + (y - r_i)^2}$$

$$\delta_t(t, x, z) = \| \vec{t} \|_2 \frac{fs}{c}, \delta_r(r, y, z) = \| \vec{r_i} \|_2 \frac{fs}{c}$$

Delay and sum for RCA



- D4L-05, Thursday, September 7, 2:45, *Real-Time 3D Imaging Pipeline for Row-Column Array Probe and Holographic Display*, Marcin Lewandowski et al.
- exhibition boot

References

- *Programming Massively Parallel Processors*, David Kirk, Wen-Mei W. Hwu
- *Optimizing Parallel Reduction in CUDA*, Mark Harris
- *CUDA Toolkit Documentation 12.2*, NVIDIA

CUDA by example: delay and sum

Thank you!
→ Billy, Real-time Speed of Sound Estimation using CNN Inferencing Using GPU