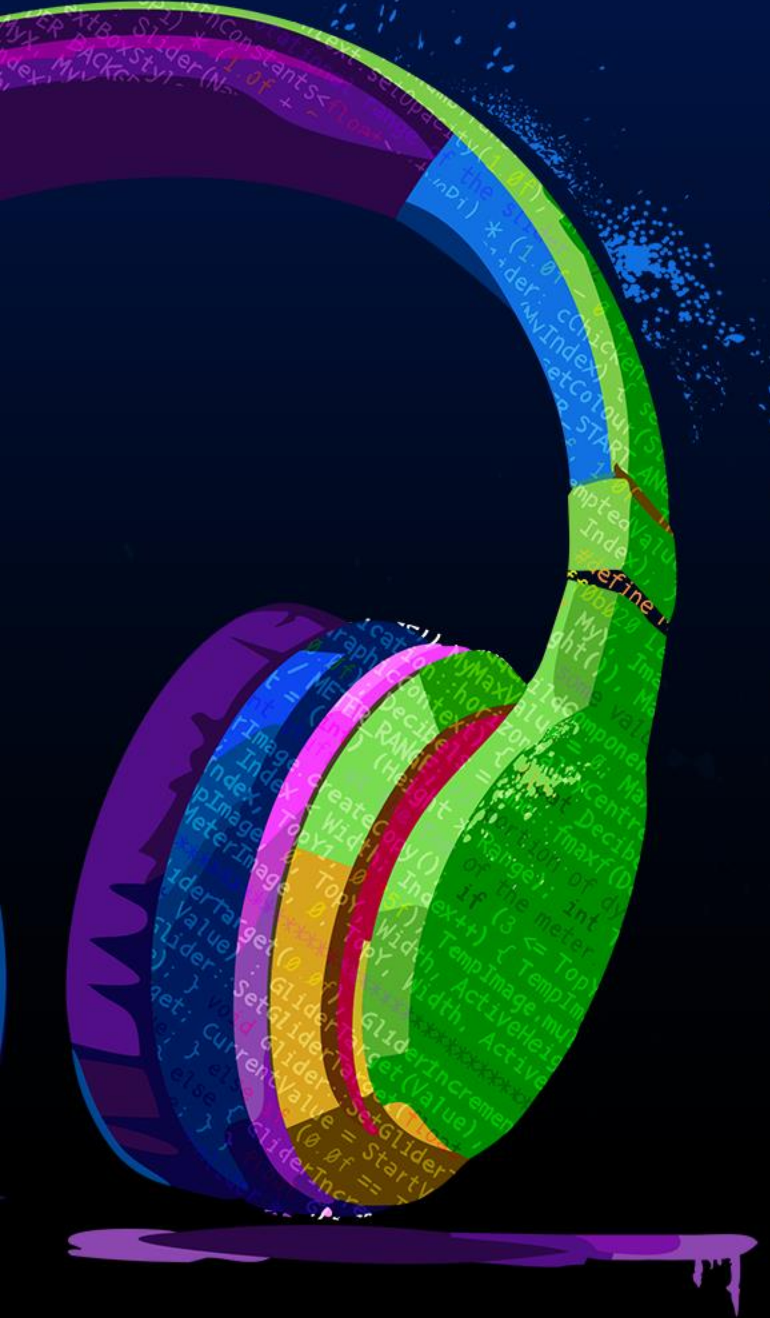
ADC24 Bristol



SYMPHONY OF A THOUSAND

GPU SYNTHESIS WITH MASSIVELY PARALLEL OSCILLATORS

CECILL ETHEREDGE

Who am I?

KoalaDSP – plugins, music tech

Snowcrash – hardware engineering

ijsf · CИTROPY

‘00s	Video game development
	Video game music
	BSc – Computer Science
‘10s	GPGPU, graphics & raytracers
	Physics engines for medical (CUDA)
	Hardware engineering
	Music production & experiments
	MSc – Embedded Systems (EE)
‘20s	Music technology – plugins & HW
	ADC2024!

Today’s goals

How to get started building a GPU wavetable synth

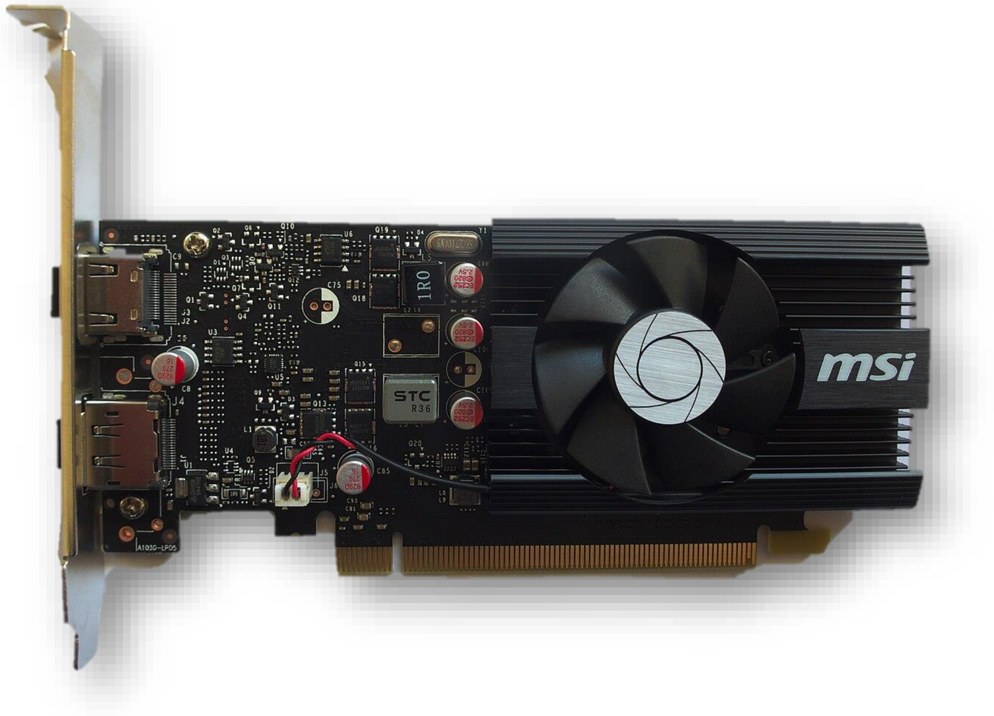
- Try to cover most aspects.
- Get acquainted.
- Get motivated. Its not rocket science!

What’s not the goal?

- Not a deep dive into an ideal, super-optimized implementation.
- No high-end GPUs.

No high-end GPUs? 🧐

- **Mission: find the cheapest retail NVIDIA GPU I can buy**

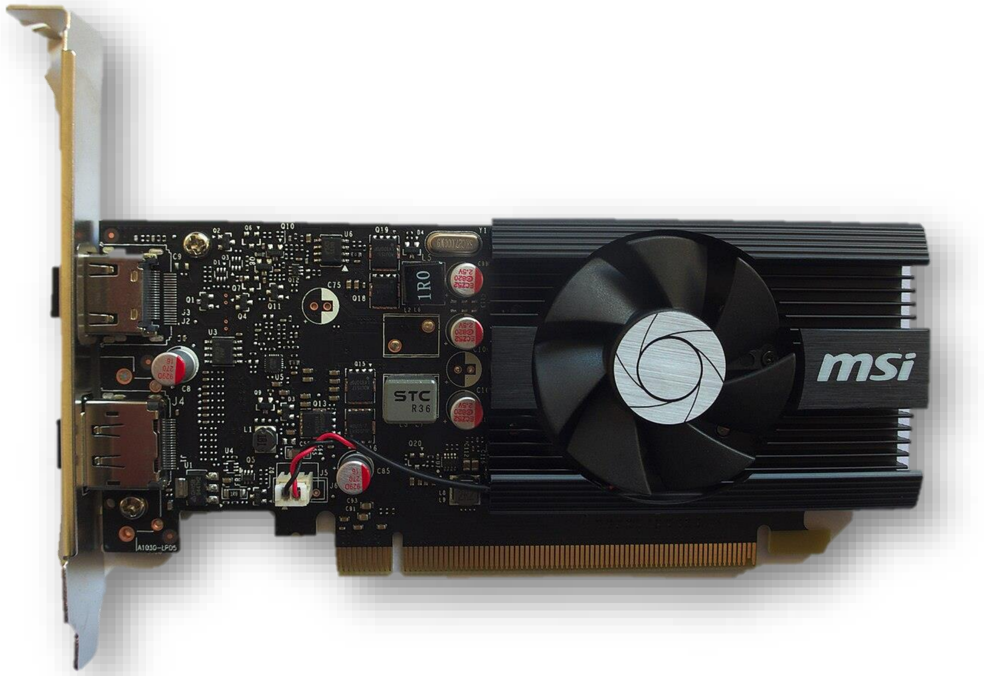


NVIDIA GT1030 2GB DDR4 (‘18)

- 3 SMs (RTX 4090 has 128 SMs)
- DDR4 (RTX 4090 has GDDR6X)

No high-end GPUs? 🧐

- Mission: find the cheapest retail NVIDIA GPU I can buy



NVIDIA GT1030 2GB DDR4 ('18)

- 3 SMs (RTX 4090 has 128 SMs)
- DDR4 (RTX 4090 has GDDR6X)

"Worst GPU of the last decade"
"The DDR4 abomination"

SMs

Nyquist

pre-emption

aliasing

pageable memory

windowed sinc

shared memory

periodic waveform

kernel

persistence

mipmap

block

parallel sum

TCC

pinned

interpolation

band limiting

thread

zero copy

convolver

task parallel

cycle

texture memory

WDDM

periodic waveform

warp

drift

data dependency

Brief history

of digital wave(table) synthesizers (and parallel programming?)

1978 - Palm Wavecomputer 360

1980 - Palm PPG Wave

1985 - Palm PPG Wave 2.3

64 waves
harmonic aliasing
loadable wavetables
8x2 hardware oscillators
discrete logic, adders, flipflops

Brief history

of digital wave(table) synthesizers (and parallel programming?)

1978 - Palm Wavecomputer 360

1980 - Palm PPG Wave

1985 - Palm PPG Wave 2.3 .... 1985 - Wersi MK1, EX-20

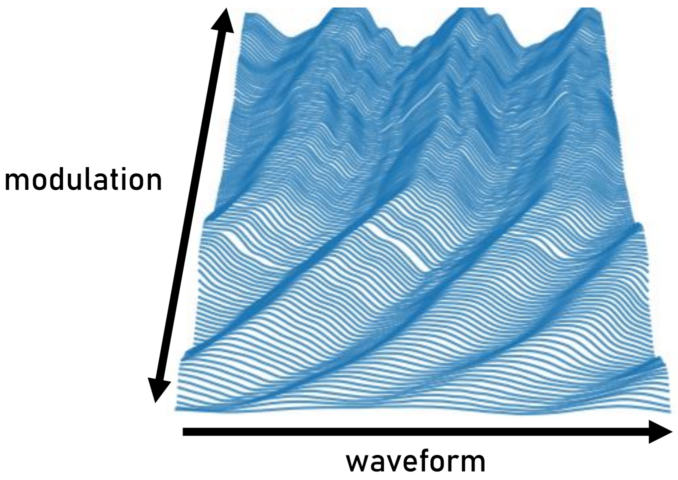
Palm PPG Wave 2.3	Wersi MK1, EX-20
64 waves	4 waves
harmonic aliasing	fft band limited
loadable wavetables	live tunable waveforms
8x2 hardware oscillators	20x software oscillators
discrete logic, adders, flipflops	small supercomputer: 22 CPUs 🚀

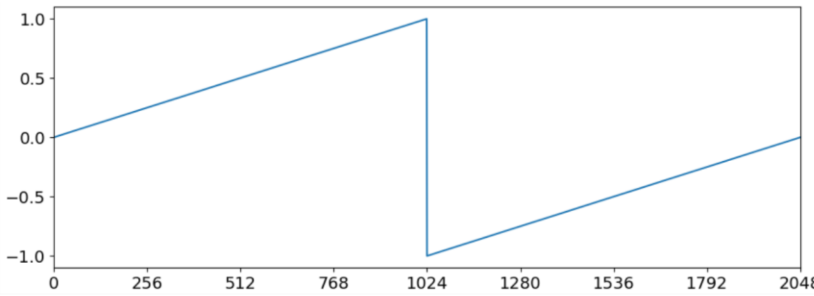
Wavetable synthesis

+

GPU programming

Wavetable synthesis





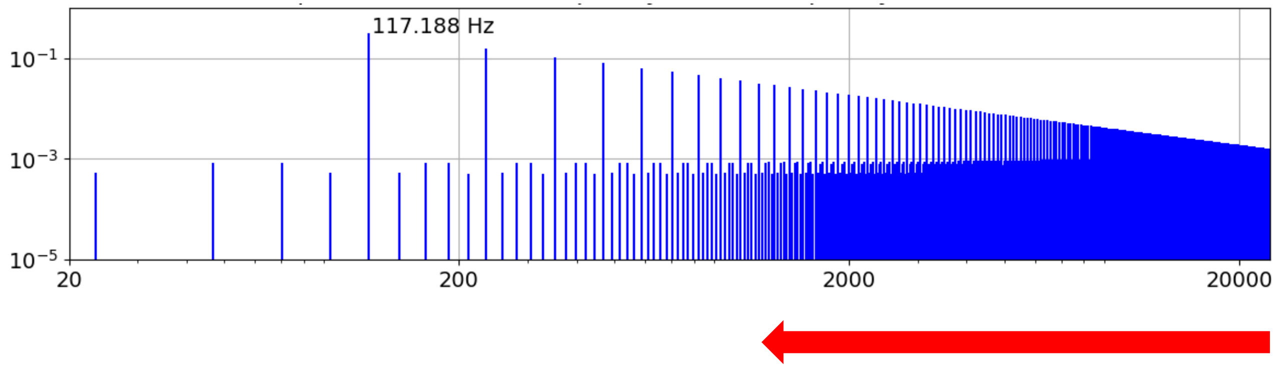
Fundamentally:

- Sampling periodic waveforms.
- Wavetable modulation during playback.

Aliasing

Nyquist-Shannon theorem

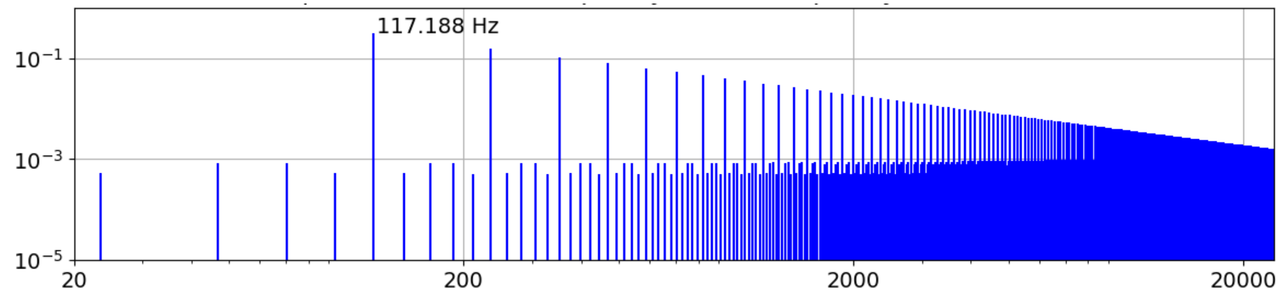
Signal frequency
(spectrum bandwidth)
<
0.5 × sampling rate

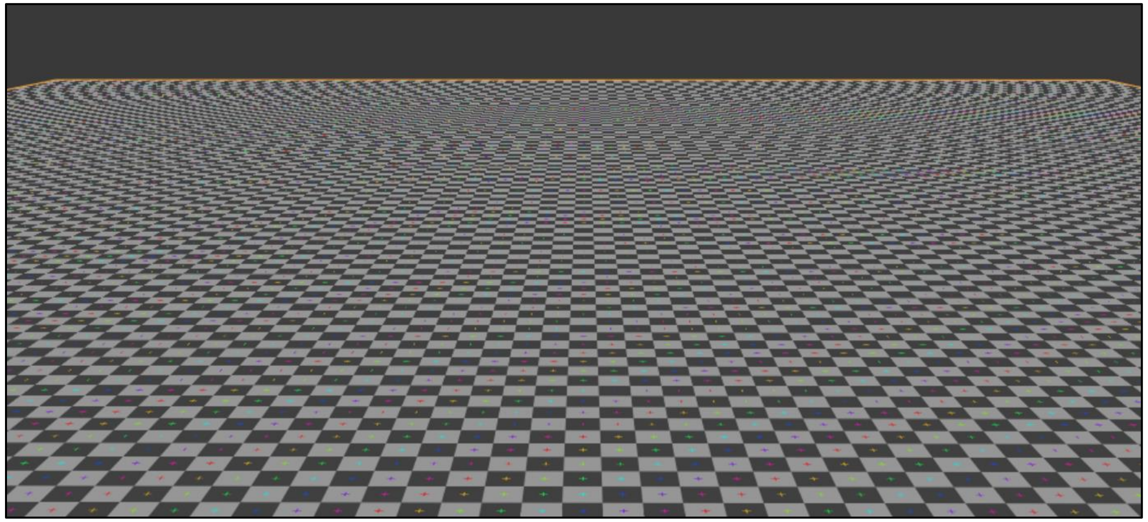


Aliasing

Nyquist-Shannon theorem

Signal frequency (spectrum bandwidth) < **0.5 × sampling rate**



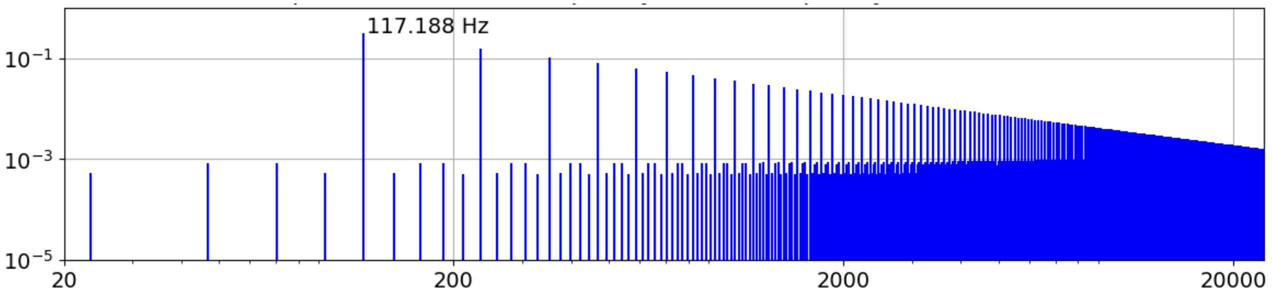


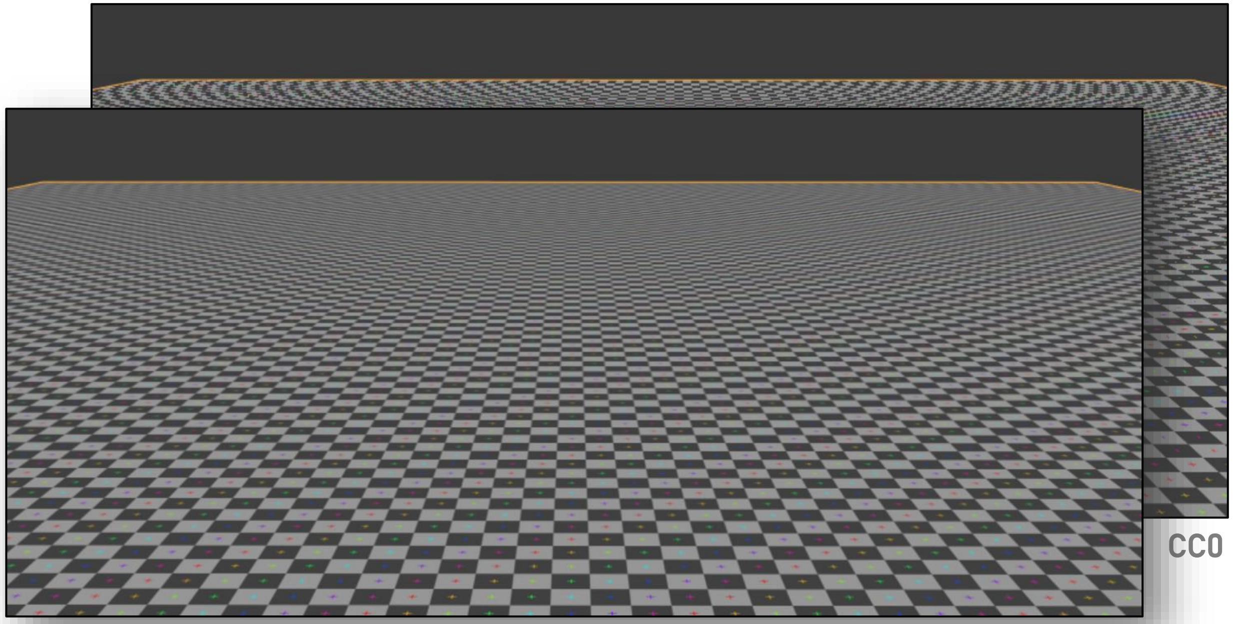
Aliasing

Nyquist-Shannon theorem

Signal frequency (spectrum bandwidth) **<** **0.5 × sampling rate**

💡 Solution: **mipmap**





CC0

CC0

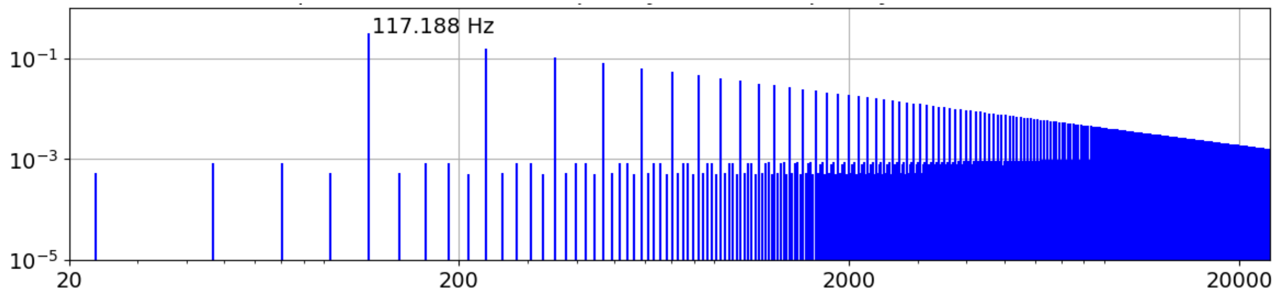
Aliasing

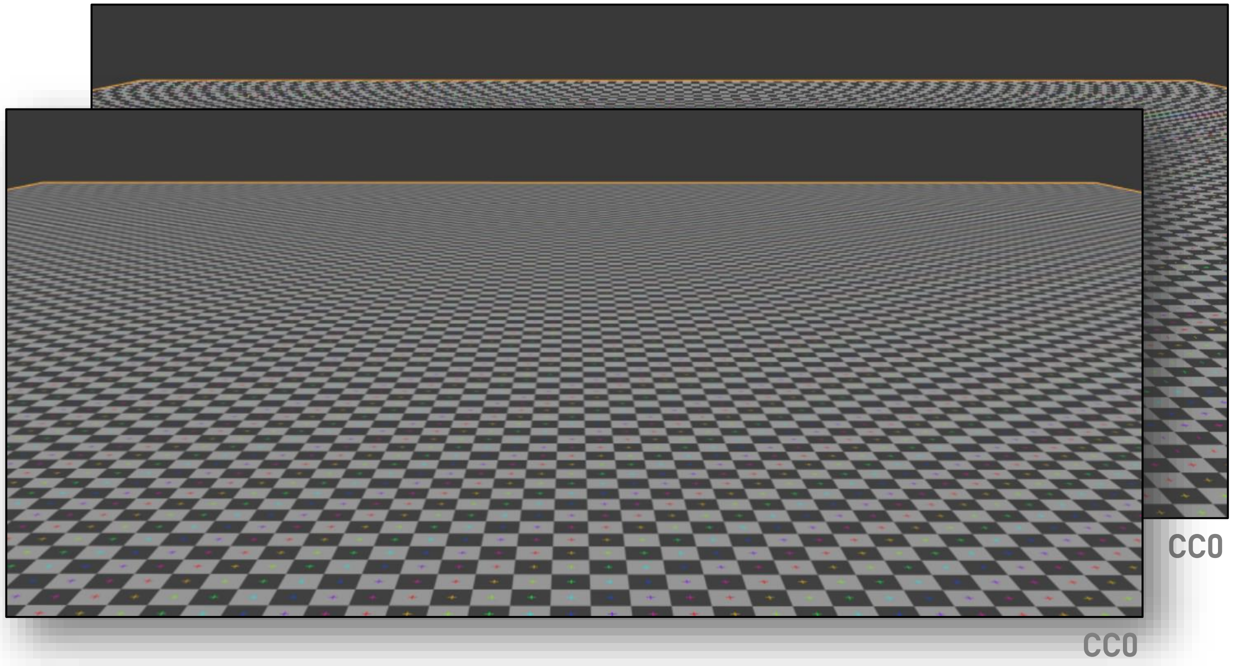
Nyquist-Shannon theorem

Signal frequency (spectrum bandwidth) **<** **0.5 × sampling rate**

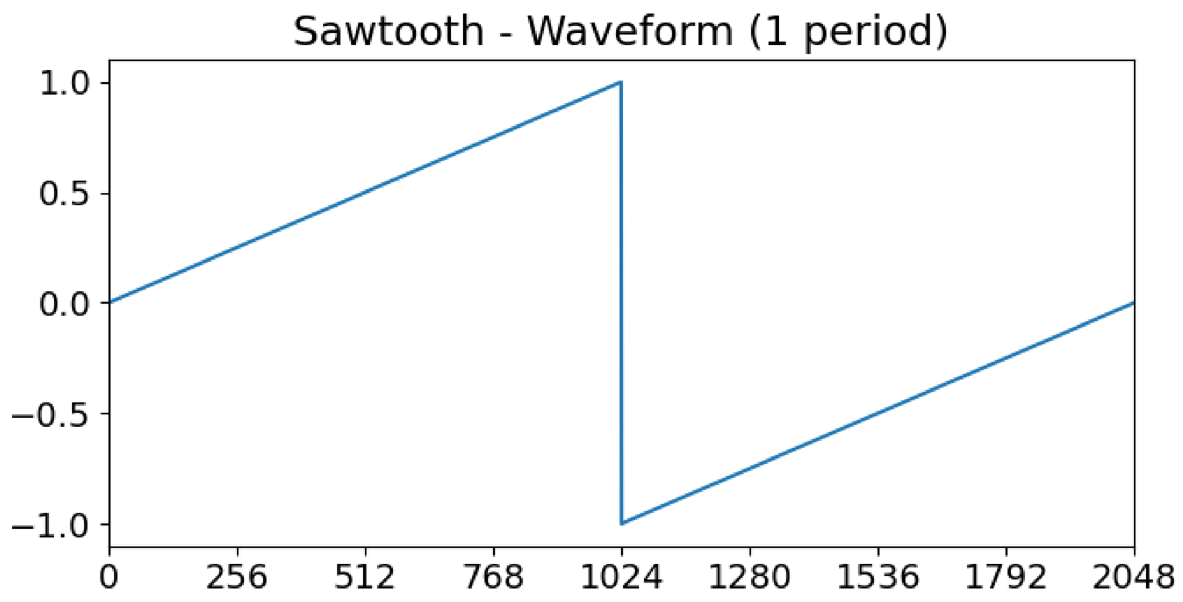
💡 Solution: ~~mipmap~~ **band limiting**

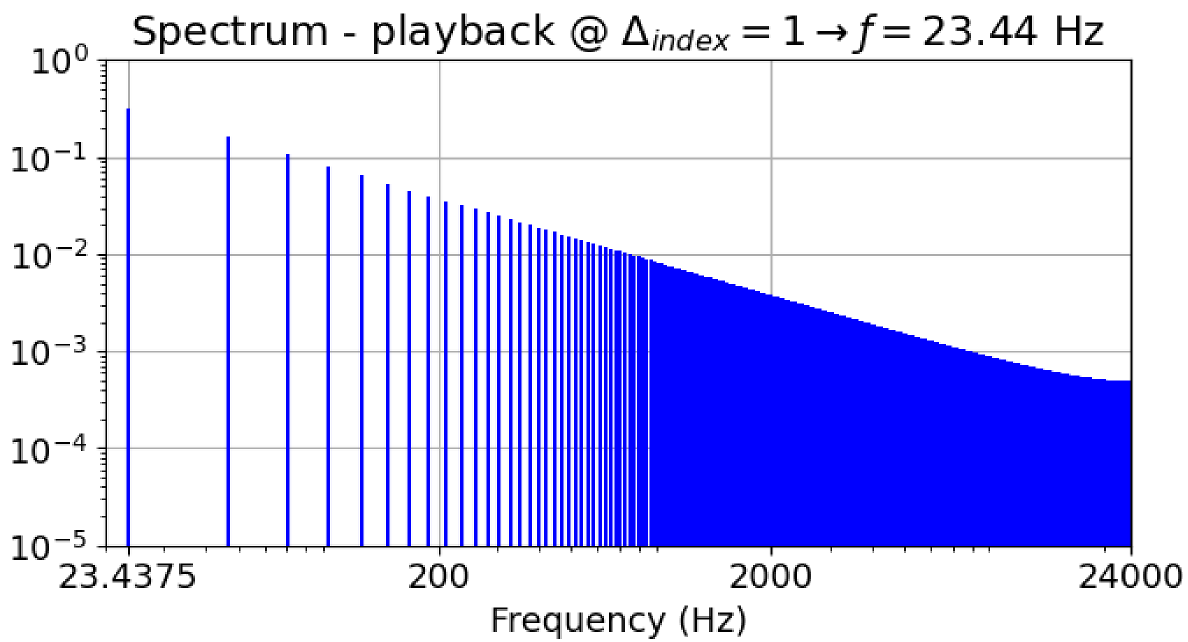
(filtering!)



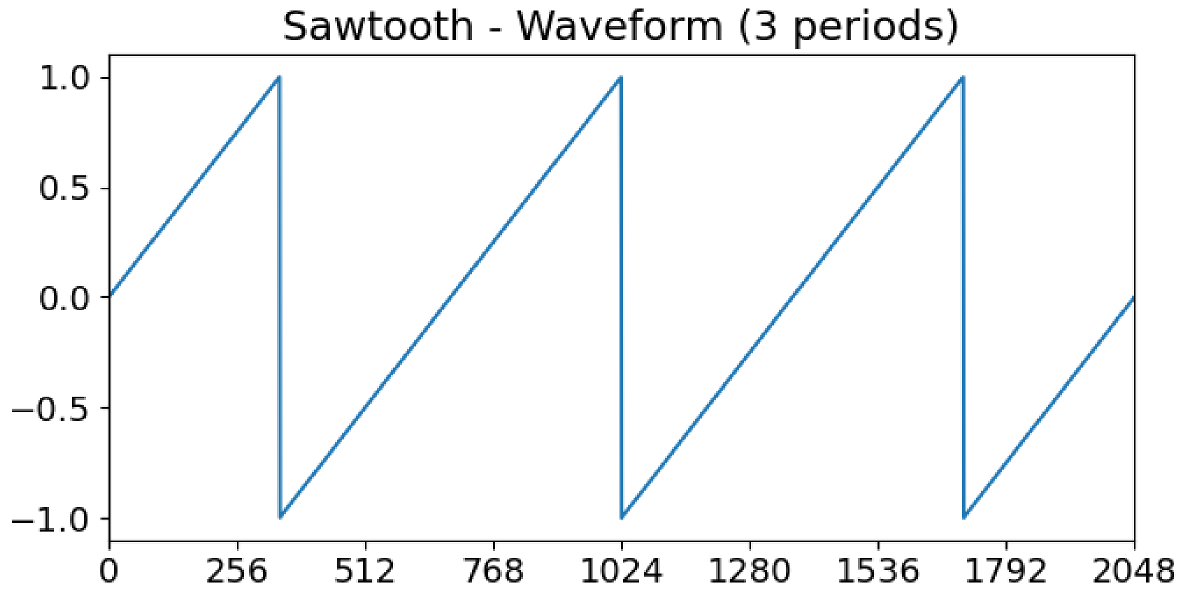


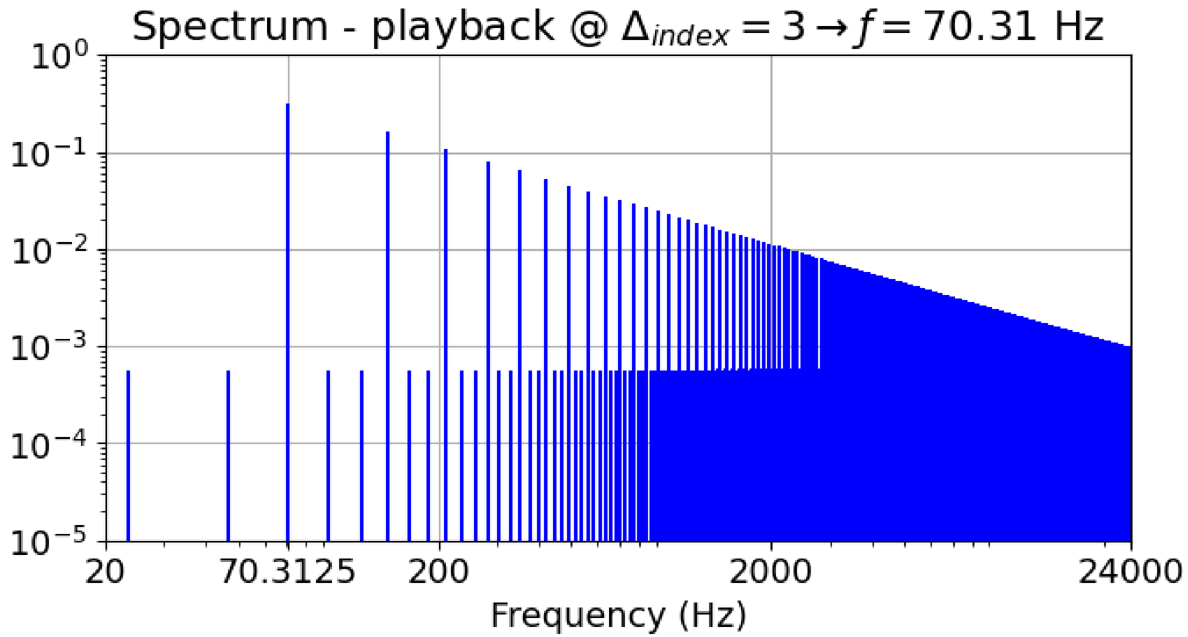
Band limiting



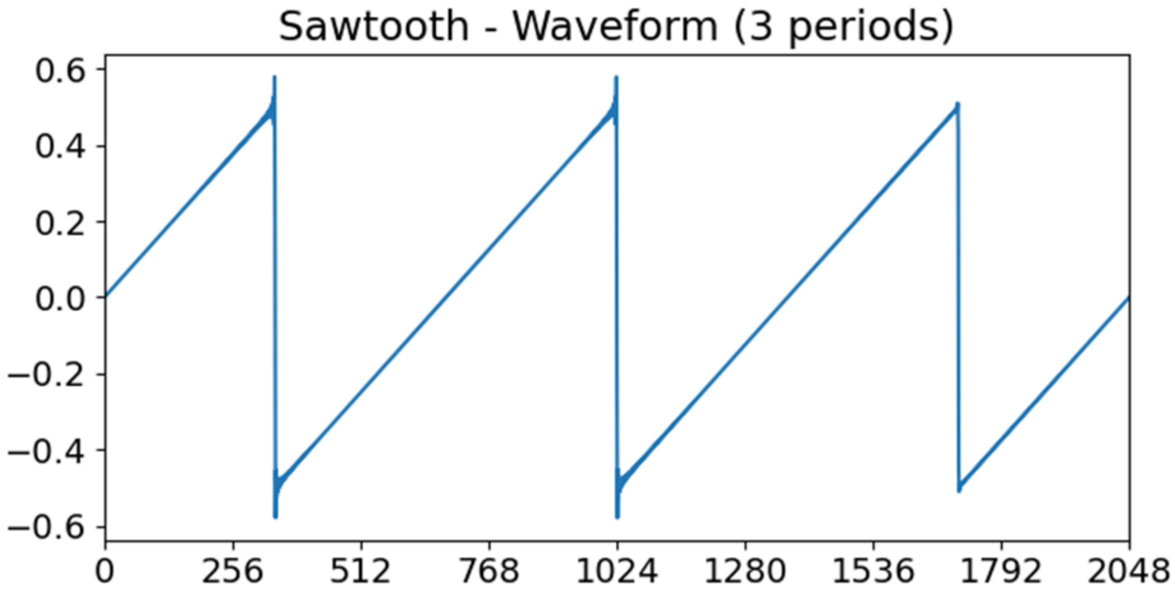


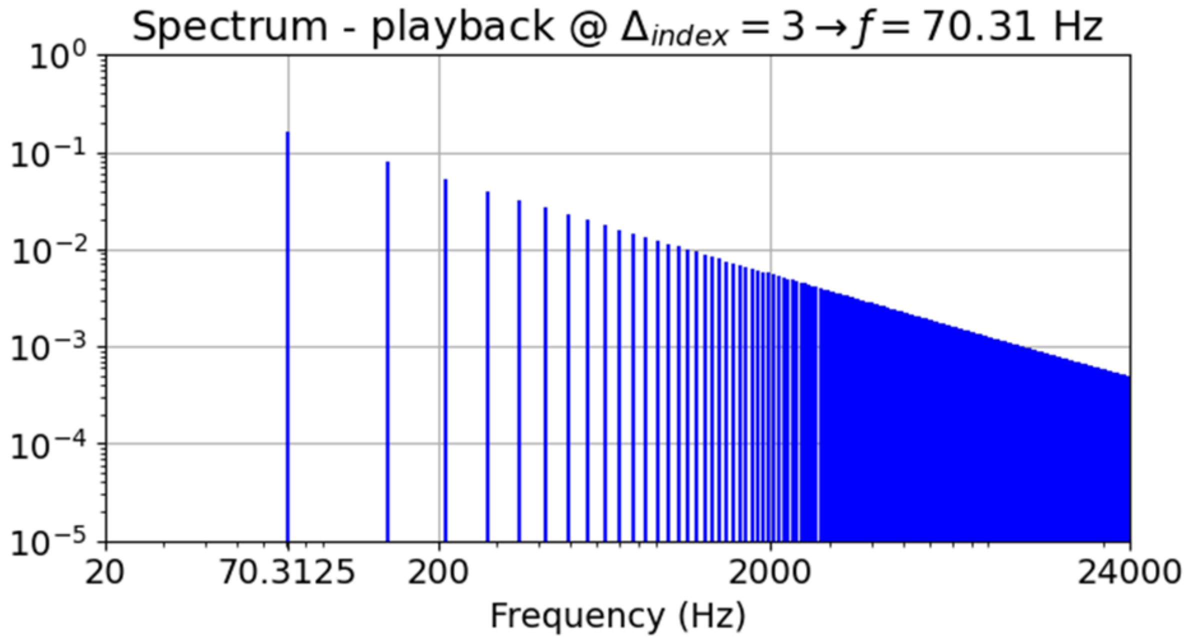
Band limiting

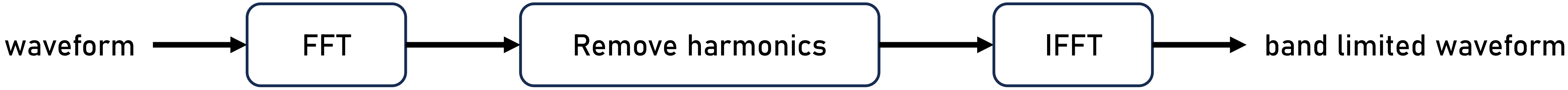




Band limiting

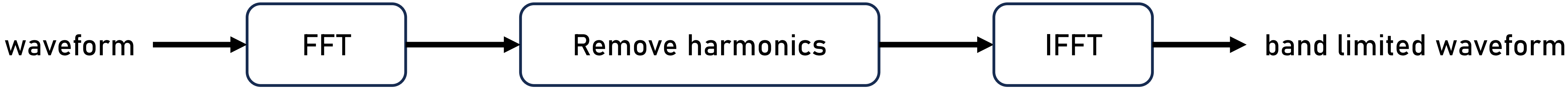






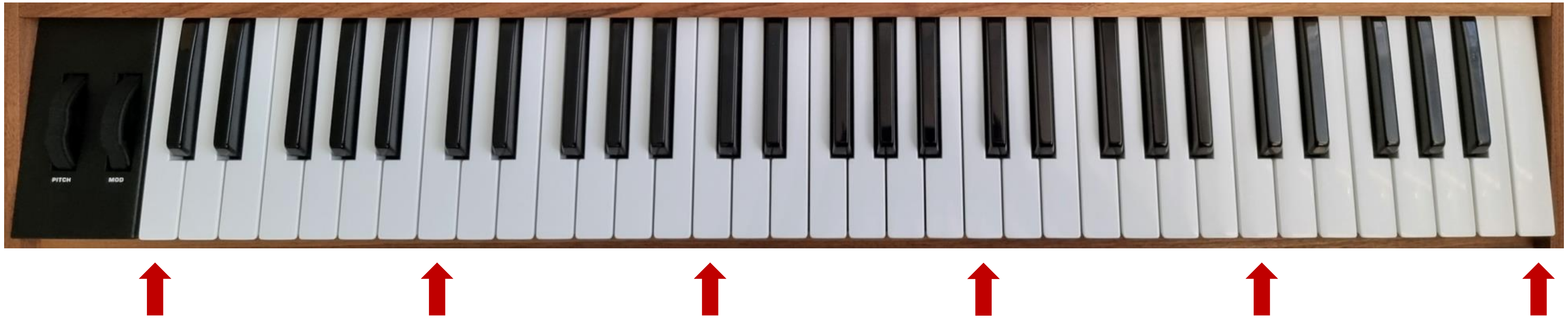
Band limiting

- 1 waveform for every single possible MIDI note: 127× 😲



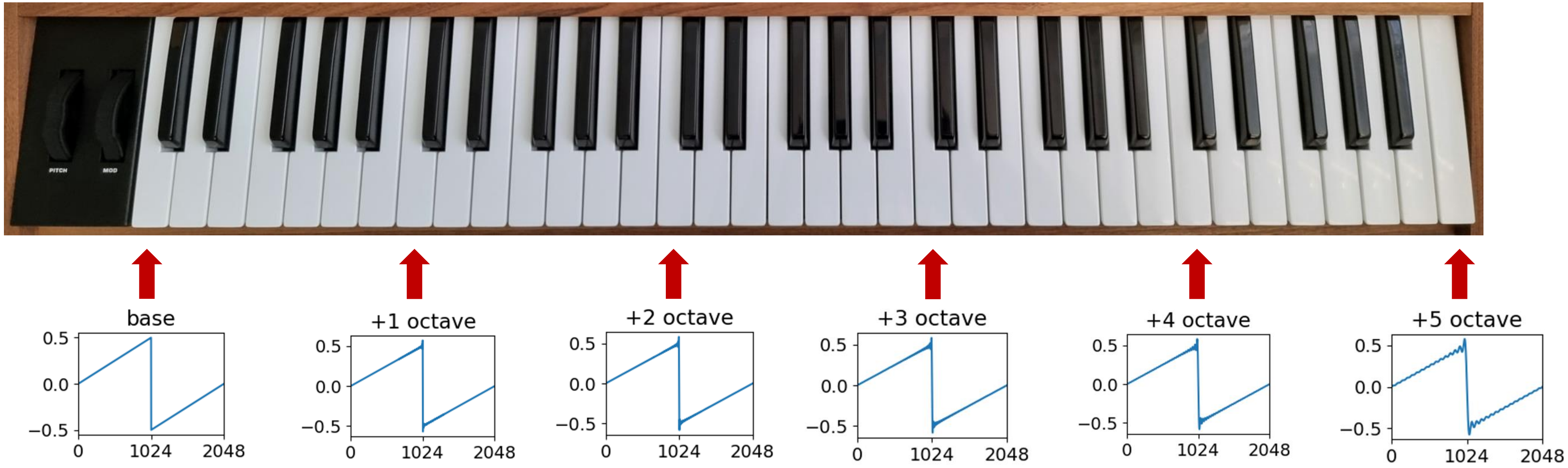
Band limiting

- ~~1 waveform for every single possible MIDI note~~
- 1 waveform for every octave

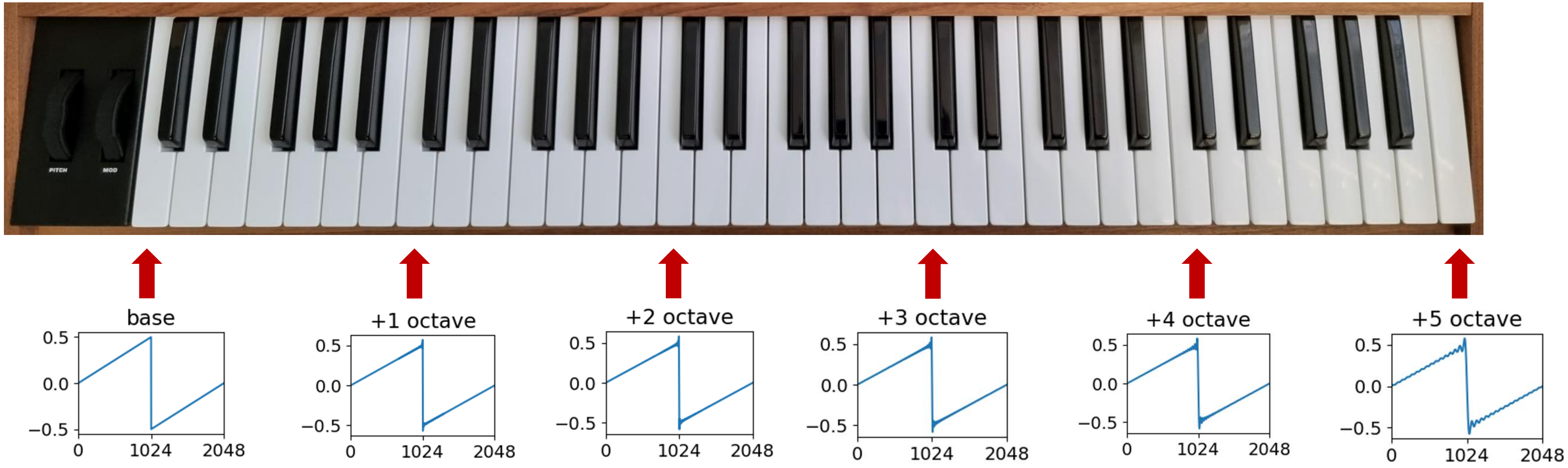


waveform → FFT → Remove harmonics → IFFT → band limited waveform

Band limiting

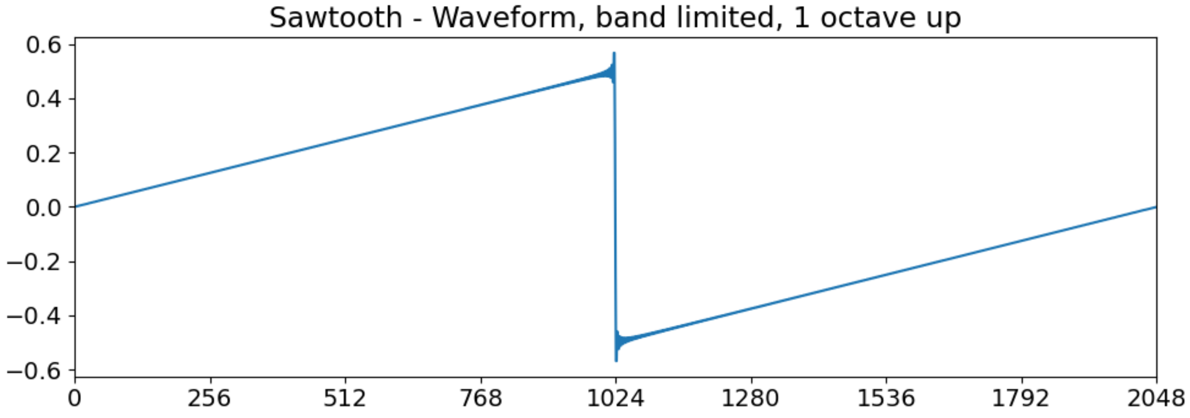


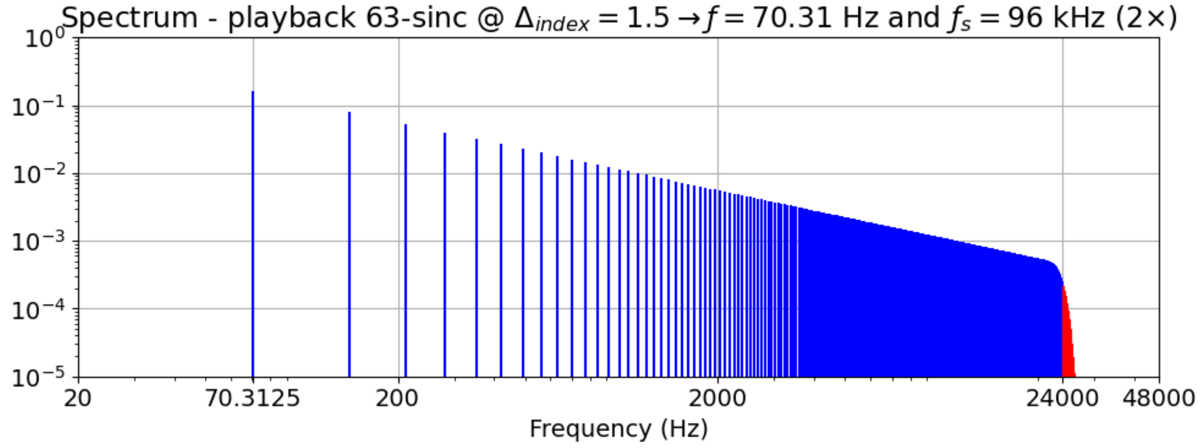
Band limiting

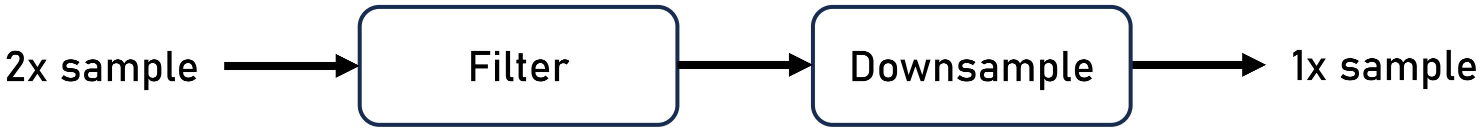


Oversampling & anti-aliasing *(yes, even more filtering!)*

- **Oversampling ⟶ spectral headroom without aliasing**
- **“🧱” Filter**







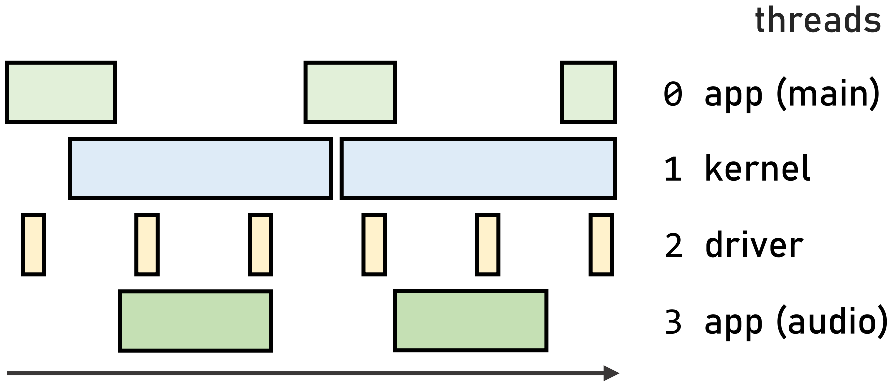
Wavetable synthesis

+

GPU programming

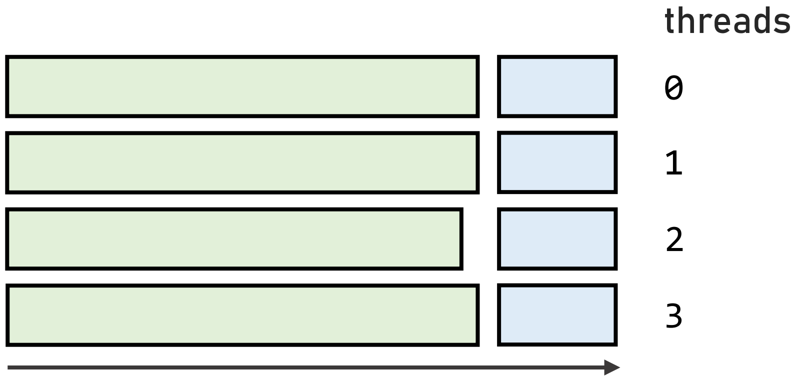
Parallelism

- **Task parallel (CPU)**



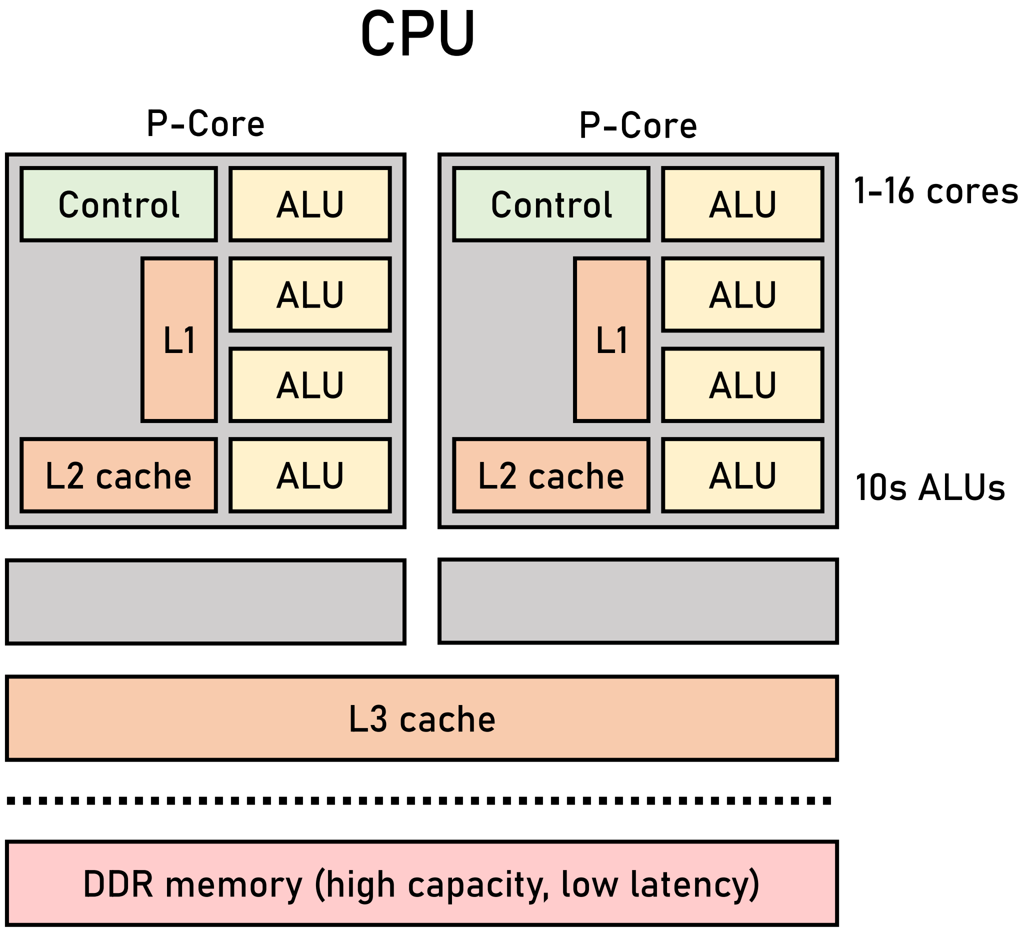
- Multi task, generic computation.
- Arbitrary, serial program flow.
- Scalability: 10s of threads.

- **Data parallel (GPU)**

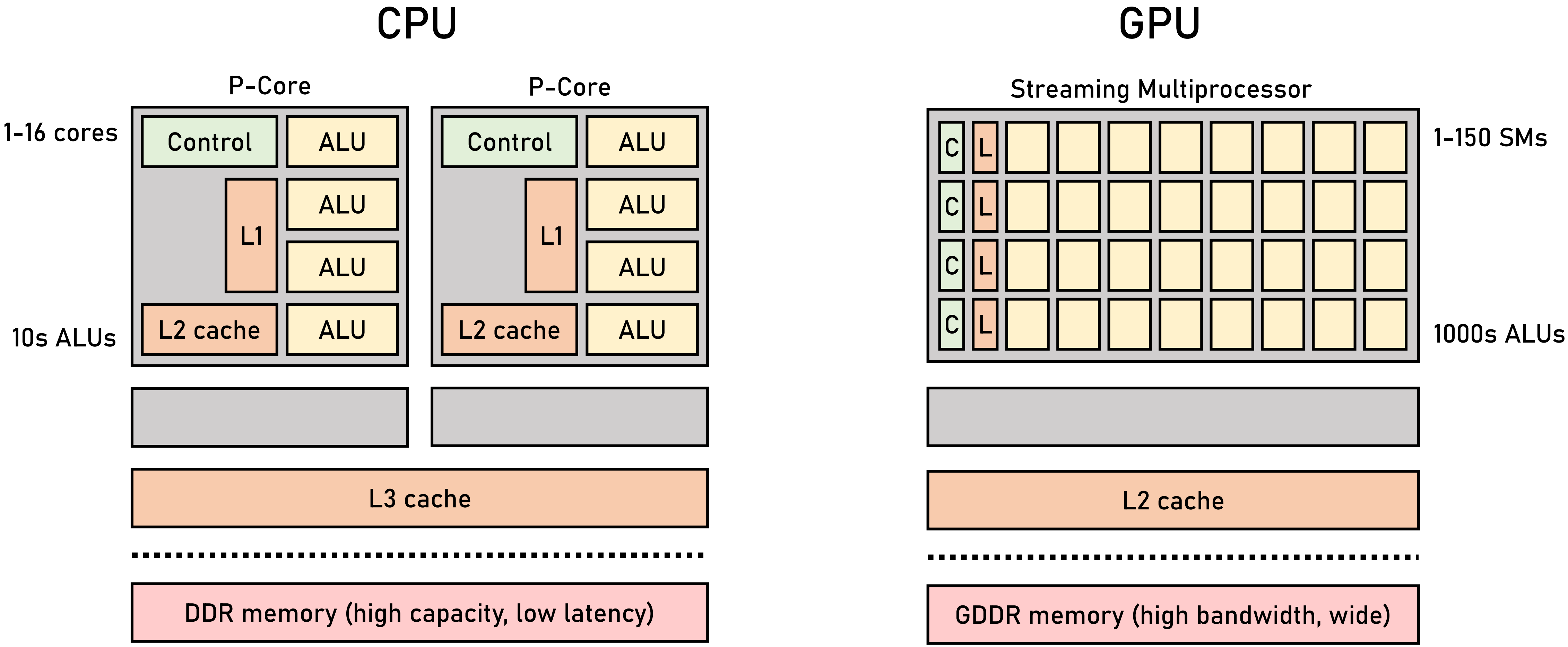


- Single task, **same computation**.
- Parallel program flow.
- Scalability: **1000s** of threads.

Core architecture

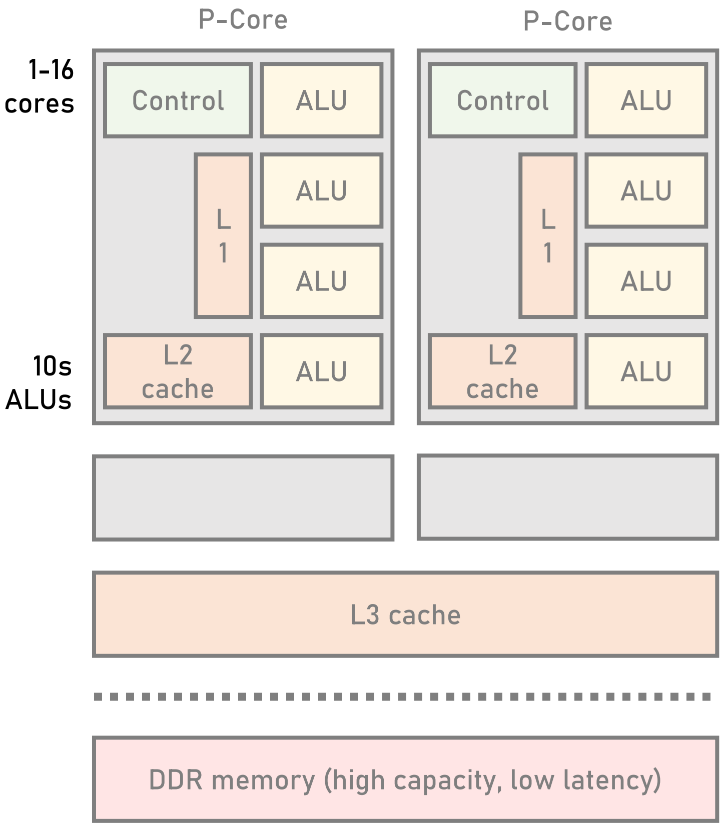


Core architecture

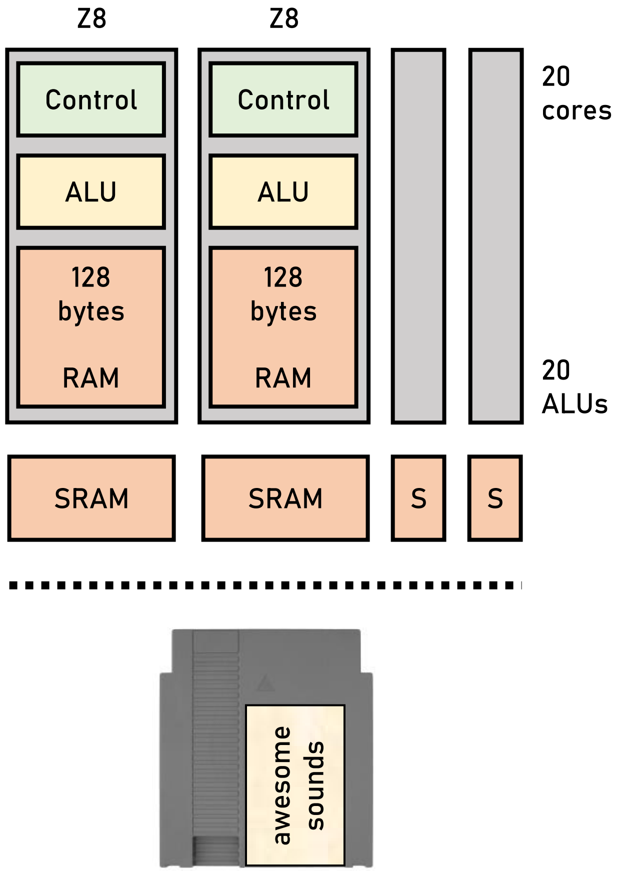


Core architecture

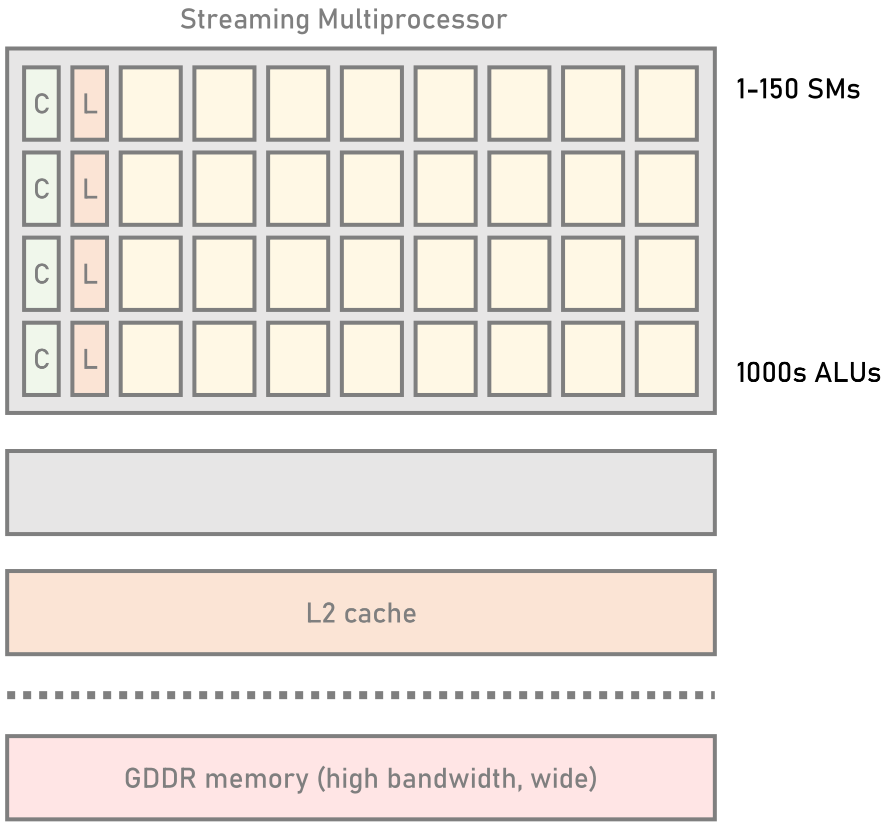
CPU



Wersi MK1 ('85)



GPU



Parallelism

Not all problems are suitable for parallel processing.

- Branch divergence

```
{
  int type = data[idx];
  if(type == 0) { … }
  if(type == 1) { … }
  if(type == 2) { … }
}
```

e.g. relational databases, graph traversal

- Data non-locality & dependency

```
{
  uint s = 0;
  for(int i=0;i<32;++i) {
    s = s ^ data1[hash(idx)];
  }
  …
}
```

e.g. cryptography (key derivation), complex reduction

Parallelism

Some problems are suitable for parallel processing.

- Image processing (shading)

```
{
  uint4 c = image[idx];
  c.rgb = dot(c.rgb,
    vec3(0.299,0.587,0.114)
  );
  image[idx] = c;
}
```

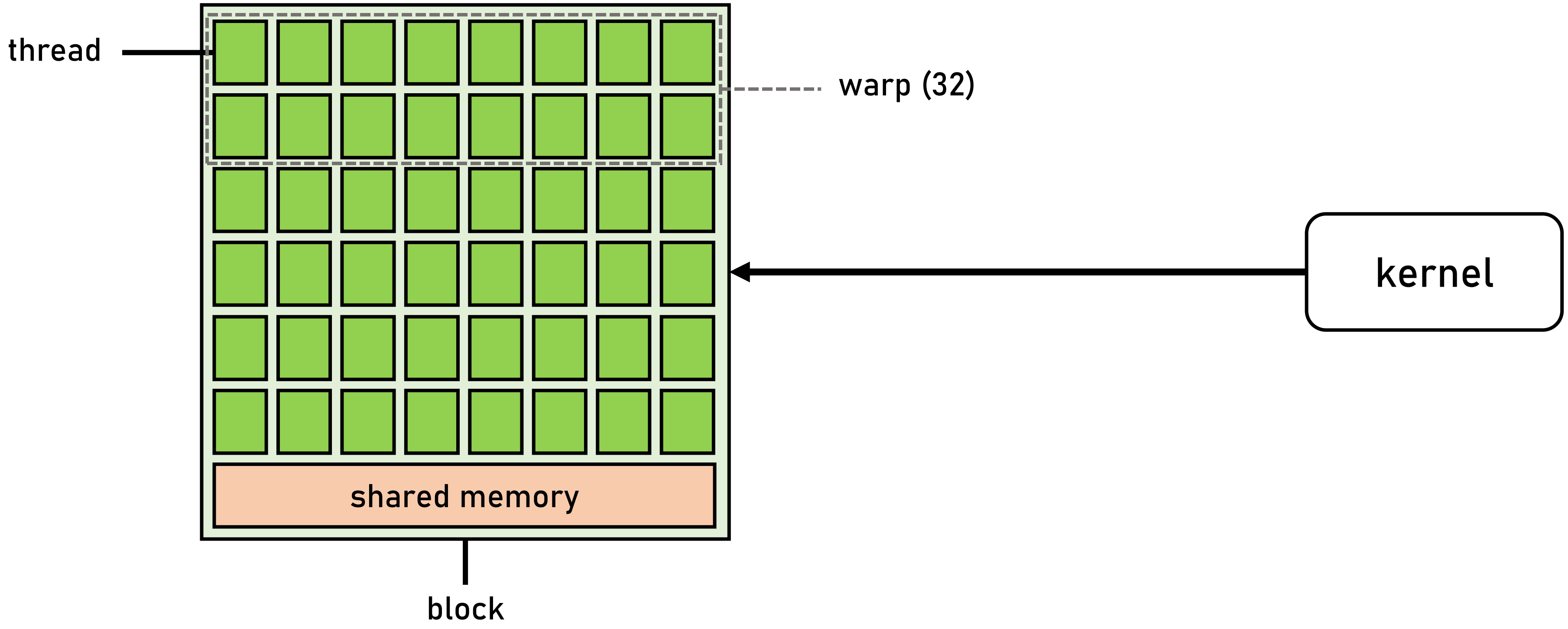
- Audio synthesis? 😺

```
{
  float x = osc(freq);
  x *= adsr();
}
```

Mapping to GPU

- thread
- warp (32 threads)
- block (1 to 2048 threads)

- Total workload divided into blocks.



GPU architecture

Platforms:

- NVIDIA CUDA
- OpenCL
- Vulkan Compute
- DirectCompute
- Metal Performance Shaders

Typical HW platforms:

NVIDIA	AMD
Intel	Apple
ARM Mali	PowerVR

GPU architecture

Platforms:

- NVIDIA CUDA
- OpenCL
- Vulkan Compute
- DirectCompute
- Metal Performance Shaders

Typical HW platforms:

NVIDIA
Intel
ARM Mali

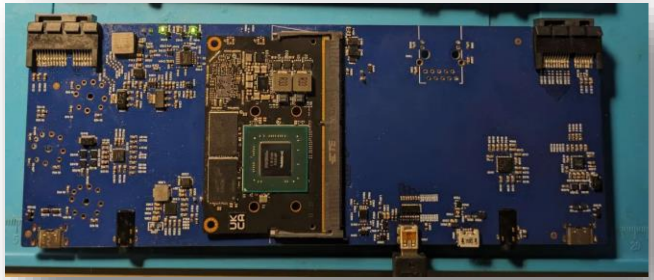
AMD
Apple
PowerVR

Why CUDA?

- More integrated environment & libraries
- Niche low-level intrinsics (e.g. `__half`, `__shfl`)

(But honestly..)

- Old school
- Working on GPU-based hardware synthesizer
    - NVIDIA Tegra SoC



CUDA

```
__global__ void kernel() {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```

```
int blocks = 1, threads = 1024;
kernel<<<blocks, threads, 0>>>();
```

CUDA

```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```

```
int blocks = 1, threads = 1024;
kernel<<<blocks, threads, 0>>>();
```

CUDA

```
__global__ void kernel(control_t* g_in,
                        output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaMalloc(&d_in,  n*sizeof(control_t));
cudaMalloc(&d_out, n*sizeof(output_t));
kernel<<<blocks, threads, 0>>>(d_in, d_out);
```

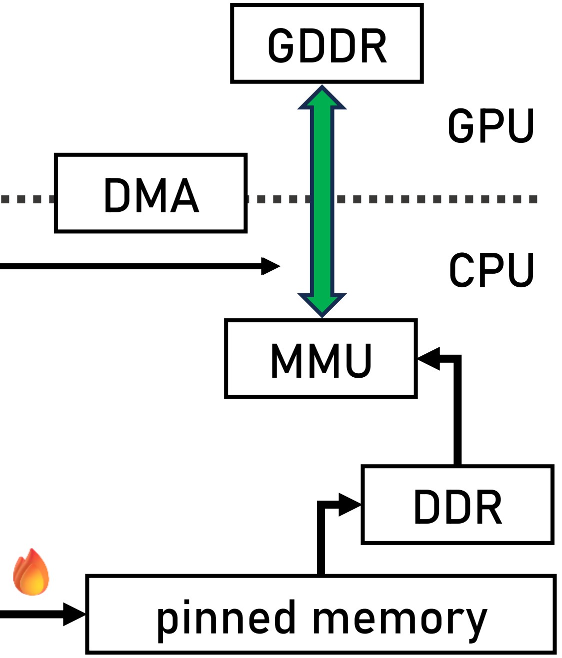
CUDA

```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaMalloc(&d_in,  n*sizeof(control_t));
cudaMalloc(&d_out, n*sizeof(output_t));
cudaMemcpy(d_in,  h_in,  n*sizeof(control_t), …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
cudaMemcpy(h_out, d_out, n*sizeof(output_t) , …);
```

CUDA

```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```



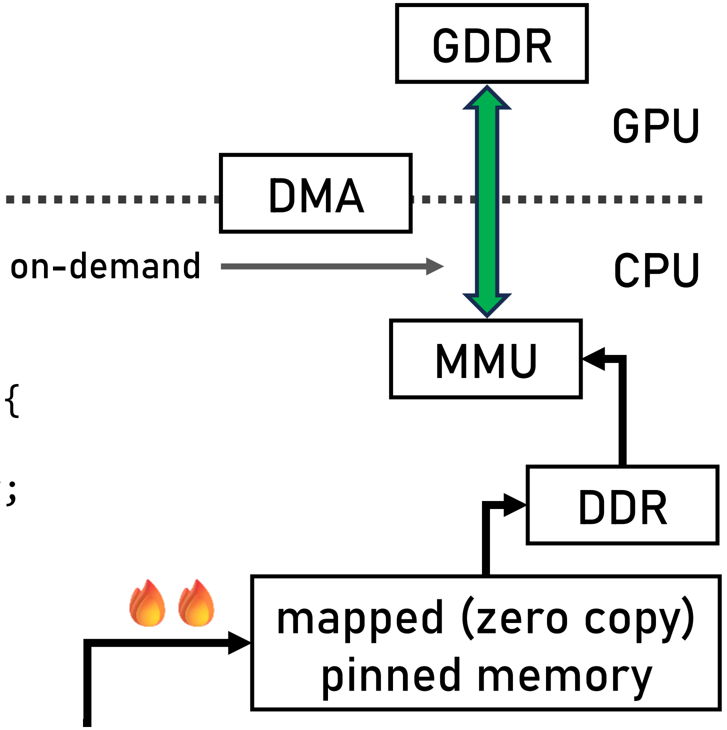

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaMalloc(&d_in,  n*sizeof(control_t));
cudaMalloc(&d_out, n*sizeof(output_t));
cudaMemcpy(d_in,  h_in,  n*sizeof(control_t), …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
cudaMemcpy(h_out, d_out, n*sizeof(output_t) , …);
```

```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};
```

```
cudaHostAlloc(&h_in,  n*sizeof(control_t));
cudaHostAlloc(&h_out, n*sizeof(output_t ));
```

CUDA

```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```




```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaMalloc(&d_in,  n*sizeof(control_t));
cudaMalloc(&d_out, n*sizeof(output_t));
cudaMemcpy(d_in,  h_in,  n*sizeof(control_t), …);
cudaHostGetDevicePointer(&d_in,  h_in,  …);
cudaHostGetDevicePointer(&d_out, h_out, …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
cudaMemcpy(h_out, d_out, n*sizeof(output_t) , …);
```

```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};

cudaHostAlloc(&h_in,  …, cudaHostAllocMapped);
cudaHostAlloc(&h_out, …, cudaHostAllocMapped);
```

CUDA

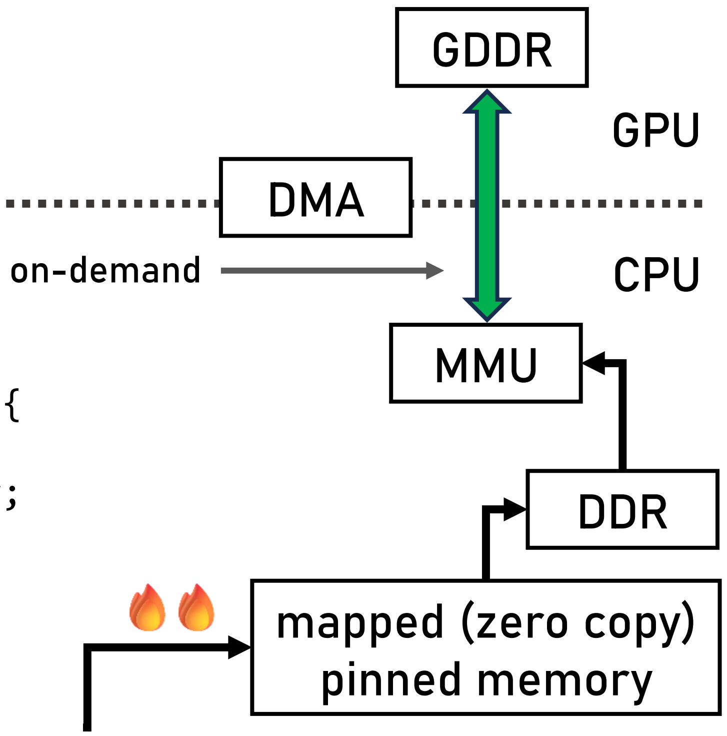
Need more performance?

- write-combined memory
- unified memory
- GPUDirect RDMA

```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaMalloc(&d_in,  n*sizeof(control_t));
cudaMalloc(&d_out, n*sizeof(output_t));
cudaMemcpy(d_in,  h_in,  n*sizeof(control_t), …);
cudaHostGetDevicePointer(&d_in,  h_in,  …);
cudaHostGetDevicePointer(&d_out, h_out, …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
cudaMemcpy(h_out, d_out, n*sizeof(output_t) , …);
```

```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};
```




```
cudaHostAlloc(&h_in,  …, cudaHostAllocMapped);
cudaHostAlloc(&h_out, …, cudaHostAllocMapped);
```

CUDA

- ✅ **Registers** (1 KB)
- ✅ **Global memory** (many GBs)

```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  g_out.sample[idx] = ?
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaHostGetDevicePointer(&d_in,  h_in,  …);
cudaHostGetDevicePointer(&d_out, h_out, …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
```

```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};

cudaHostAlloc(&h_in,  …, cudaHostAllocMapped);
cudaHostAlloc(&h_out, …, cudaHostAllocMapped);
```

CUDA

- ✅ Registers (1 KB)
- ✅ Global memory (many GBs)
- ✅ Shared memory (64-224 KB)

```
__shared__ state_t s_state[BLOCKSIZE];

__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  state_t& state = s_state[idx];
  g_out.sample[idx] = osc(c, state) * adsr(c, state);
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaHostGetDevicePointer(&d_in,  h_in,  …);
cudaHostGetDevicePointer(&d_out, h_out, …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
```

```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};

cudaHostAlloc(&h_in,  …, cudaHostAllocMapped);
cudaHostAlloc(&h_out, …, cudaHostAllocMapped);
```

CUDA

- ✅ Registers (1 KB)
- ✅ Global memory (many GBs)
- ✅ Shared memory (64-224 KB)
- ✅ Constant memory (64 KB)

```
__constant__ float c_ir[256];
__shared__ state_t s_state[BLOCKSIZE];
__global__ void kernel(control_t* g_in,
                        output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  state_t& state = s_state[idx];
  g_out.sample[idx] = filter(c_ir, osc(c, state) * adsr(c, state));
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaHostGetDevicePointer(&d_in,  h_in,  …);
cudaHostGetDevicePointer(&d_out, h_out, …);
kernel<<<blocks, threads, 0>>>(d_in, d_out);
```

```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};

cudaHostAlloc(&h_in,  …, cudaHostAllocMapped);
cudaHostAlloc(&h_out, …, cudaHostAllocMapped);
cudaMemcpyToSymbol(&c_ir, h_ir, …);
```

CUDA

- ✅ Registers (1 KB)
- ✅ Global memory (many GBs)
- ✅ Shared memory (64-224 KB)
- ✅ Constant memory (64 KB)
- ✅ Texture memory (many GBs)

```
__constant__ float c_ir[256];
__shared__ state_t s_state[BLOCKSIZE];
__global__ void kernel(control_t* g_in,
                        output_t* g_out,
              cudaTextureObject_t  t_wave) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  state_t& state = s_state[idx];
  g_out.sample[idx] = filter(c_ir, osc(c, state, t_wave) * adsr(c, state));
}
```

```
int blocks = 1, threads = 1024, n = blocks*threads;
cudaHostGetDevicePointer(&d_in,  h_in,  …);
cudaHostGetDevicePointer(&d_out, h_out, …);
kernel<<<blocks, threads, 0>>>(d_in, d_out, t_wave);
```

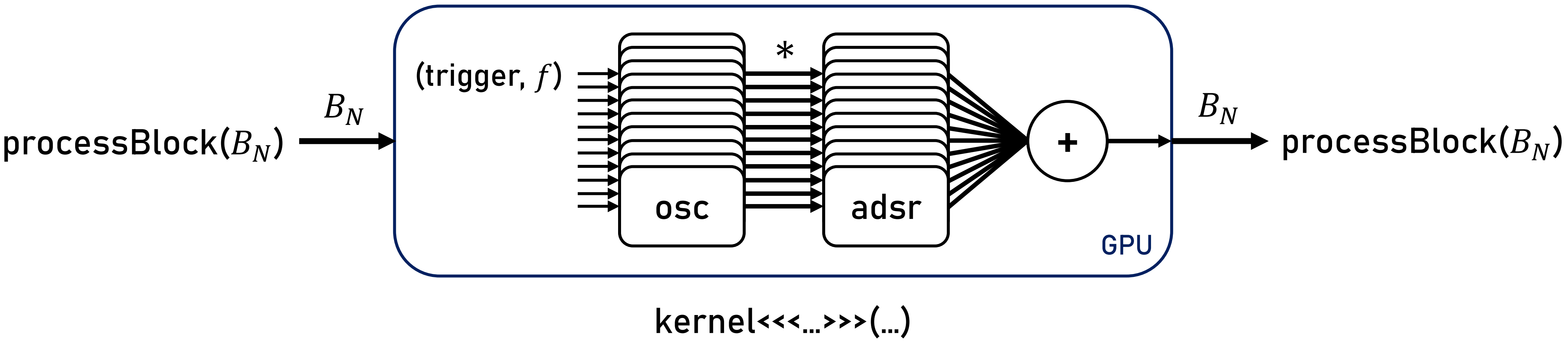
```
struct control_t {
  bool trigger;
  float frequency;
};
struct output_t {
  float2 sample;
};

cudaHostAlloc(&h_in,  …, cudaHostAllocMapped);
cudaHostAlloc(&h_out, …, cudaHostAllocMapped);
cudaMemcpyToSymbol(&c_ir, h_ir, …);
cudaCreateTextureObject(t_wave, …);
```

Wavetable synthesis

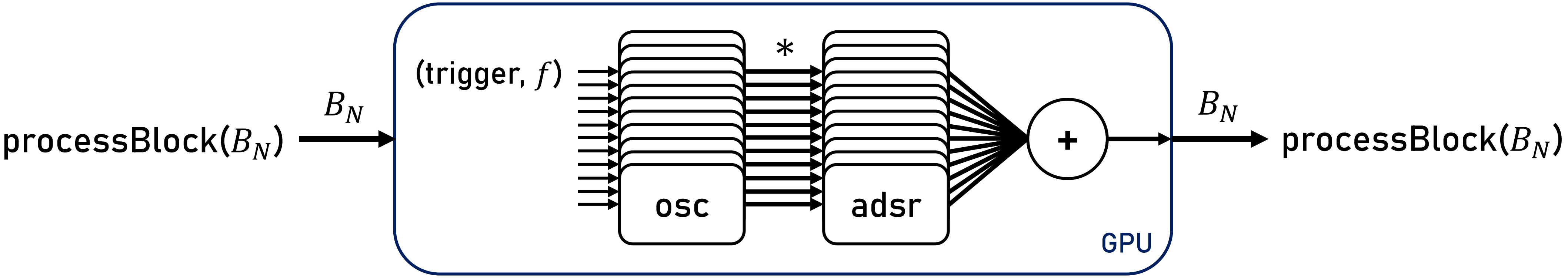
+

GPU programming

Signal chain



Signal chain



kernel<<<…>>>(…)

- Kernel launch strategy
  - Persistent kernel
  - Short kernel

Kernel launch

```
__global__ void kernel(control_t* g_in,
                        output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  while(1) {
    control_t c = wait_and_fetch(g_in[idx]);
    float2 sample = osc(c, …) * adsr(c, …));
    write_and_signal(g_out[idx], sample);
  }
}
```

persistent_kernel<<<…>>>(…)

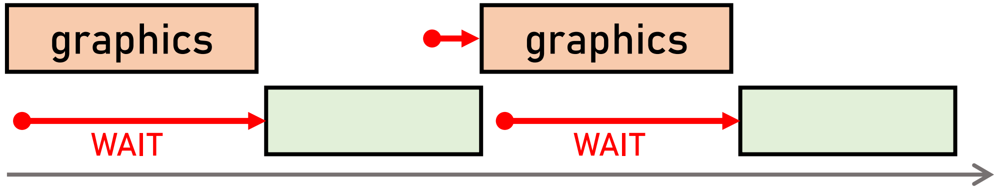
Ideal

- Runs "forever" on GPU.
- Can exclusively lock GPU.
- GPU always ready to start immediately.

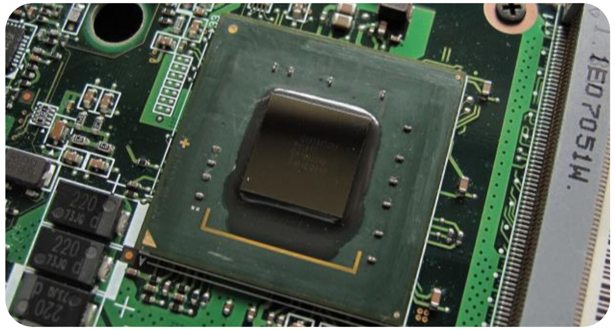
Kernel launch

```
__global__ void kernel(control_t* g_in,
                        output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  while(1) {
    control_t c = wait_and_fetch(g_in[idx]);
    float2 sample = osc(c, …) * adsr(c, …));
    write_and_signal(g_out[idx], sample);
  }
}
```

- ~~Runs "forever" on GPU.~~
- ~~Can exclusively lock GPU.~~
- ~~GPU always ready to start immediately.~~



Ideal



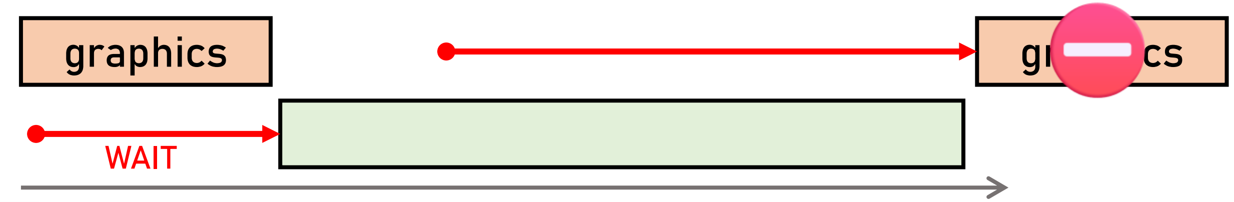
Reality

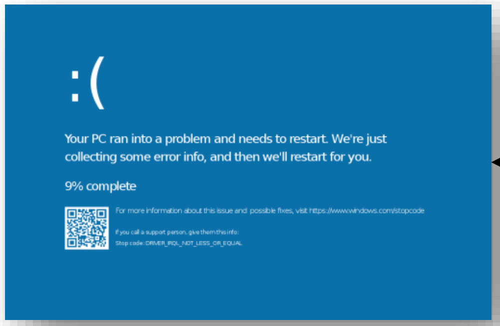
Kernel launch

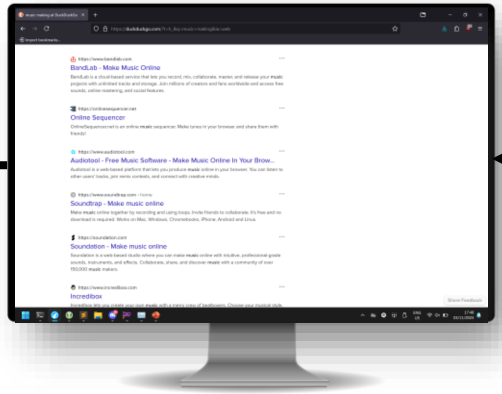
```
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  while(1) {
    control_t c = wait_and_fetch(g_in[idx]);
    float2 sample = osc(c, …) * adsr(c, …));
    write_and_signal(g_out[idx], sample);
  }
}
```

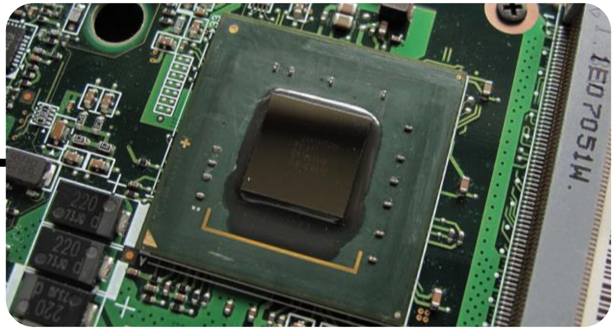
- ~~Runs “forever” on GPU.~~
- ~~Can exclusively lock GPU.~~
- ~~GPU always ready to start immediately.~~

GPU = shared (integrated) device 😳









Reality

Kernel launch

```
const int N = 512; // buffer samples
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    g_out.sample[idx][i] = osc(c, ...) * adsr(c, ...);
  }
}
```

short_kernel<<<…>>>(…)

```
void processBlock(buffer) {
  short_kernel<<<…>>>(…); // launch & wait
}
```

- Runs once, processes buffer on GPU.
- No exclusivity, pre-emption possible.
- GPU may already be busy.

Kernel launch

```
const int N = 512; // buffer samples
__global__ void kernel(control_t* g_in,
                       output_t* g_out) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    g_out.sample[idx][i] = osc(c, ...) * adsr(c, ...);
  }
}
```

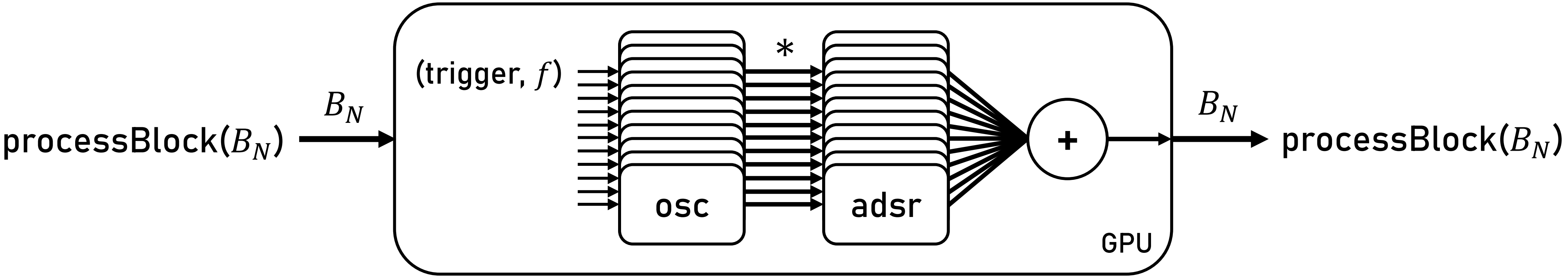
short_kernel<<<…>>>(…)

```
void processBlock(buffer) {
  short_kernel<<<…>>>(…); // launch & wait
}
```

- Runs once, short, processes buffer on GPU.
- No exclusivity, pre-emption possible.
- GPU may be busy.

- Audio buffer scheduling:
  - 512 samples (~10 ms @ 48000 Hz)
  - 1 launch every ~93 Hz
  - GPU completion deadline ~10 ms
- Measurable

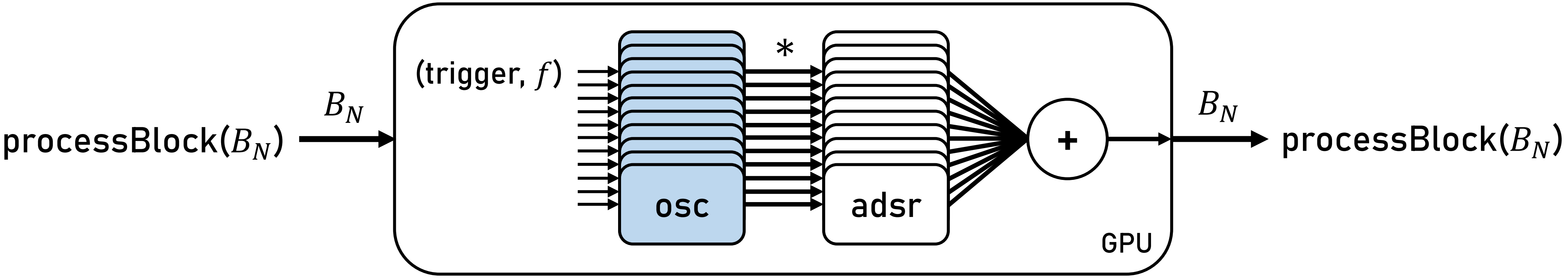
Signal chain



short_kernel<<<…>>>(…)

Up to 1024 oscillators (threads per block)!

Oscillator



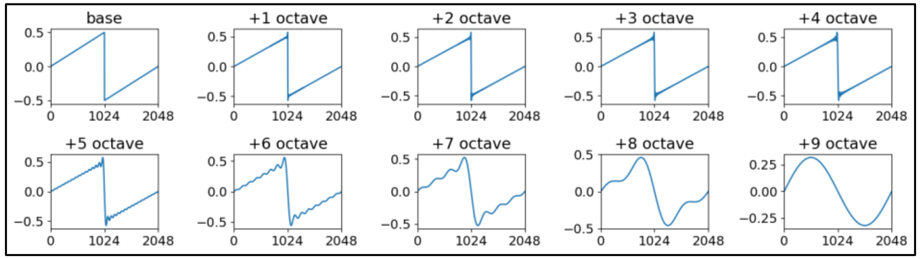

```
__device__ float osc(float frequency,
                     float phase,
                     float samplerate,
                     float cycle,
                     cudaTextureObject_t t_wave,
                     state_t& s_state)
{
  float u = fmod(s_state.osc.t + phase, 1.0f) * float(2048);
  float v = cycle;
  int   w = wavetableIndex(samplerate, frequency);
  sample = tex2DLayered<float>(t_wave, u, v, w);

  float delta = frequency / samplerate;
  s_state.osc.t = fmod(s_state.osc.t + delta, 1.0f);
}
```

2D layered texture = band limited wavetable

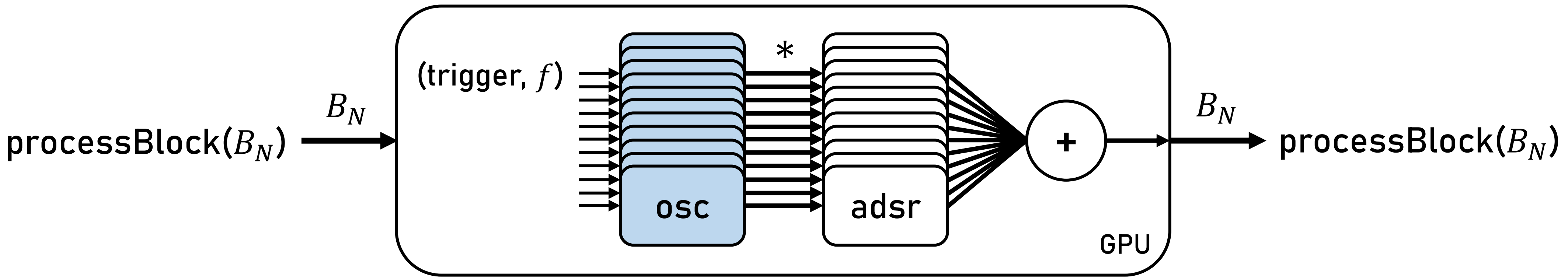
2048 (samples) x 256 (cycles) x 10 (octaves)

= 21 MiB

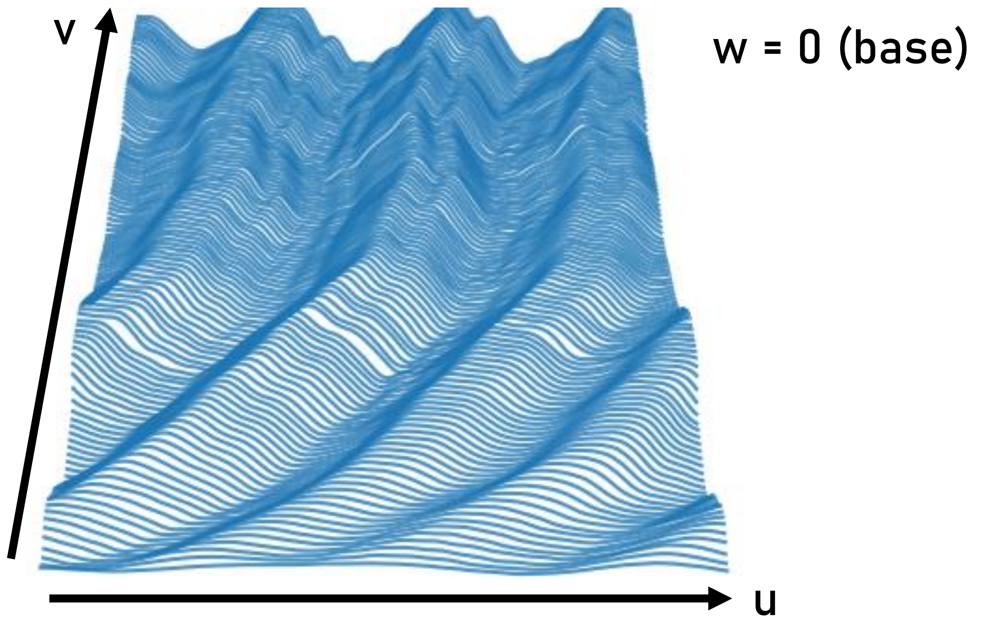
Maximum size: 16384 x 16384 x 2048



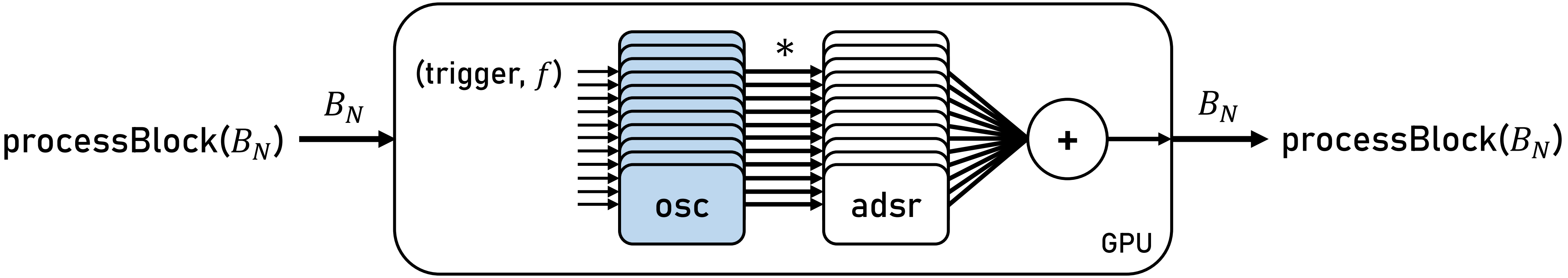
Oscillator



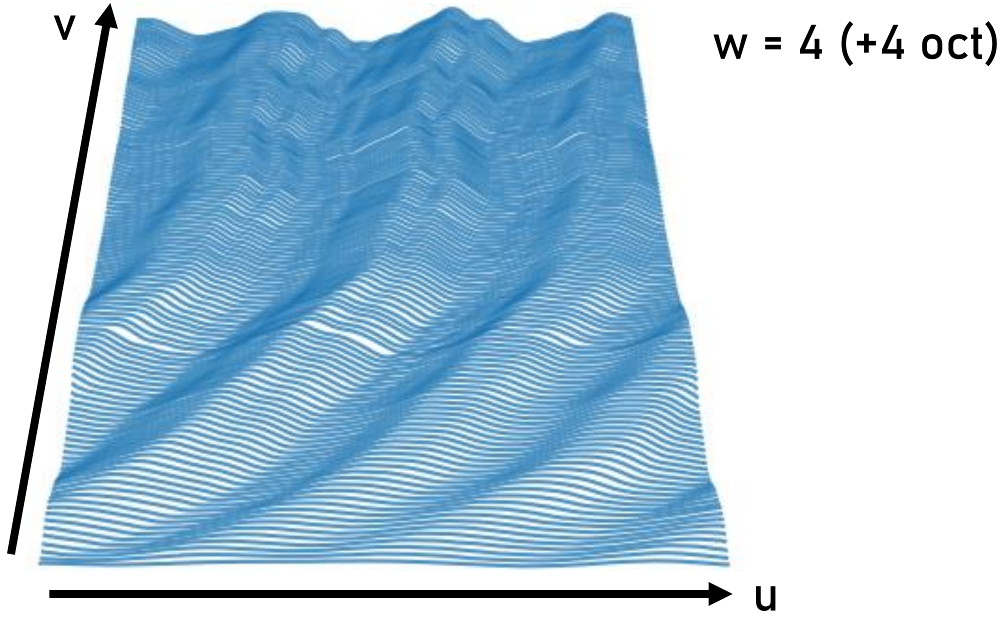

```
__device__ float osc(float frequency,
                     float phase,
                     float samplerate,
                     float cycle,
                     cudaTextureObject_t t_wave,
                     state_t& s_state)
{
  float u = fmod(s_state.osc.t + phase, 1.0f) * float(2048);
  float v = cycle;
  int   w = wavetableIndex(samplerate, frequency);
  sample = tex2DLayered<float>(t_wave, u, v, w);

  float delta = frequency / samplerate;
  s_state.osc.t = fmod(s_state.osc.t + delta, 1.0f);
}
```



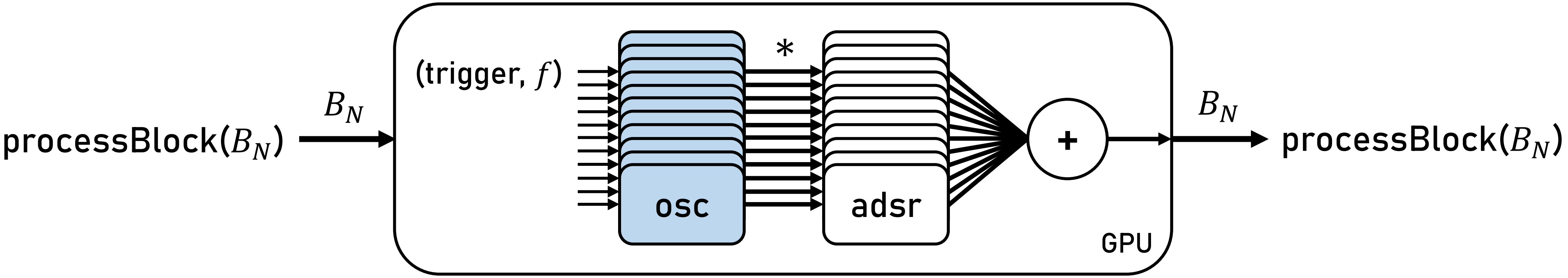
Oscillator



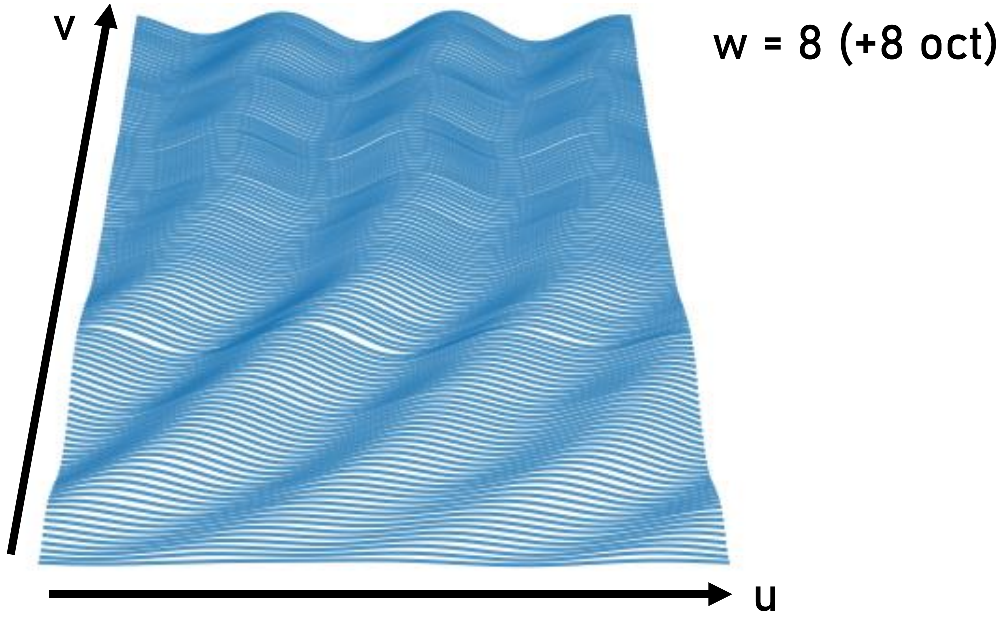

```
__device__ float osc(float frequency,
                     float phase,
                     float samplerate,
                     float cycle,
                     cudaTextureObject_t t_wave,
                     state_t& s_state)
{
  float u = fmod(s_state.osc.t + phase, 1.0f) * float(2048);
  float v = cycle;
  int   w = wavetableIndex(samplerate, frequency);
  sample = tex2DLayered<float>(t_wave, u, v, w);

  float delta = frequency / samplerate;
  s_state.osc.t = fmod(s_state.osc.t + delta, 1.0f);
}
```



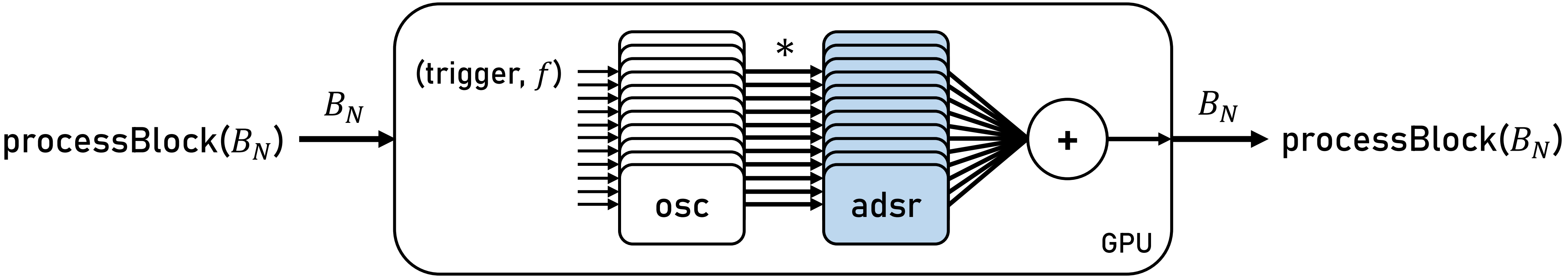
Oscillator




```
__device__ float osc(float frequency,
                     float phase,
                     float samplerate,
                     float cycle,
                     cudaTextureObject_t t_wave,
                     state_t& s_state)
{
  float u = fmod(s_state.osc.t + phase, 1.0f) * float(2048);
  float v = cycle;
  int   w = wavetableIndex(samplerate, frequency);
  sample = tex2DLayered<float>(t_wave, u, v, w);

  float delta = frequency / samplerate;
  s_state.osc.t = fmod(s_state.osc.t + delta, 1.0f);
}
```

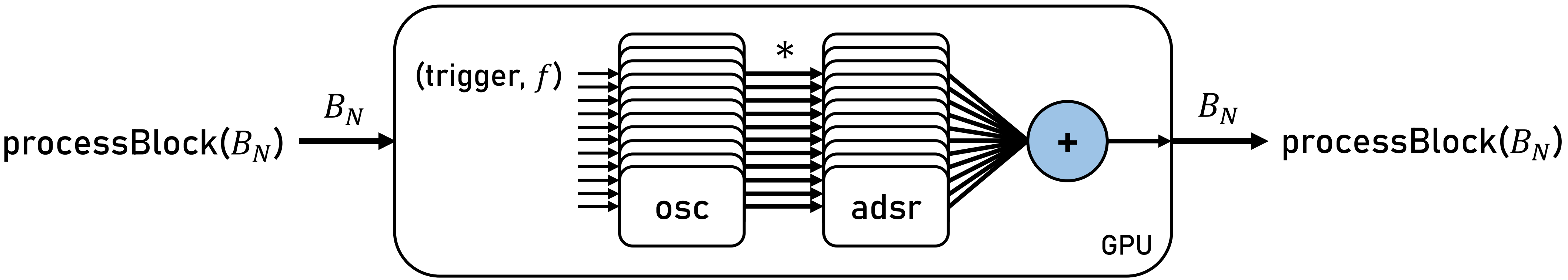


ADSR (*VCA*)



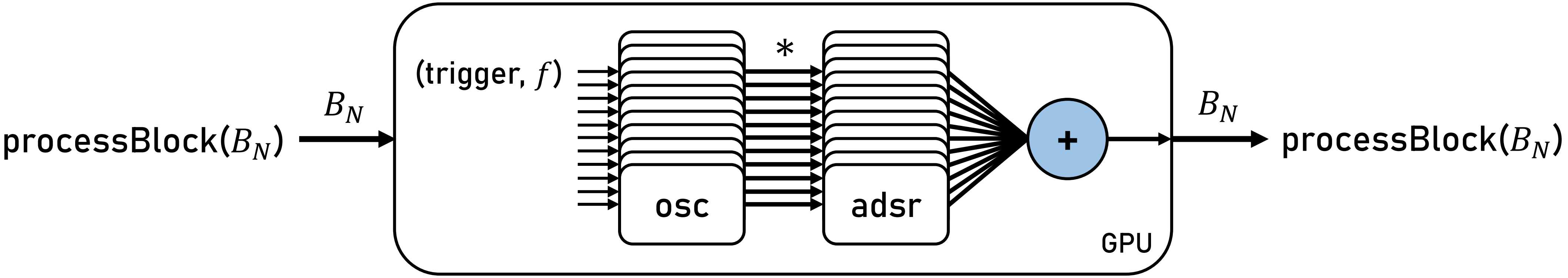

```
__device__ float adsr(float a,
                      float d,
                      float s,
                      float r,
                      float samplerate,
                      state_t& s_state)
{
  float t = s_state.adsr.t;
  s_state.adsr.t += 1.0f / samplerate;
  if (t < a) { y = t / a };                                // attack
  else if (t < a + d) { y = 1 - (1 - s) * (t - a) / d; } // decay
  else if (t < 1 - r) { y = s };                          // sustain
  else { y = s * (1 - (t - (1 - r)) / r) };               // release
  return y;
}
```

Parallel sum

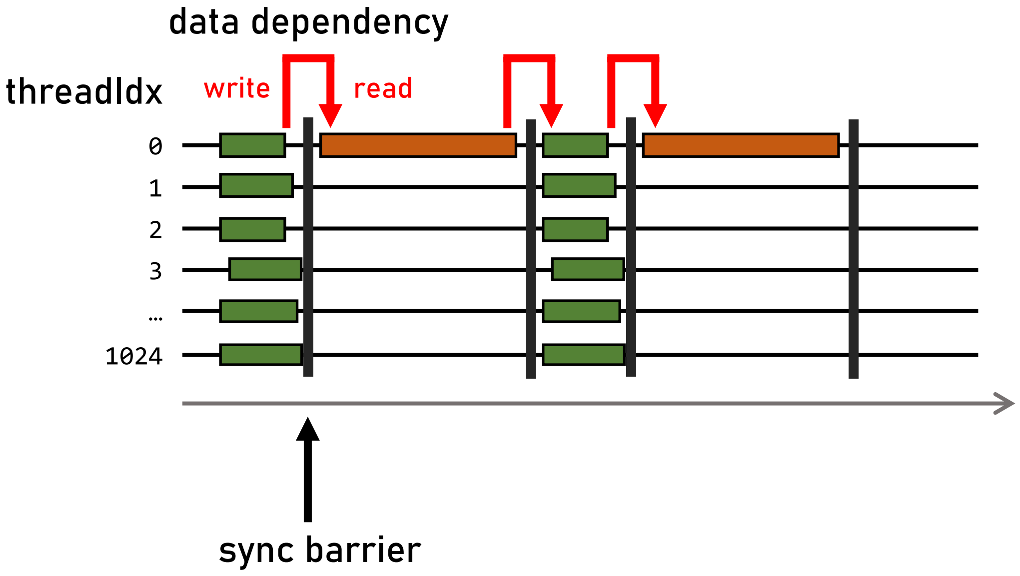



```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    g_out.sample[blockIdx.x] = osc(c, ...) * adsr(c, ...);   } A

    __syncthreads();
    if (threadIdx.x == 0) { // 1 summing thread
      float sum = 0;
      for(int i = 0; i < blockDim.x; ++i) {
        sum += g_out.sample[i];
      }                                                       } B
      g_out.sample[0] = sum;
    }
   __syncthreads();
  }
}
```

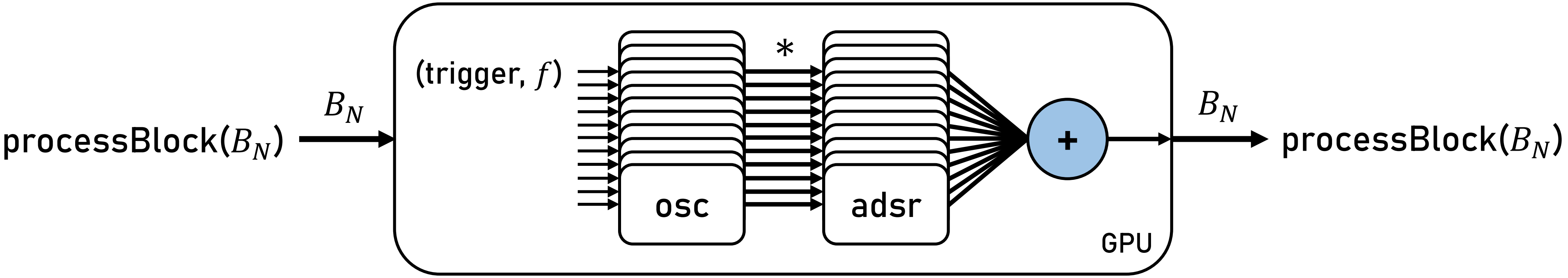
Parallel sum



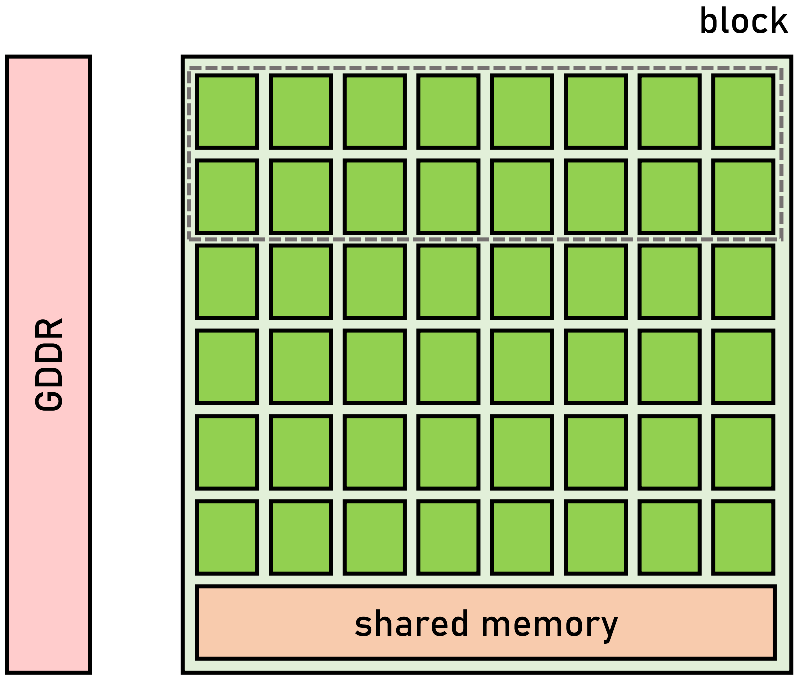

```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    g_out.sample[blockIdx.x] = osc(c, ...) * adsr(c, ...);   } A

    __syncthreads();
    if (threadIdx.x == 0) { // 1 summing thread
      float sum = 0;
      for(int i = 0; i < blockDim.x; ++i) {
        sum += g_out.sample[i];
      }                                                       } B
      g_out.sample[0] = sum;
    }
   __syncthreads();
  }
}
```



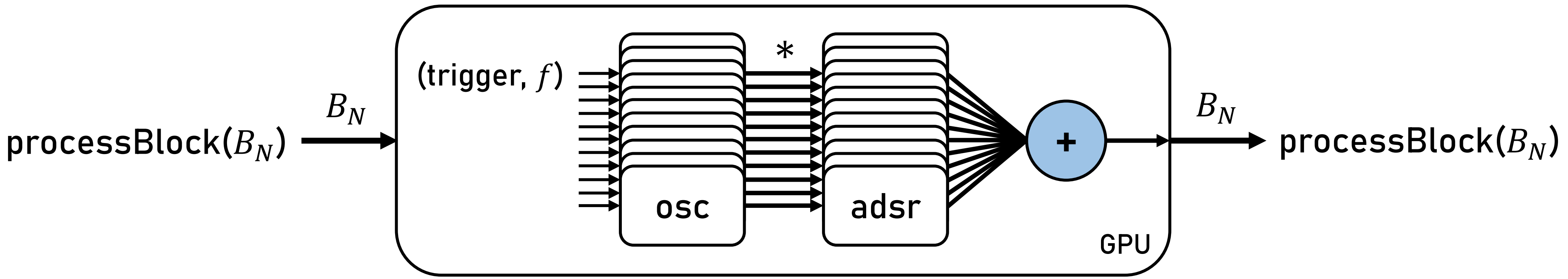
Parallel sum



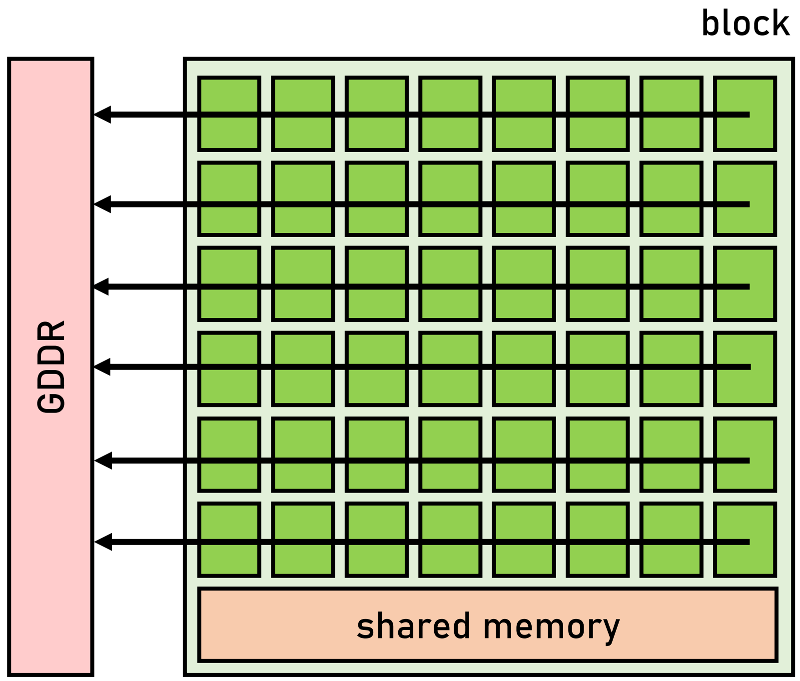

```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    g_out.sample[blockIdx.x] = osc(c, ...) * adsr(c, ...);

    __syncthreads();
    if (threadIdx.x == 0) { // 1 summing thread
      float sum = 0;
      for(int i = 0; i < blockDim.x; ++i) {
        sum += g_out.sample[i];
      }
      g_out.sample[0] = sum;
    }
   __syncthreads();
  }
}
```



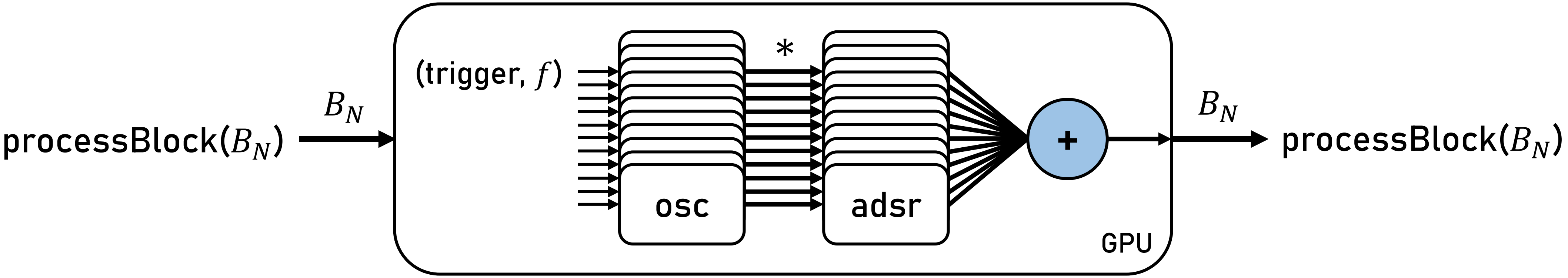
Parallel sum



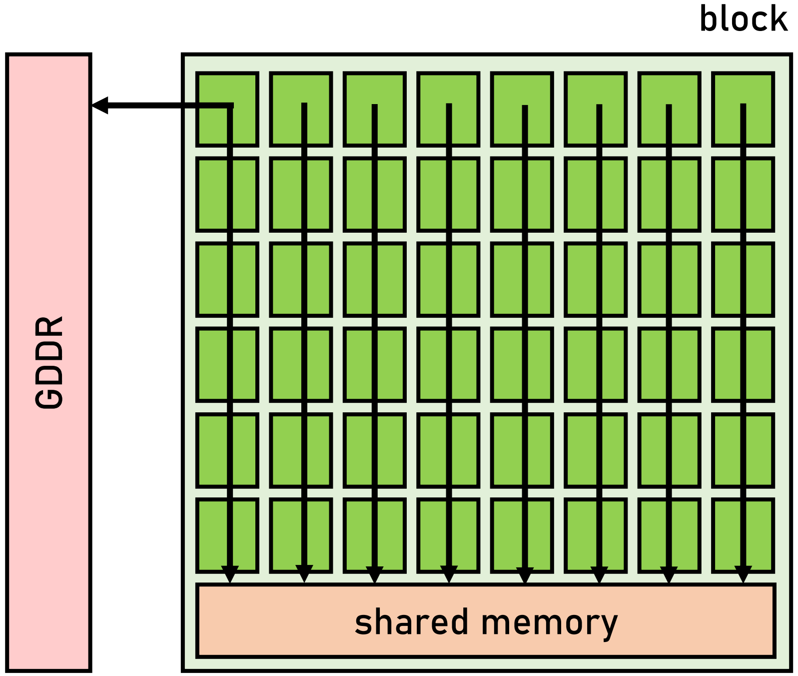

```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    g_out.sample[blockIdx.x] = osc(c, ...) * adsr(c, ...);

    __syncthreads();
    if (threadIdx.x == 0) { // 1 summing thread
      float sum = 0;
      for(int i = 0; i < blockDim.x; ++i) {
        sum += g_out.sample[i];
      }
      g_out.sample[0] = sum;
    }
   __syncthreads();
  }
}
```



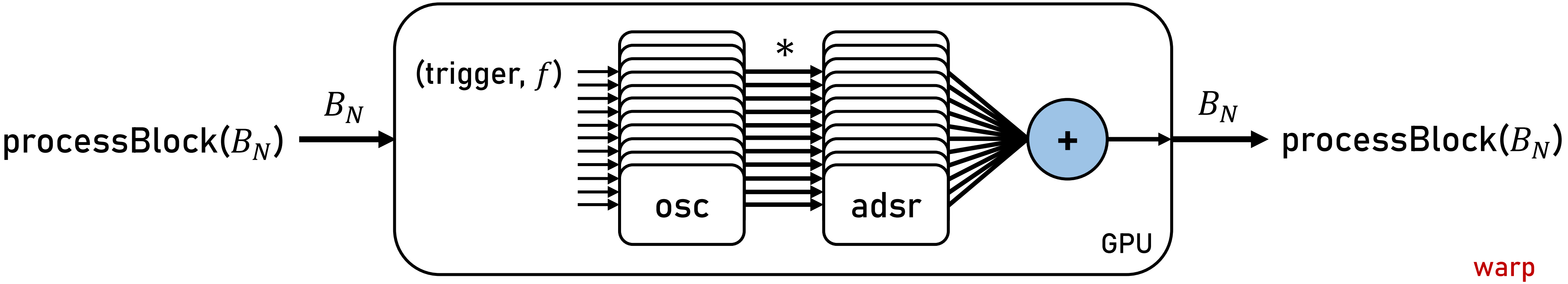
Parallel sum



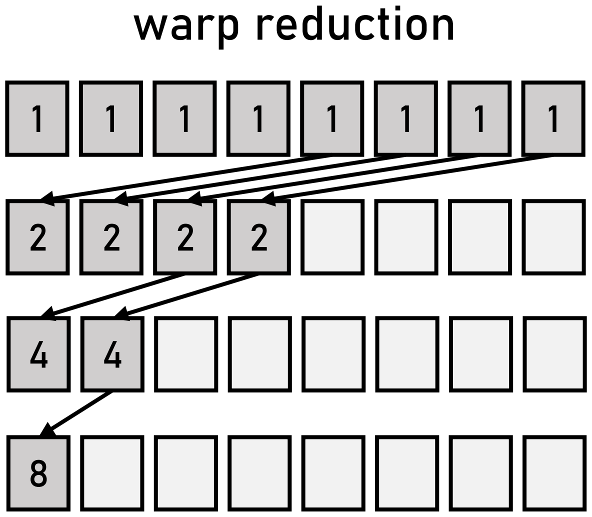

```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  __shared__ float2 s_samples[blockDim.x];
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    s_sample[i] = osc(c, ...) * adsr(c, ...);
    __syncthreads();
    if (threadIdx.x == 0) { // 1 summing thread
      float sum = 0;
      for(int i = 0; i < blockDim.x; ++i) {
        sum += s_sample[i];
      }
      g_out.sample[0] = sum;
    }
   __syncthreads();
  }
}
```

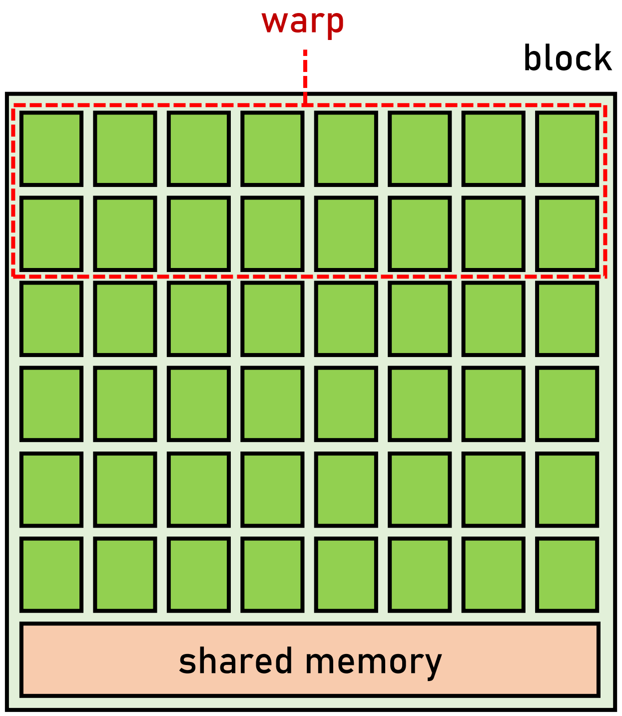


Parallel sum

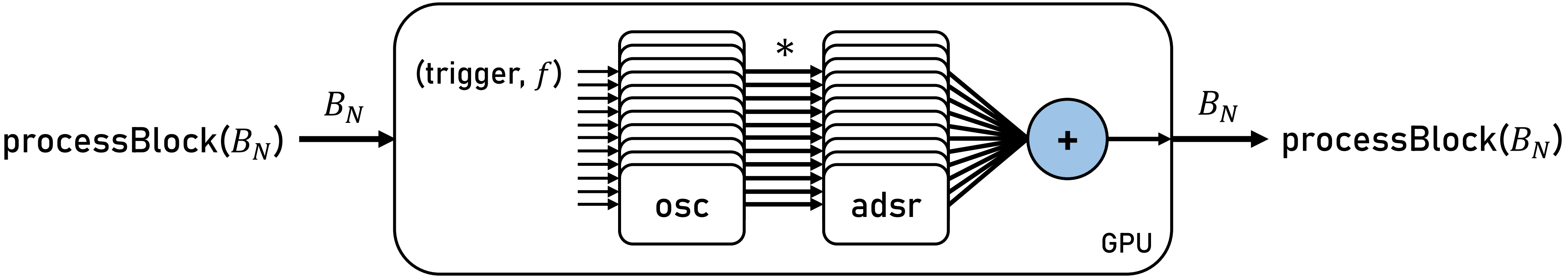



```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    float2 sample = osc(c, ...) * adsr(c, ...);
    __syncthreads();
    sample = blockSum(sample); // parallel block reduction
    if (threadIdx.x == 0) {    // 1 global write thread
      g_out.sample[0] = sample;
    }
  }
}
```

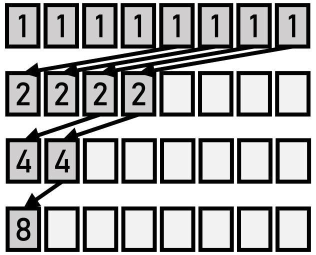




Parallel sum



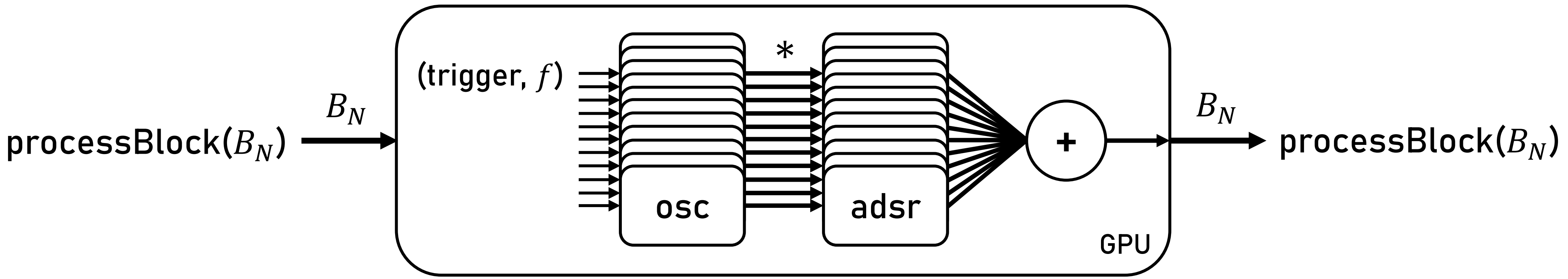

```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  const control_t c = g_in[idx];
  for(int i = 0; i < N; ++i) {
    float2 sample = osc(c, ...) * adsr(c, ...);
    __syncthreads();
    sample = blockSum(sample); // parallel block reduction
    if (threadIdx.x == 0) {    // 1 global write thread
      g_out.sample[0] = sample;
    }
  }
}
```



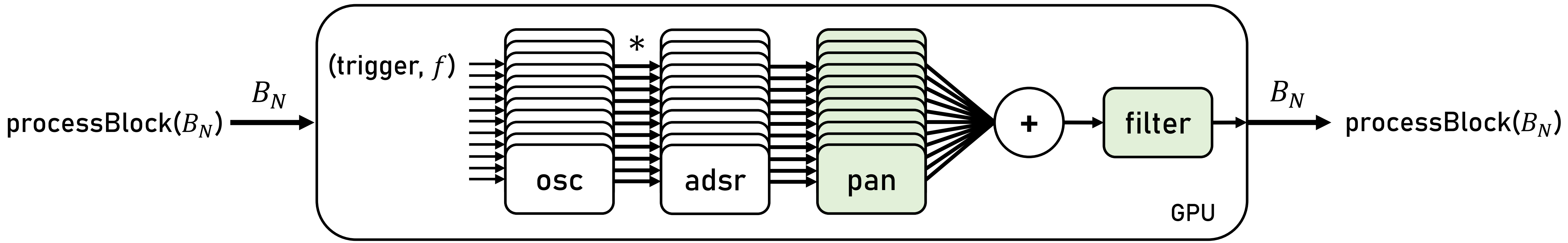

```
__device__ T blockSum(T val) {
  __shared__ T sum[32];
  int laneId = threadIdx.x%32, warpId = threadIdx.x/32;
  val = warpSum(val);
  if (laneId == 0) { sum[warpId] = val; }
  __syncthreads();
  val = sum[laneId];
  if (warpId == 0) { val = warpSum(val); }
  return val;
}
__device__ T warpSum(T sum) {
  sum += __shfl_down_sync(~0U, sum, 16);
  sum += __shfl_down_sync(~0U, sum, 8);
  sum += __shfl_down_sync(~0U, sum, 4);
  sum += __shfl_down_sync(~0U, sum, 2);
  sum += __shfl_down_sync(~0U, sum, 1);
  return sum;
}
```

warp primitives

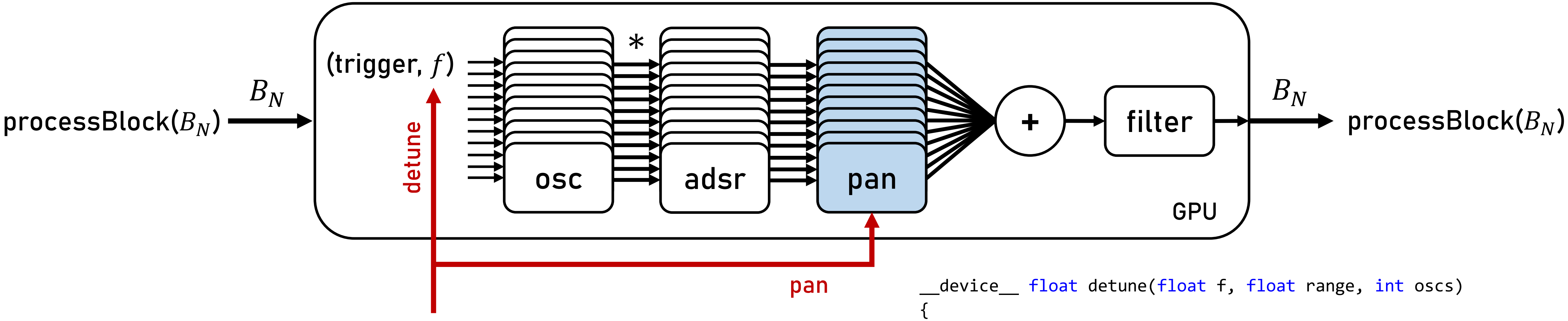
Signal chain



Signal chain

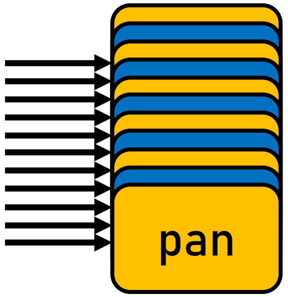


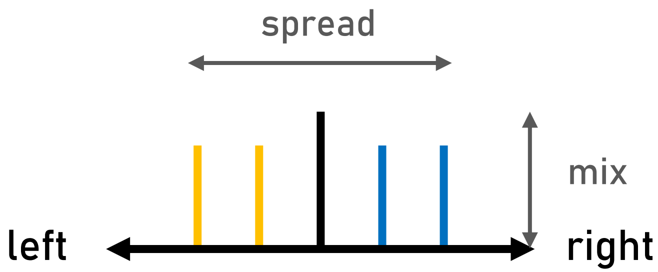
“Unison”



e.g. Roland JP-8000
“super saw” inspired

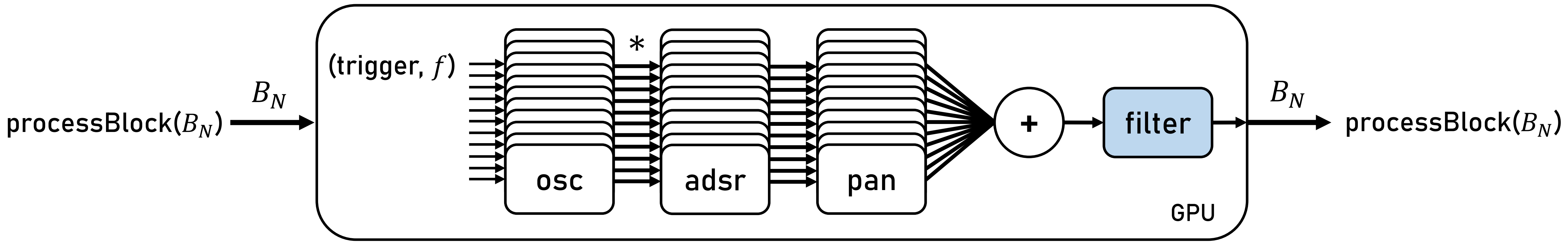
same computation

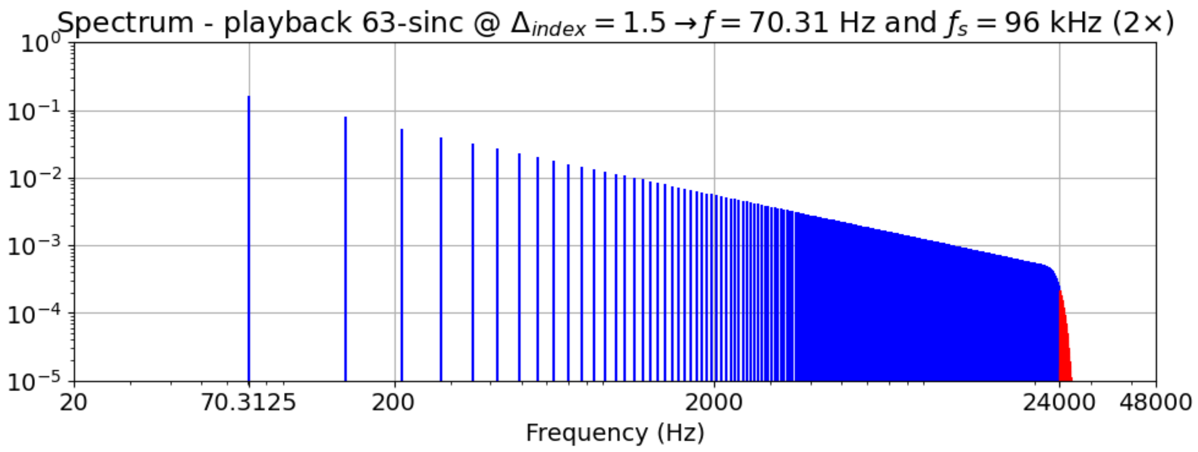




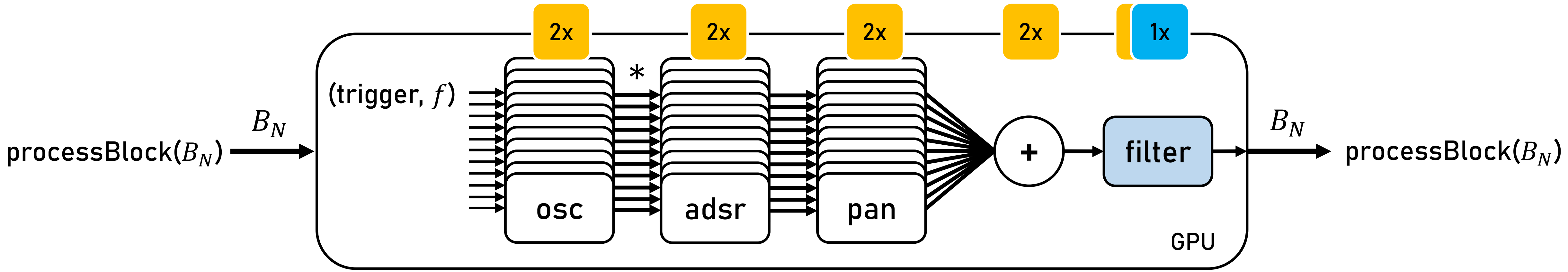

```
__device__ float detune(float f, float range, int oscs)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float t = float(idx) / float(oscs-1);
  float p = range * (-0.5f + t);
  return f * exp2f(range * (1.0f/1200.0f));
}

__device__ float2 pan(float2 s, int oscs)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float t = float(idx) / float(oscs-1);
  float angle = spread(t);
  s.x *= 0.5f*sqrt(2) * (cosf(angle)-sinf(angle)) * blend(t);
  s.y *= 0.5f*sqrt(2) * (cosf(angle)+sinf(angle)) * blend(t);
  return s;
}
```

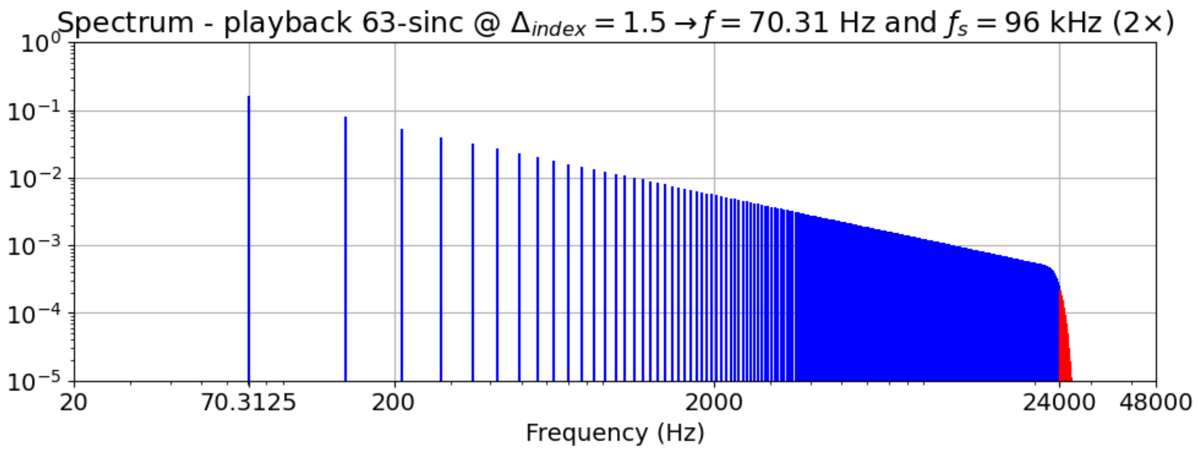
Anti-alias filter



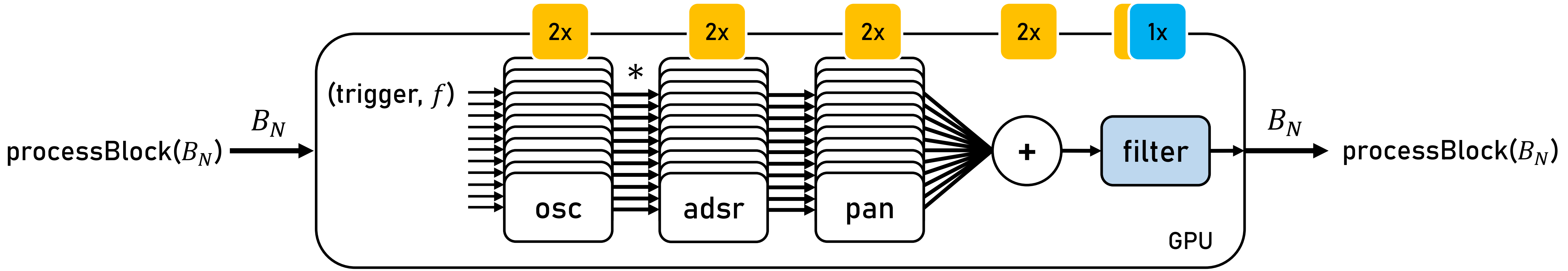


Anti-alias filter





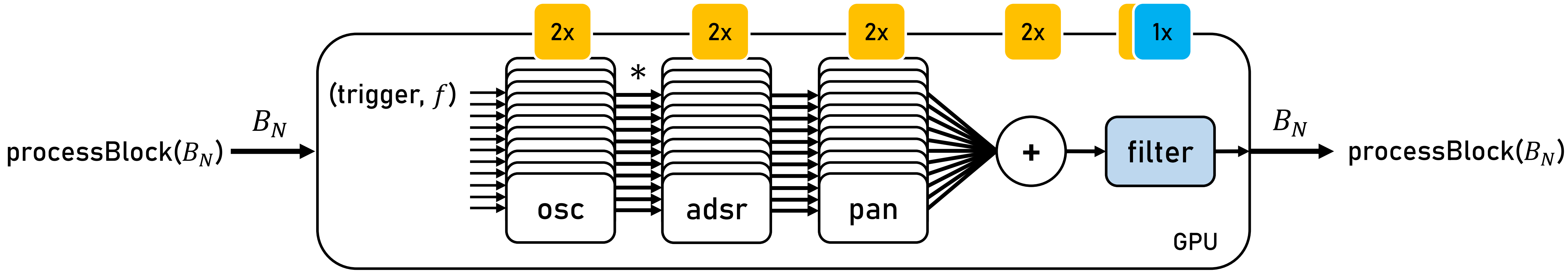
Anti-alias filter



Sinc filter

- “Ideal” low-pass filter
  - 🧱 → flat pass-band
  - Infinite IR, impractical

Anti-alias filter

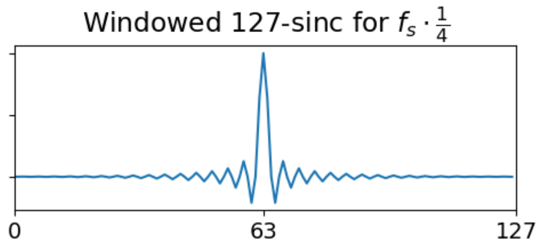


Sinc filter

- “Ideal” low-pass filter
  - 🧱 → flat pass-band
  - Infinite IR, impractical

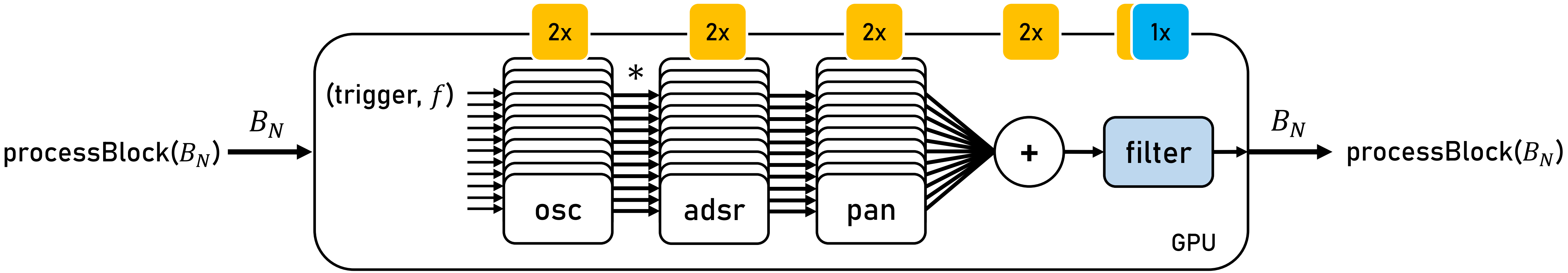
Windowed sinc filter (approximate)

- Convolve IR with audio signal



1. Frequency domain
   - FFT $\rightarrow * \rightarrow$ IFFT
   - Scattered data dependencies
   - Not trivial (e.g. cuFFTdx)

Anti-alias filter

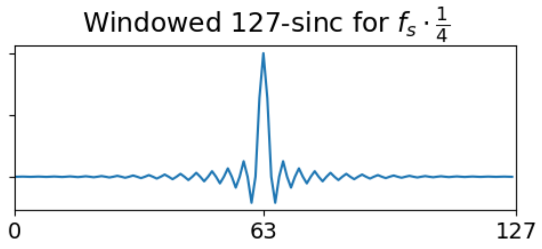


Sinc filter

- "Ideal" low-pass filter
  - 🧱 → flat pass-band
  - Infinite IR, impractical

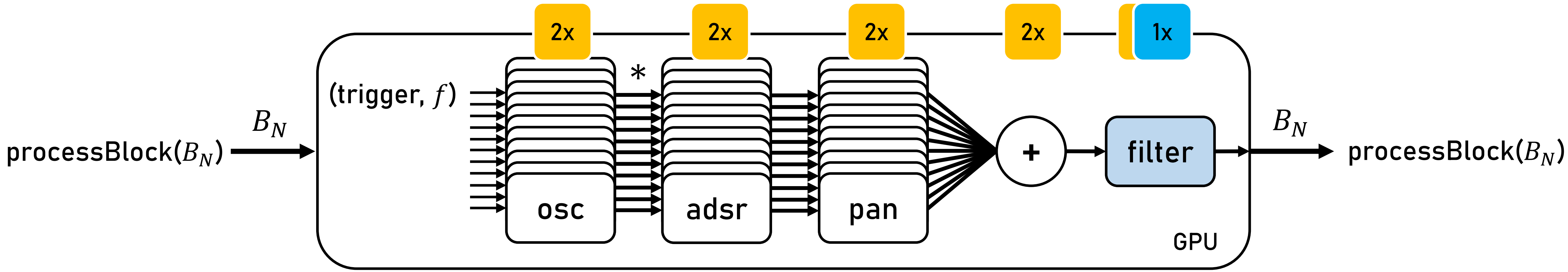
Windowed sinc filter (approximate)

- Convolve IR with audio signal

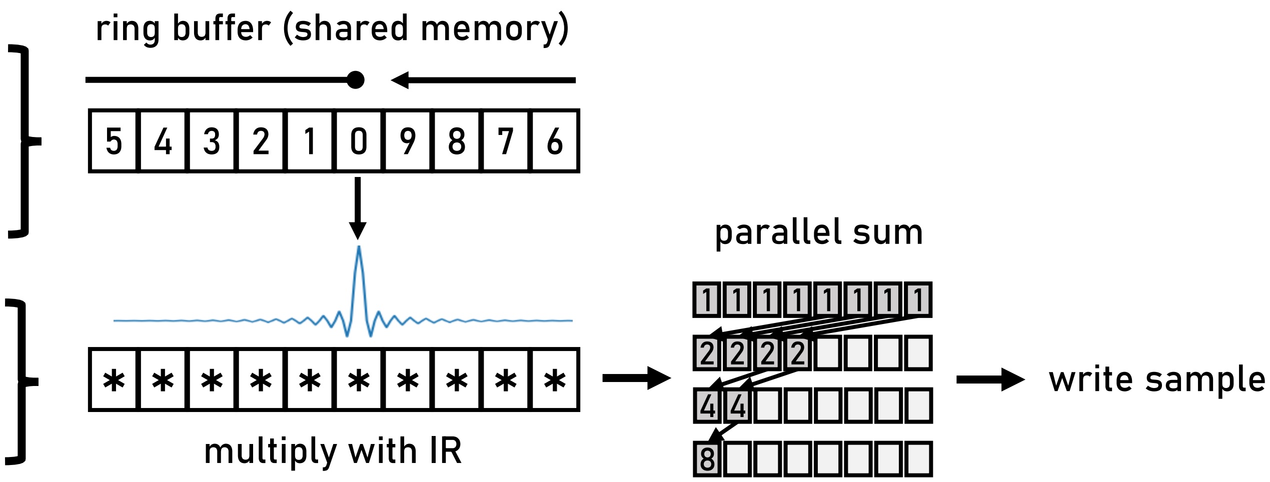


1. Frequency domain
   - FFT → * → IFFT
   - Scattered data dependencies
   - Not trivial (e.g. cuFFTdx)
2. Time domain 🤐
   - More operations…
   - Contiguous data dependencies
   - Easy (parallel reduction)

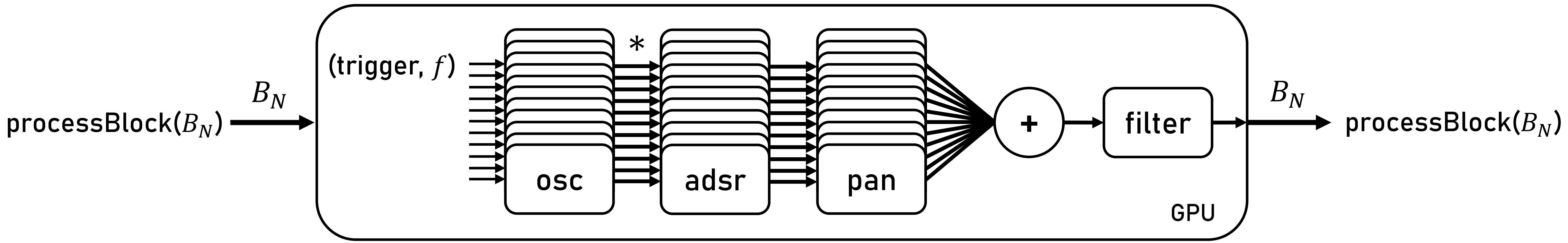
Anti-alias filter




```
__global__ void kernel(control_t* g_in, output_t* g_out) {
  __static__ convolver_t s_convolver;
  for(int i = 0; i < N; ++i) {
    …
    if (threadIdx.x == 0) {
      // sample1,2 contain summed signal for thread 0
      s_convolver.push(sample1.x, sample1.y);
      s_convolver.push(sample2.x, sample2.y);
    }
    __syncthreads();
    float ir = c_sincIR[threadIdx.x];
    float2 sample = blockSum(s_convolver.pull(threadIdx.x)*ir);
    if (threadIdx.x == 0) {    // 1 global write thread
      g_out.sample[0] = sample;
    }
  }
}
```

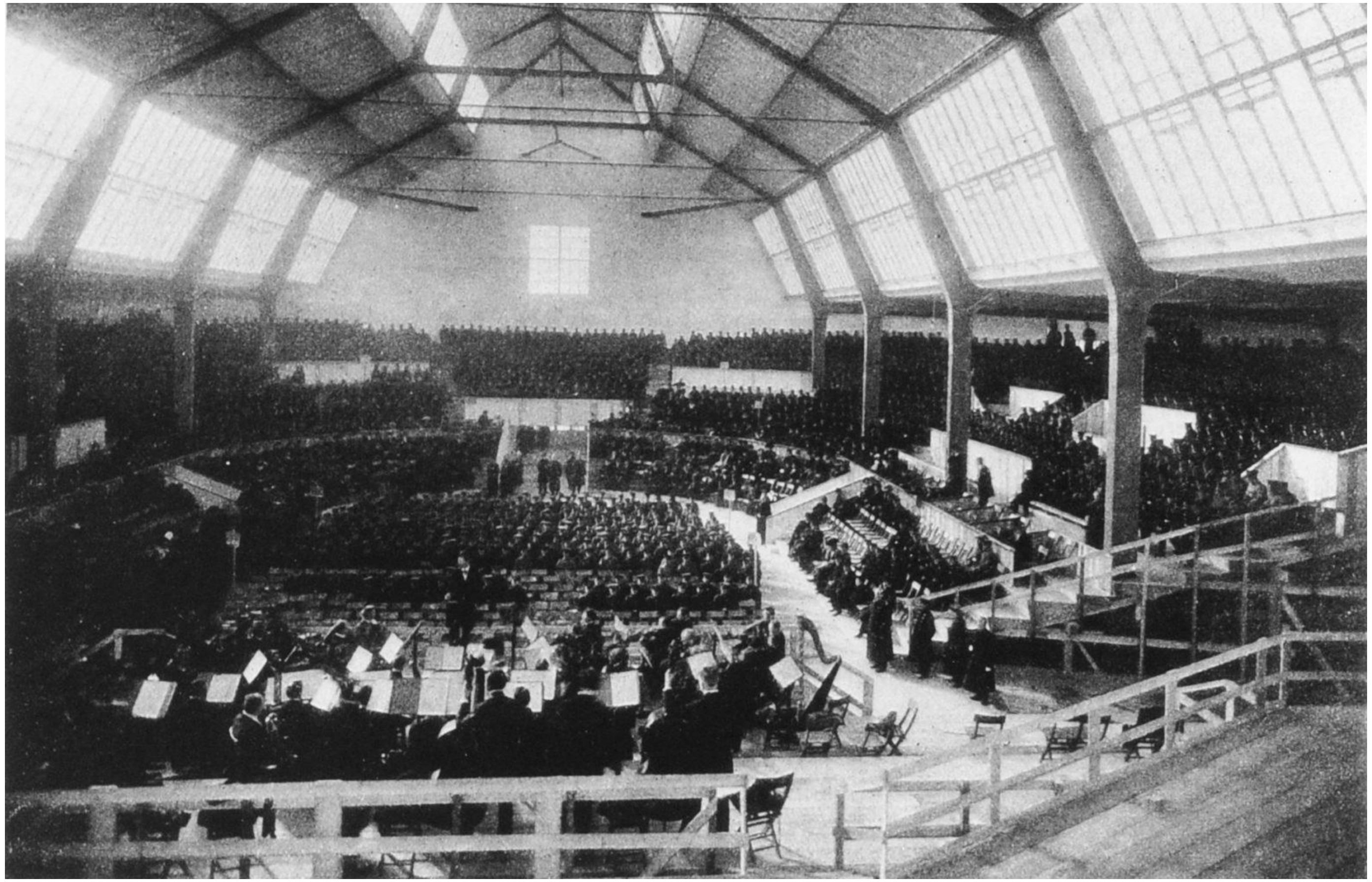


Done?




```
void processBlock(buffer) {
  short_kernel<<<…>>>(…); // launch & wait
}
```

Why so many oscillators?



Mahler's Symphony No. 8
"Symphony of a Thousand"

A symphonic universe.

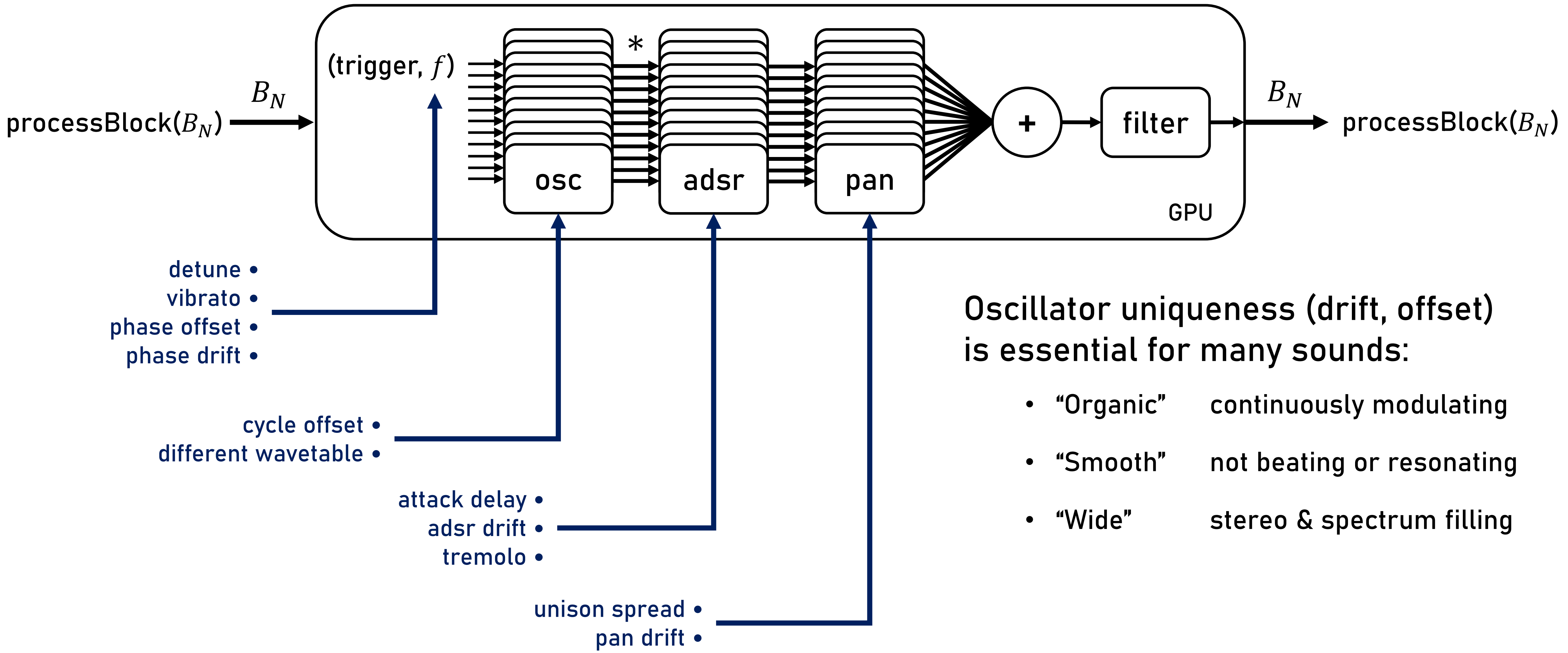
1910 premiere:

858 singers
171 instrumentalists

= 1029 sounds

CASCADE GPU SYNTH

Randomize



Oscillator uniqueness (drift, offset) is essential for many sounds:

- "Organic" continuously modulating
- "Smooth" not beating or resonating
- "Wide" stereo & spectrum filling

Future

- More variation! More experimentation!
- Different synthesis algorithms?
- Emulate delay, reverb with oscillators?
- Cross-hardware compatibility
- GPU-based hardware products

- Suggestions very welcome.

Thank you!

Want to talk more?

cecill@koaladsp.com

ADC2024 Discord: @ijsf