

Deep dive into Mamba

An overview by
Ushakov V., Romanov M.

S4/S4D/S5*

*friendly reminder

State Space Models (SSM)

State → Output

$$\dot{x}(t) = \mathbf{A}x(t) + \mathbf{B}u(t)$$

$$y(t) = \mathbf{C}x(t) + \mathbf{D}u(t)$$

Parameters

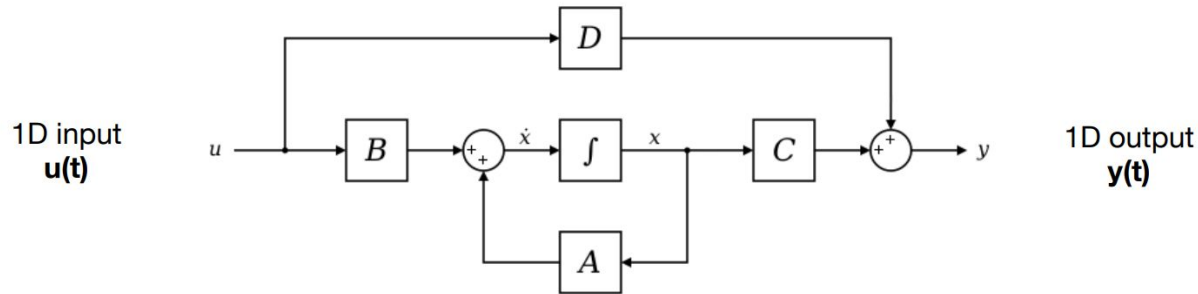
$$\mathbf{A} \in \mathbb{R}^{N \times N}$$

$$\mathbf{B} \in \mathbb{R}^{N \times 1}$$

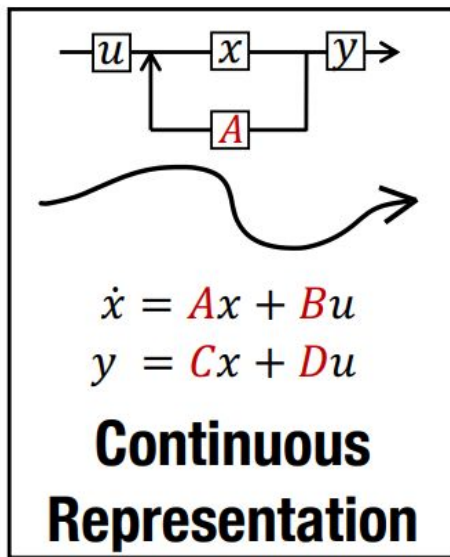
$$\mathbf{C} \in \mathbb{R}^{1 \times N}$$

$$\mathbf{D} \in \mathbb{R}^{1 \times 1}$$

Function-to-function map $u(t) \mapsto y(t)$

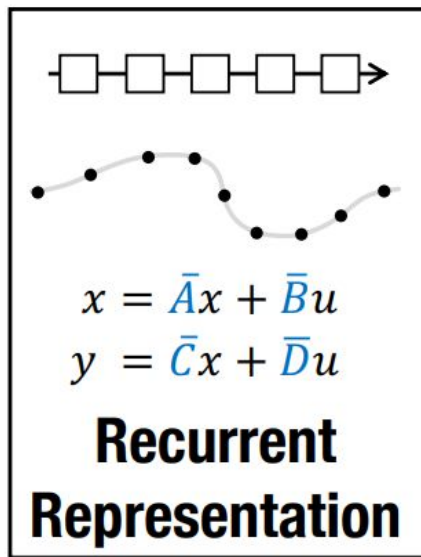


Properties of SSMs



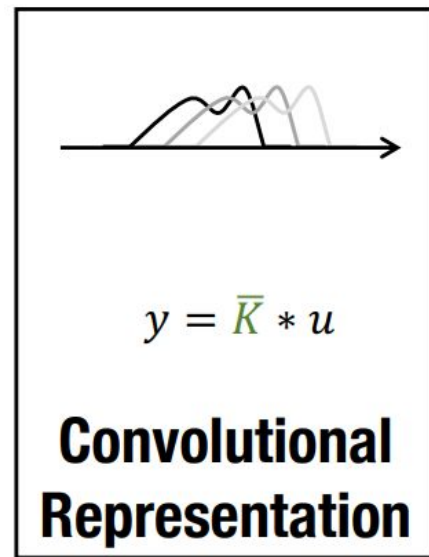
Handle continuous time series

Discretize
→



Efficient autoregressive inference

Unroll
→



Efficient training, scalability

SSM: Convolution View

$$x_k = \overline{A}x_{k-1} + \overline{B}u_k$$

$$y_k = \overline{C}x_k$$

Can explicitly unroll the linear recurrence in closed form

$$\begin{array}{llll} x_0 = \overline{B}u_0 & x_1 = \overline{A}\overline{B}u_0 + \overline{B}u_1 & x_2 = \overline{A}^2\overline{B}u_0 + \overline{A}\overline{B}u_1 + \overline{B}u_2 & \dots \\ y_0 = \overline{C}\overline{B}u_0 & y_1 = \overline{C}\overline{A}\overline{B}u_0 + \overline{C}\overline{B}u_1 & y_2 = \overline{C}\overline{A}^2\overline{B}u_0 + \overline{C}\overline{A}\overline{B}u_1 + \overline{C}\overline{B}u_2 & \dots \end{array}$$

SSM: Convolution View

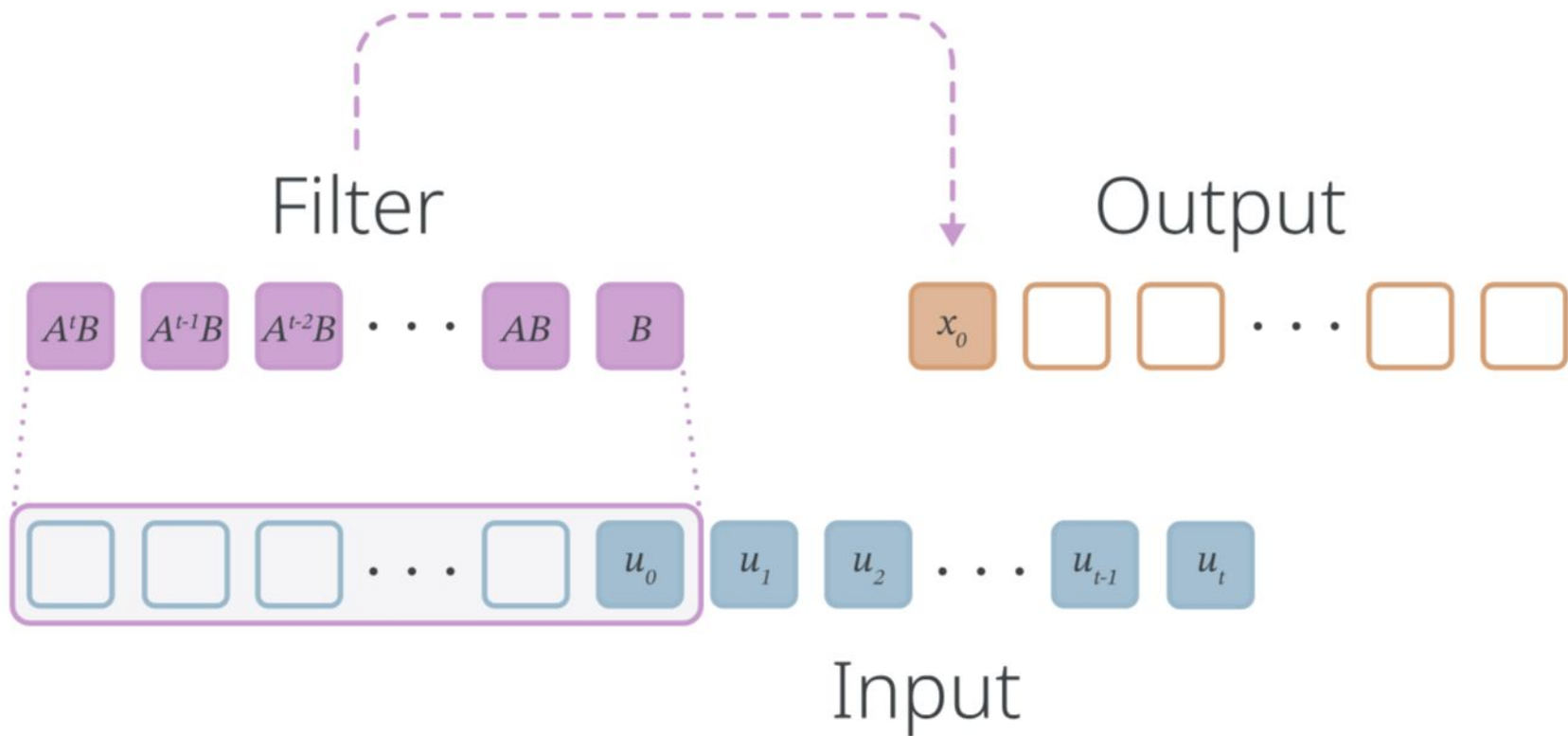
$$y_k = \overline{CA}^k \overline{B} u_0 + \overline{CA}^{k-1} \overline{B} u_1 + \cdots + \overline{CAB} u_{k-1} + \overline{CB} u_k$$

$$\overline{K} \in \mathbb{R}^L := (\overline{CB}, \overline{CAB}, \dots, \overline{CA}^{L-1} \overline{B})$$

$$y = \overline{K} * u$$

Can be computed with **convolutions**, similar to CNNs

SSM: Convolution View



Structured Matrices

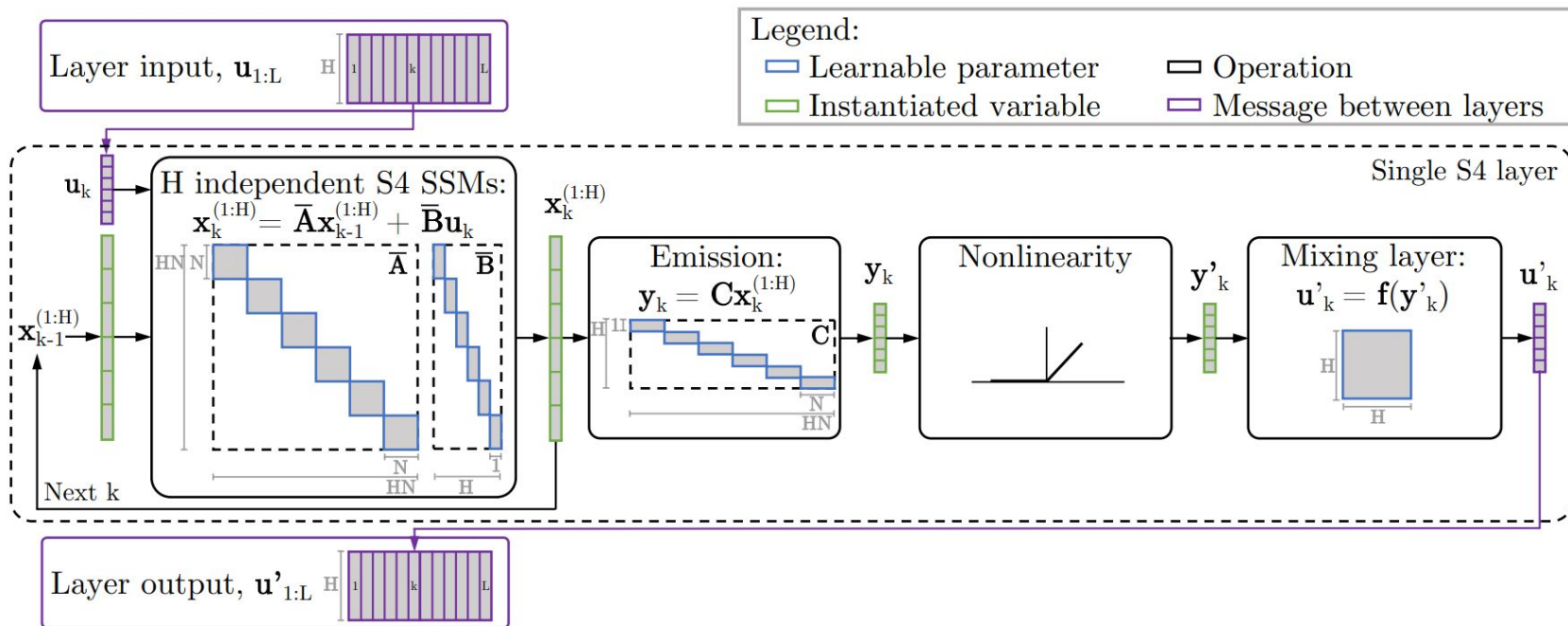
Algorithm 1 S4 CONVOLUTION KERNEL (SKETCH)

Input: S4 parameters $\mathbf{\Lambda}, \mathbf{P}, \mathbf{Q}, \mathbf{B}, \mathbf{C} \in \mathbb{C}^N$ and step size Δ

Output: SSM convolution kernel $\overline{\mathbf{K}} = \mathcal{K}_L(\overline{\mathbf{A}}, \overline{\mathbf{B}}, \overline{\mathbf{C}})$ for $\mathbf{A} = \mathbf{\Lambda} - \mathbf{P}\mathbf{Q}^*$ (equation (5))

- 1: $\tilde{\mathbf{C}} \leftarrow (\mathbf{I} - \overline{\mathbf{A}}^L)^* \overline{\mathbf{C}}$ ▷ Truncate SSM generating function (SSMGF) to length L
 - 2: $\begin{bmatrix} k_{00}(\omega) & k_{01}(\omega) \\ k_{10}(\omega) & k_{11}(\omega) \end{bmatrix} \leftarrow [\tilde{\mathbf{C}} \mathbf{Q}]^* \left(\frac{2}{\Delta} \frac{1-\omega}{1+\omega} - \mathbf{\Lambda} \right)^{-1} [\mathbf{B} \mathbf{P}]$ ▷ Black-box Cauchy kernel
 - 3: $\hat{\mathbf{K}}(\omega) \leftarrow \frac{2}{1+\omega} [k_{00}(\omega) - k_{01}(\omega)(1 + k_{11}(\omega))^{-1}k_{10}(\omega)]$ ▷ Woodbury Identity
 - 4: $\hat{\mathbf{K}} = \{\hat{\mathbf{K}}(\omega) : \omega = \exp(2\pi i \frac{k}{L})\}$ ▷ Evaluate SSMGF at all roots of unity $\omega \in \Omega_L$
 - 5: $\overline{\mathbf{K}} \leftarrow \text{iFFT}(\hat{\mathbf{K}})$ ▷ Inverse Fourier Transform
-

S4: single-input, single-output (SISO)



S4D

$$CA^\ell B = \begin{bmatrix} c_1 & c_2 & c_3 & \cdots & c_N \end{bmatrix} \times \begin{bmatrix} a_1^\ell & 0 & 0 & \cdots & 0 \\ 0 & a_2^\ell & 0 & \cdots & 0 \\ 0 & 0 & a_3^\ell & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_N^\ell \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_N \end{bmatrix}$$

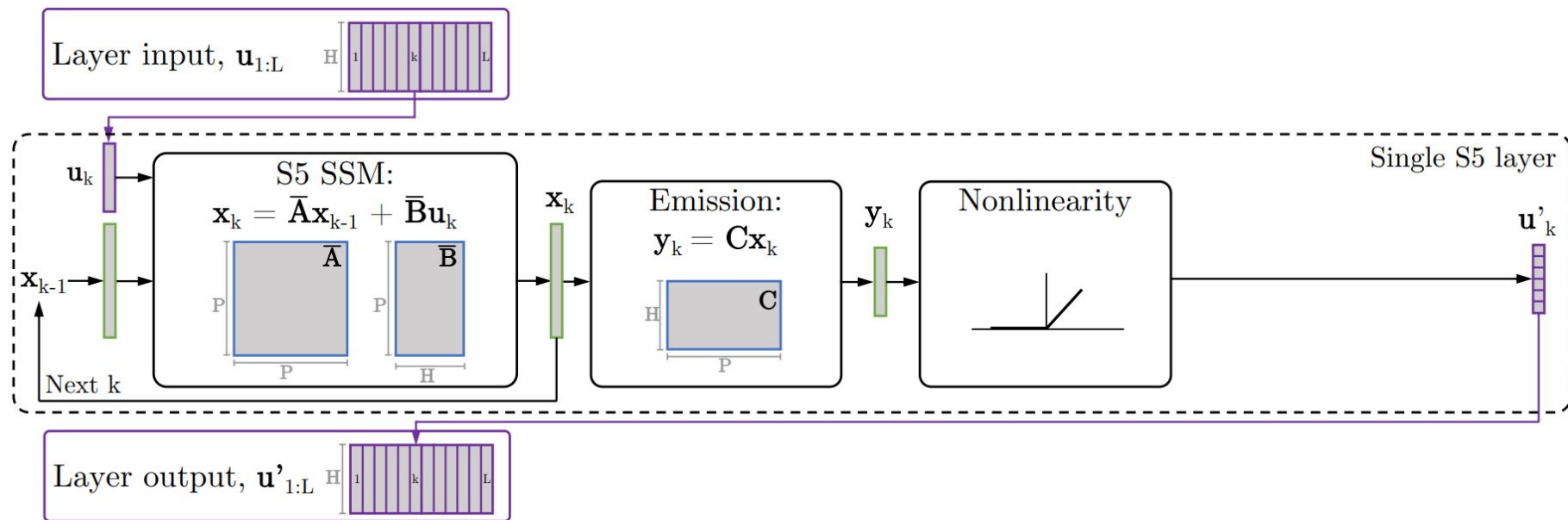
$$CA^\ell B = \begin{bmatrix} c_1 a_1^\ell & c_2 a_2^\ell & c_3 a_3^\ell & \cdots & c_N a_N^\ell \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_N \end{bmatrix}$$

$$CA^\ell B = \sum_{i=0}^N c_i a_i^\ell b_i$$

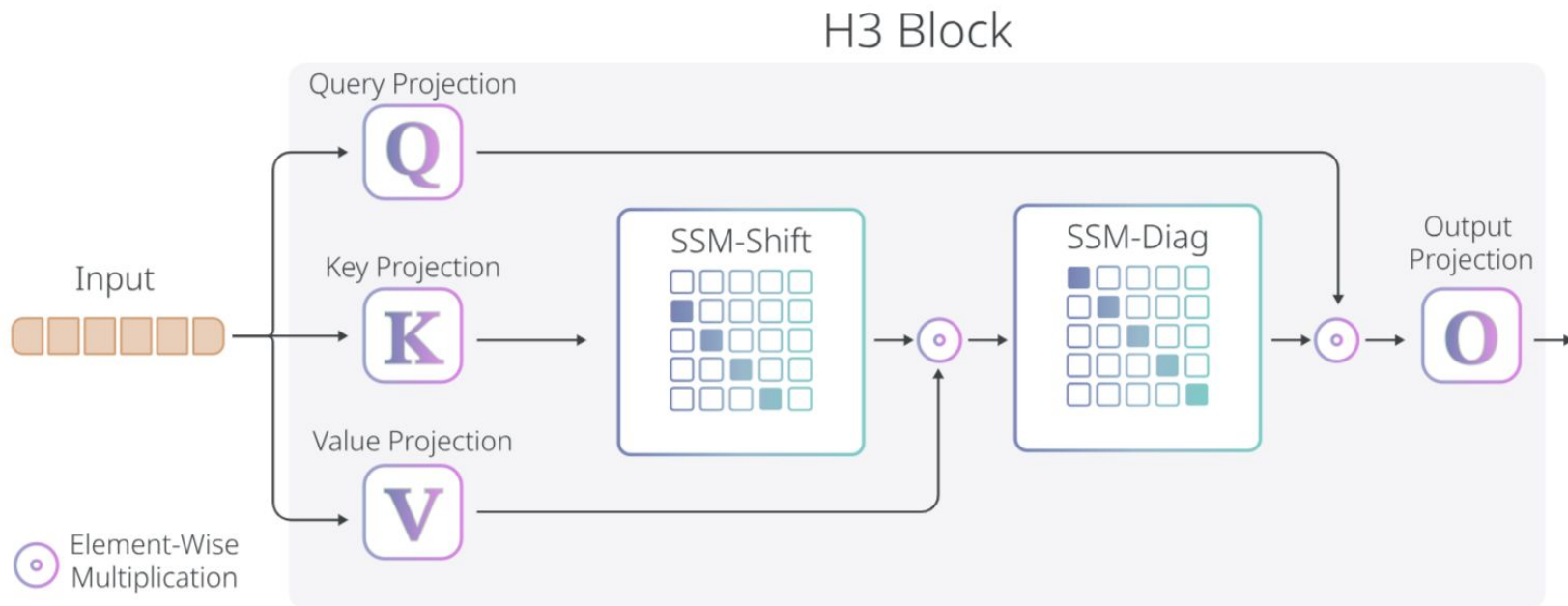
S4D

$$K = \begin{bmatrix} c_1 b_1 & c_2 b_2 & c_3 b_3 & \cdots & c_N b_N \end{bmatrix} \times \begin{bmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^\ell \\ 1 & a_2 & a_2^2 & \cdots & a_2^\ell \\ 1 & a_3 & a_3^2 & \cdots & a_3^\ell \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_N & a_N^2 & \cdots & a_N^\ell \end{bmatrix}$$

S5 (MIMO)



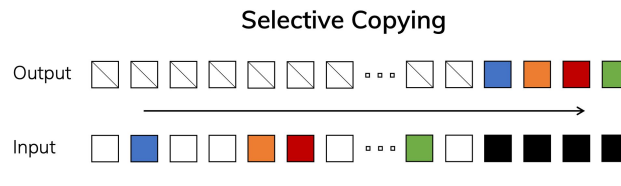
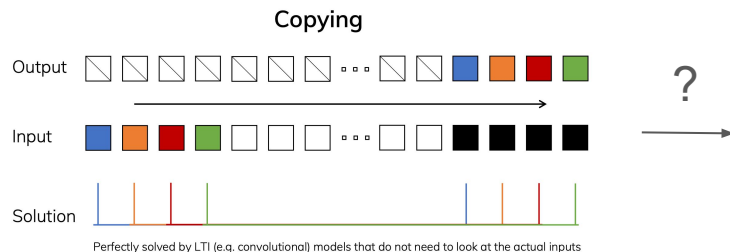
Bonus: H3 (Hungry Hungry Hippos)



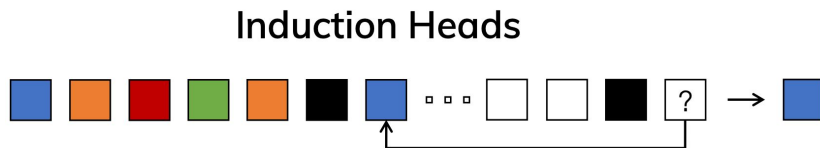
Mamba

Why S6?

- Selective copying



- Induction heads



Why S6?

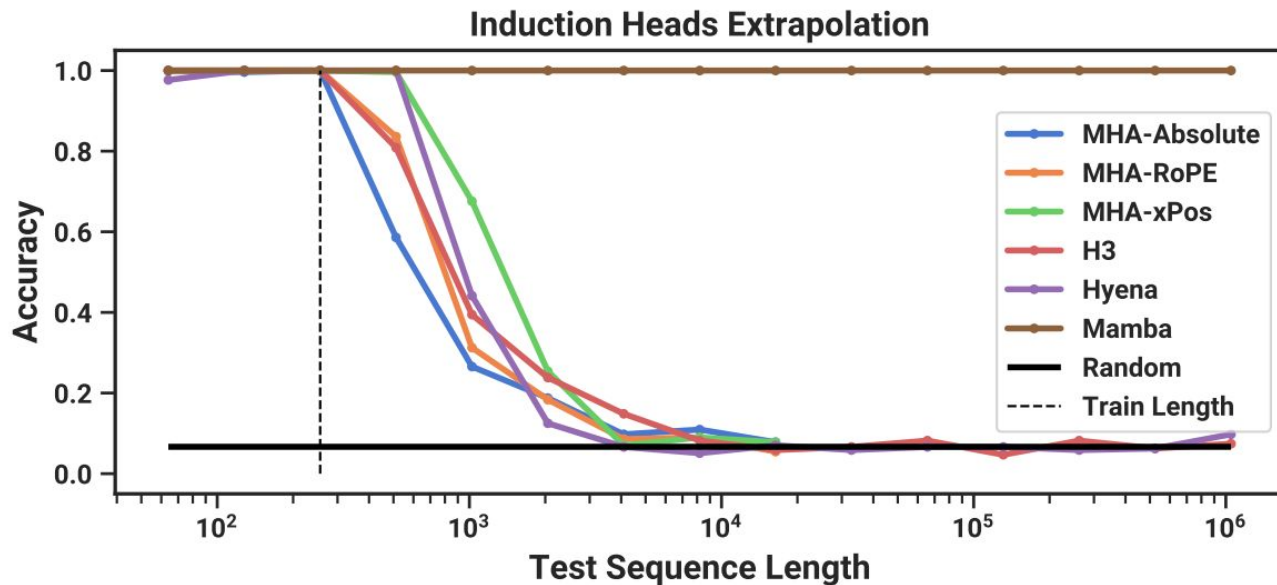


Table 2: (**Induction Heads.**) Models are trained on sequence length $2^8 = 256$, and tested on increasing sequence lengths of $2^6 = 64$ up to $2^{20} = 1048576$. Full numbers in Table 11.

Algorithm 1 SSM (S4)

Output: $y : (B, L, D)$

1: $A : (D, N) \leftarrow \text{Parameter}$

- Represents structured $N \times N$ matrix

2: $\mathbf{B} : (\mathbf{D}, \mathbf{N}) \leftarrow \text{Parameter}$ 3: $C : (D, N) \leftarrow \text{Parameter}$ 4: $\Delta : (D) \leftarrow \tau_{\Lambda}(\text{Parameter})$ 5: $\overline{A}, \overline{B} : (D, N) \leftarrow \text{discretize}(\Delta, A, B)$
$$6: y \leftarrow \text{SSM}(\overline{A}, \overline{B}, C)(x)$$

- ▷ Time-invariant: recurrence or convolution

7: **return** y

Input: $x : (B, L, D)$

Output: $y : (B, L, D)$

1: $A : (D, N) \leftarrow \text{Parameter}$

- Represents structured $N \times N$ matrix

2: $\mathbf{B} : (\mathbf{B}, \mathbf{L}, \mathbf{N}) \leftarrow s_B(x)$ 3: $C : (B, L, N) \leftarrow s_C(x)$
$$4: \Delta : (B, L, D) \leftarrow \tau_{\Delta}(\text{Parameter} + s_{\Delta}(x))$$
5: $\bar{A}, \bar{B} : (B, L, D, N) \leftarrow \text{discretize}(\Delta, A, B)$
$$6: y \leftarrow \text{SSM}(\overline{A}, \overline{B}, C)(x)$$

- ▷ Time-varying: recurrence (*scan*) only

7: **return** y

Formal

$$h_t = \bar{A}_t h_{t-1} + \bar{B}_t x_t$$

$$y_t = C_t h_t$$

$$\bar{A}_t = \exp(\Delta_t A)$$

$$\bar{B}_t = (\Delta_t A)^{-1} \cdot (\exp(\Delta_t A) - I) \Delta_t B_t$$

$$B_t = s_B(x_t) = \textit{Linear}_N(x_t)$$

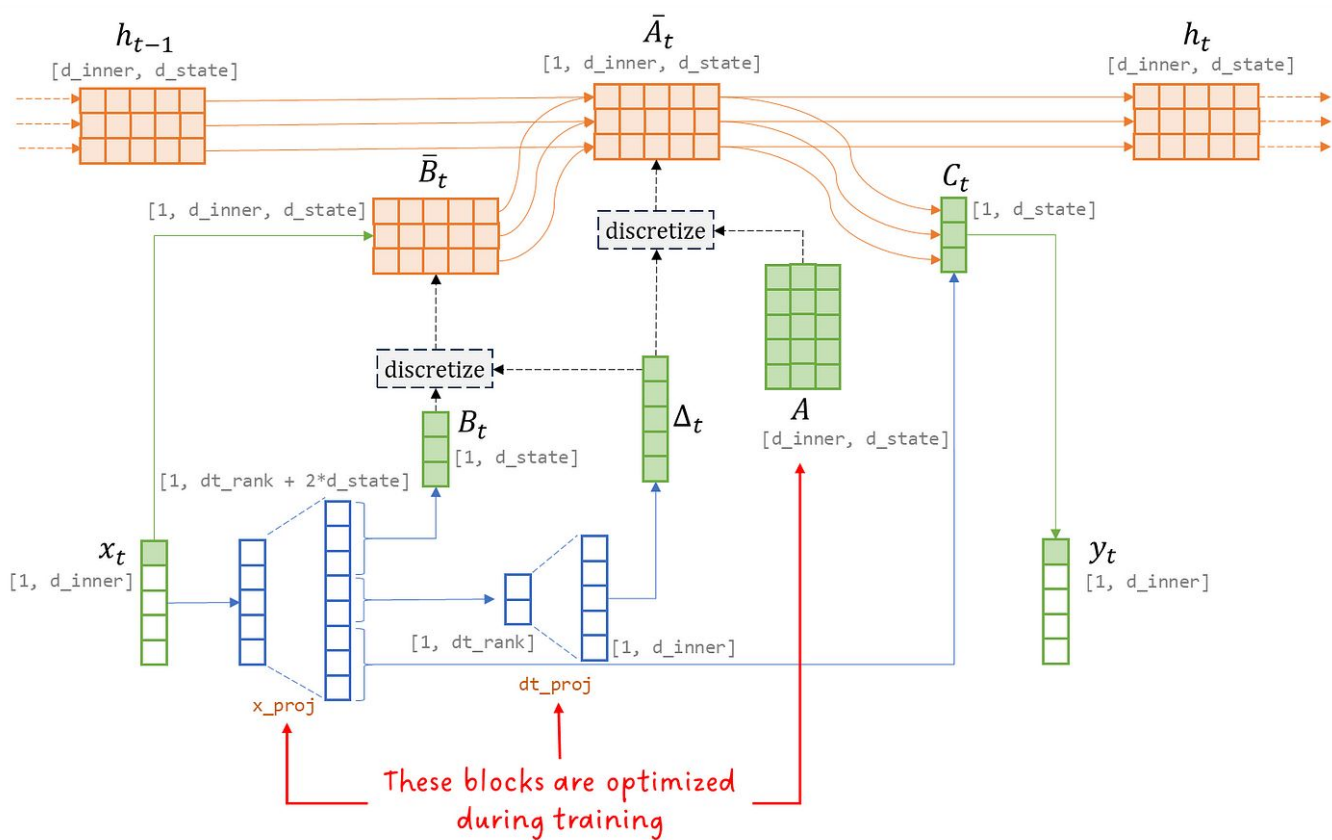
$$C_t = s_C(x_t) = \textit{Linear}_N(x_t)$$

$$\Delta_t = \text{softplus}(\text{Parameter} + s_\Delta(x_t))$$

$$s_\Delta(x_t) = \textit{Linear}_D(\textit{Linear}_R(x_t))$$

$$s_\Delta(x_t) = \textit{Broadcast}_D(\textit{Linear}_1(x_t))$$

Selective SSM (S6)



Discretize A

$$A \in R^{D \times N}$$

$$\Delta_t \in R^{1 \times D}$$

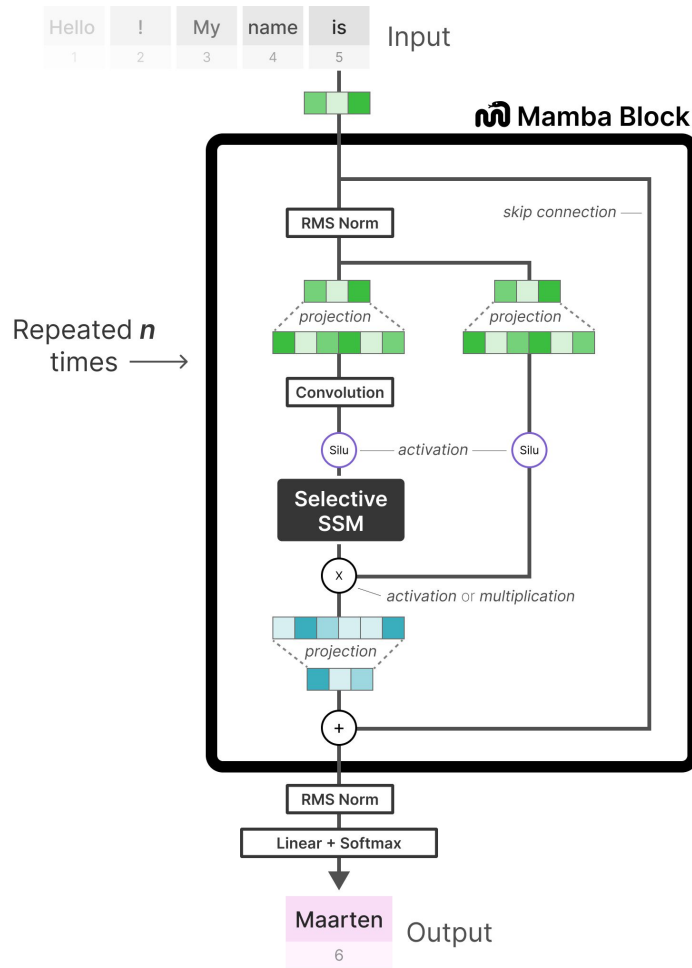
$$\bar{A}_t = \exp(\Delta_t A) = \exp(\Delta_t^{brd} \odot A^{brd}) \in R^{1 \times D \times N}$$

$$\Delta_t^{brd} = \text{reshape}(\Delta_t, [1, D, 1])$$

$$A^{brd} = \text{reshape}(A, [1, D, N])$$

Mamba architecture

- Upsample X
- Convolution to mix last(4) upsampled X
- S6 block
- Skip-connection
- Downsample



A deeper dive into Mamba

План

1. Введение в GPU-вычисления:

- Архитектура GPU
- Модель массового параллелизма

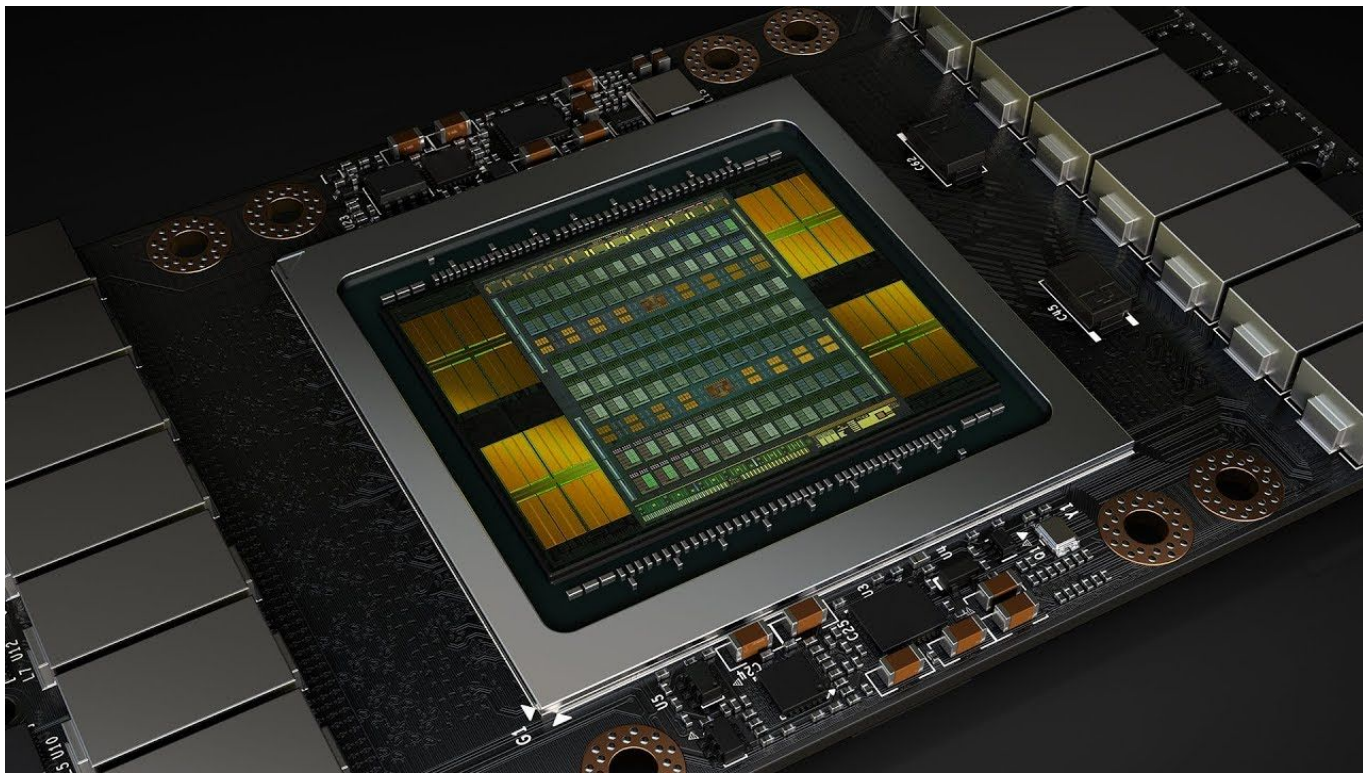
2. Разбор задачи Associative Scan

- Постановка задачи
- Sequential Scan
- Parallel Scan (Hillis-Steele, 1986)
- Work-Efficient Parallel Scan (Blelloch, 1990)

3. Связь Associative Scan с Mamba

Введение в GPU-вычисления

Архитектура GPU



Архитектура GPU



Архитектура GPU

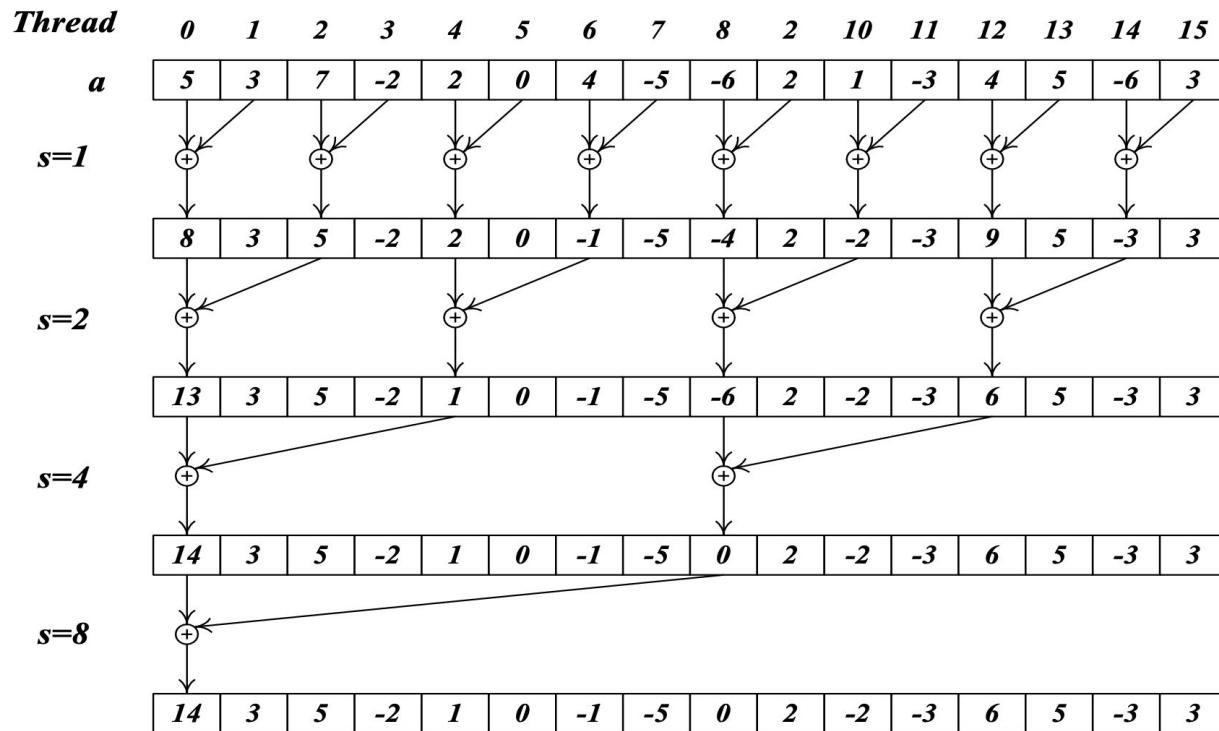
- CUDA-ядра внутри одного мультимикропроцессора работают как SIMD блоками по 32 ядра (warp)
- Один Instruction Pointer на все потоки одного warp
- Есть глобальная память, общая для всего устройства
- Есть разделяемая память, общая для всех CUDA-ядер в одном SM



Модель массового параллелизма

- Большое количество **потоков** (*thread*) параллельно выполняют одну и ту же функцию – **ядро** (*kernel*)
- Потоки группируются в **блоки** (*thread blocks*), которые исполняются на отдельных SM
- Блоки объединяются в **решетку/сетку** блоков (*grid*)
- Для взаимодействия потоков используется разделяемая память с барьерной синхронизацией
- Доступны атомарные операции для разделяемой и глобальной памяти

Пример: параллельная редукция



Пример: параллельная редукция - реализация

 reduce.cu

```
1  __global__ void reduce(int *inData, int *outData)
2  {
3      __shared__ int data[BLOCK_SIZE];
4      int tid = threadIdx.x;
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6      data[tid] = inData[i]; // load into shared memory
7      __syncthreads();
8
9      for (int s = 1; s < blockDim.x; s *= 2) {
10         if (tid % (2 * s) == 0)
11             data[tid] += data[tid + s];
12         __syncthreads();
13     }
14
15     if (tid == 0) // write result of block reduction
16         outData[blockIdx.x] = data[0];
17 }
18
```

Associative Scan

Постановка задачи

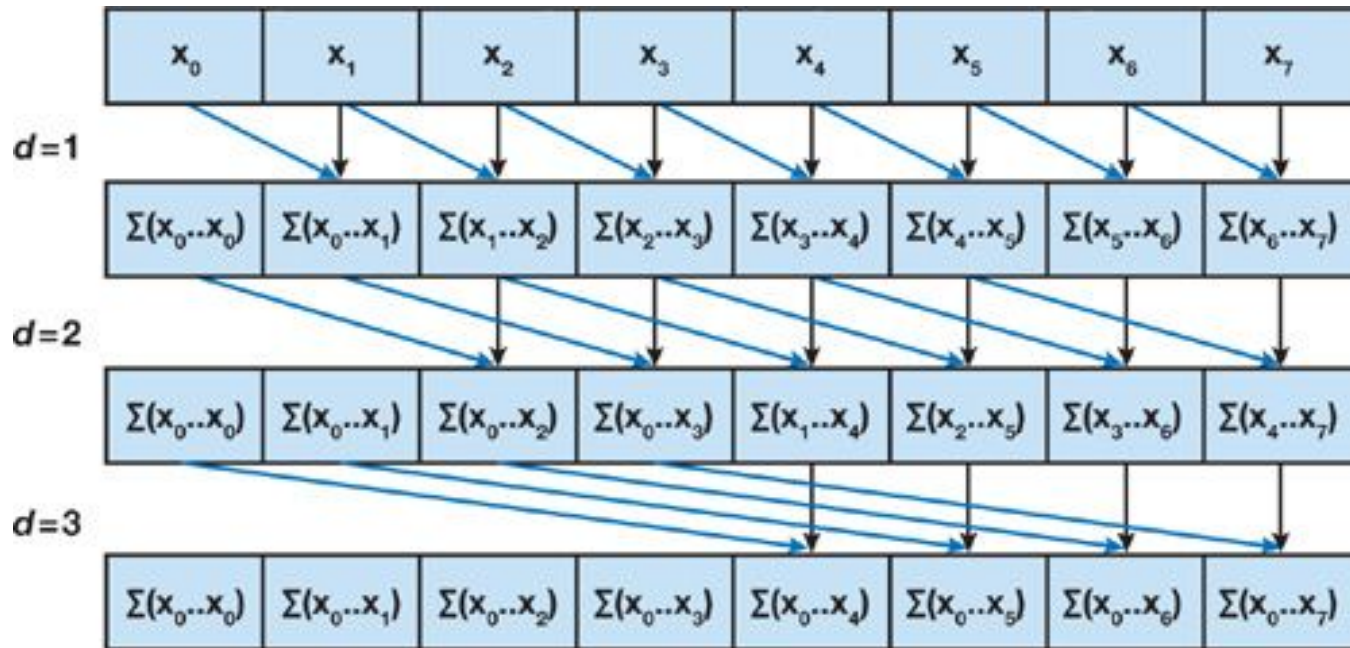
The associative scan operation takes a binary associative operator $\oplus$ with identity I , and an array of n elements

$$[a_0, a_1, \dots, a_{n-1}],$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})].$$

Parallel Scan (Hillis-Steele, 1986)

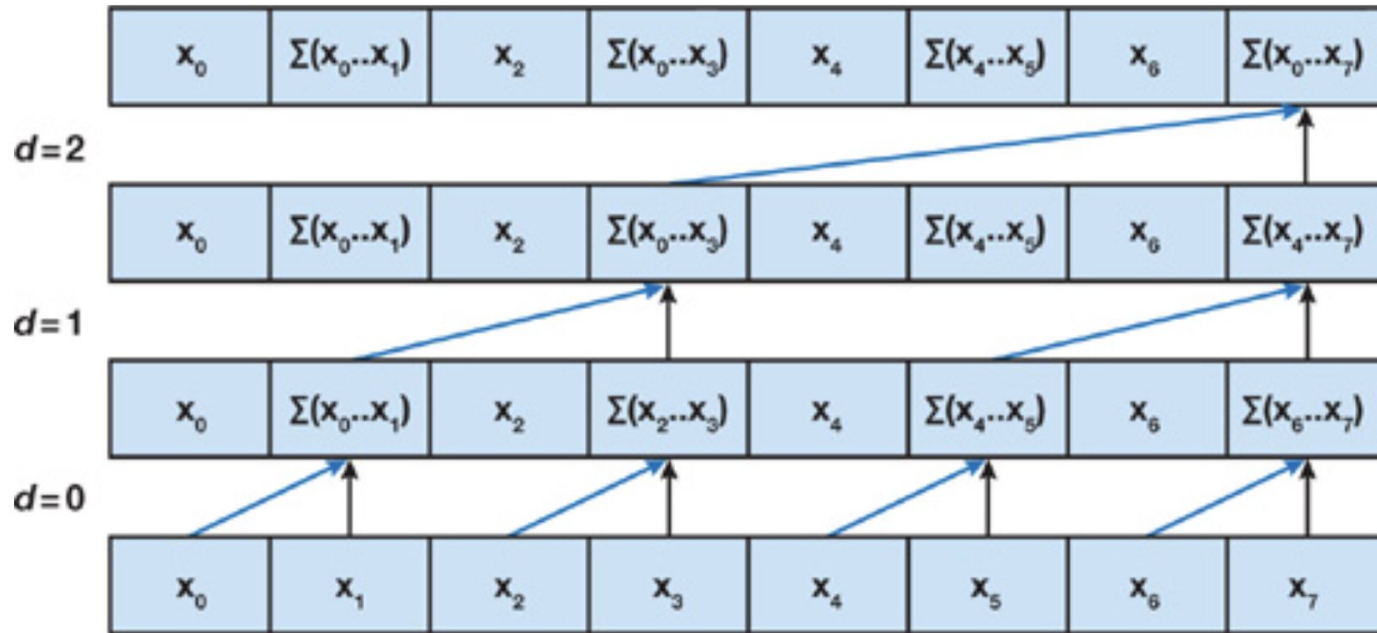


Parallel Scan (Hillis-Steele, 1986) - реализация

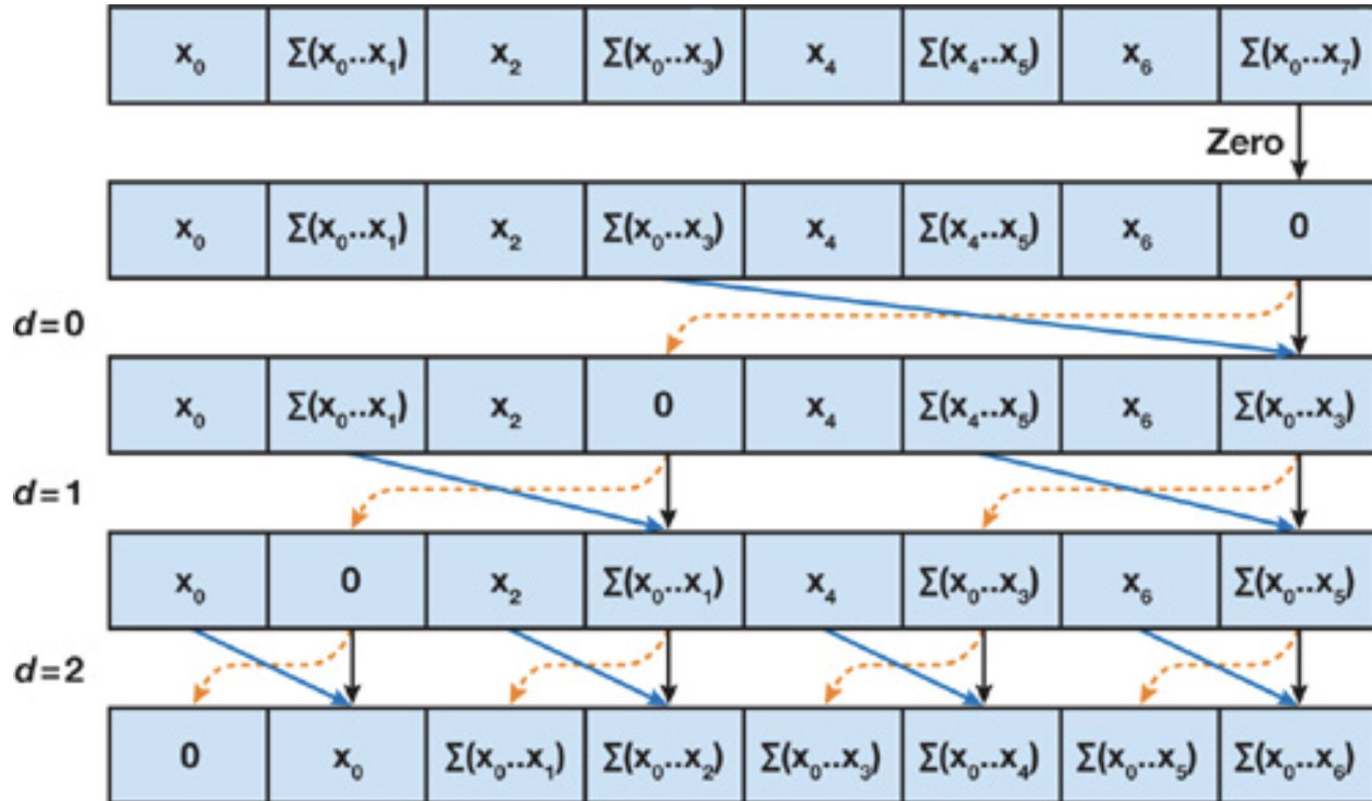
naive_scan.cu

```
1  __global__ void naive_scan(float *g_odata, float *g_idata, int n)
2  {
3      extern __shared__ float temp[];
4      int thid = threadIdx.x;
5      int pout = 0, pin = 1;
6      // Load input into shared memory.
7      temp[pout * n + thid] = (thid > 0) ? g_idata[thid - 1] : 0;
8      __syncthreads();
9      for (int offset = 1; offset < n; offset *= 2)
10     {
11         pout = 1 - pout; // swap buffer indices
12         pin = 1 - pout;
13         if (thid >= offset)
14             temp[pout * n + thid] = temp[pin * n + thid] + temp[pin * n + thid - offset];
15         else
16             temp[pout * n + thid] = temp[pin * n + thid];
17         __syncthreads();
18     }
19     g_odata[thid] = temp[pout * n + thid]; // write output
20 }
```

Work-Efficient Parallel Scan (Blelloch, 1990) - Up-Sweep



Work-Efficient Parallel Scan (Blelloch, 1990) - Down-Sweep



Work-Efficient Parallel Scan (Blelloch, 1990) - реализация

 work_efficient_scan.cu

```
1  __global__ void work_efficient_scan(float *g_odata, float *g_idata, int n)
2  {
3      extern __shared__ float temp[];
4      int thid = threadIdx.x;
5      int offset = 1;
6      temp[thid] = g_idata[thid]; // load input into shared memory
7
8      for (int d = n >> 1; d > 0; d >>= 1) // build sum in place up the tree
9      {
10         __syncthreads();
11         if (thid < d)
12         {
13             int ai = offset * (2 * thid + 1) - 1;
14             int bi = offset * (2 * thid + 2) - 1;
15             temp[bi] += temp[ai];
16         }
17         offset *= 2;
18     }
19 }
```

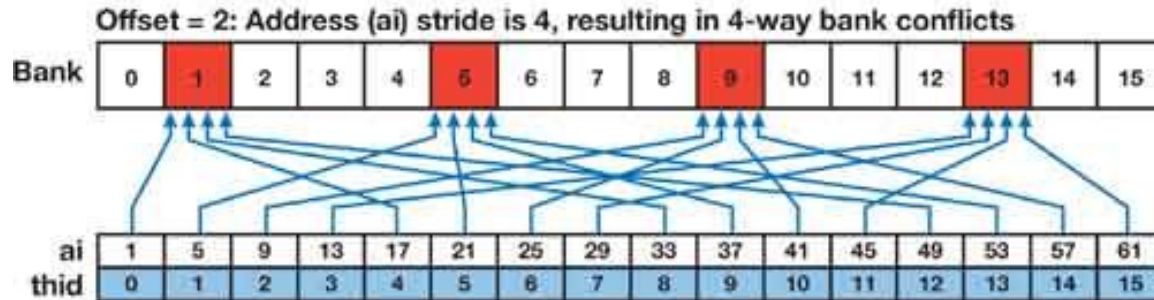
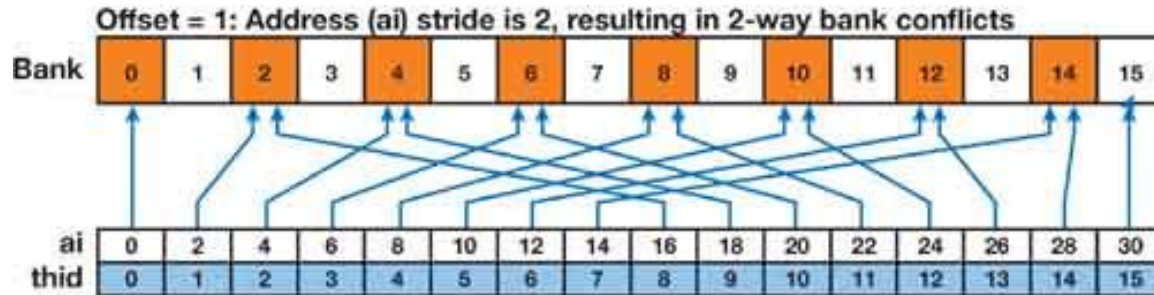
Work-Efficient Parallel Scan (Blelloch, 1990) - реализация

```
20     if (thid == 0)
21     {
22         temp[n - 1] = 0;
23     } // clear the last element
24
25     for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
26     {
27         offset >>= 1;
28         __syncthreads();
29         if (thid < d)
30         {
31             int ai = offset * (2 * thid + 1) - 1;
32             int bi = offset * (2 * thid + 2) - 1;
33             float t = temp[ai];
34             temp[ai] = temp[bi];
35             temp[bi] += t;
36         }
37     }
38     __syncthreads();
39
40     g_odata[thid] = temp[thid]; // write results to device memory
41 }
```

Fixing Bank Conflicts

```
int ai = offset*(2*thid+1)-1;  
int bi = offset*(2*thid+2)-1;  
temp[bi] += temp[ai]
```

Addressing Without Padding

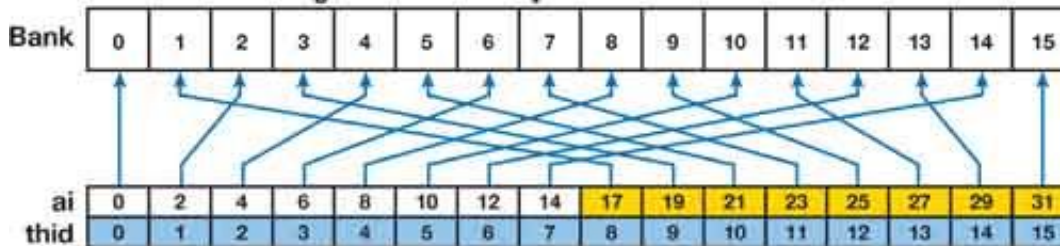


Fixing Bank Conflicts

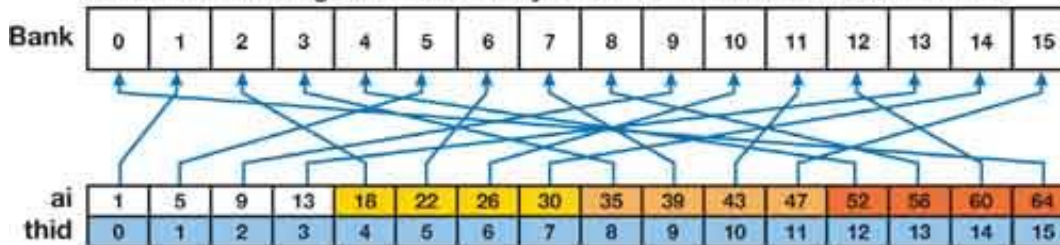
```
int ai = offset*(2*thid+1)-1;  
int bi = offset*(2*thid+2)-1;  
ai += ai / NUM_BANKS;  
bi += bi / NUM_BANKS;  
temp[bi] += temp[ai]
```

Addressing With Padding

Offset = 1: Padding addresses every 16 elements removes bank conflicts



Offset = 2: Padding addresses every 16 elements removes bank conflicts



Padding Increment: 0 1 2 3

Fixing Bank Conflicts - реализация

 work_efficient_scan_plus.cu

```
1  #define NUM_BANKS 16
2  #define LOG_NUM_BANKS 4
3  #define CONFLICT_FREE_OFFSET(n) ((n) >> NUM_BANKS + (n) >> (2 * LOG_NUM_BANKS))
4
5
6  __global__ void work_efficient_scan_plus(float *g_odata, float *g_idata, int n)
7  {
8      extern __shared__ float temp[]; // allocated on invocation
9      int thid = threadIdx.x;
10     int offset = 1;
11
12     int ai = thid;
13     int bi = thid + (n / 2);
14     int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
15     int bankOffsetB = CONFLICT_FREE_OFFSET(bi);
16     temp[ai + bankOffsetA] = g_idata[ai];
17     temp[bi + bankOffsetB] = g_idata[bi];
18
```

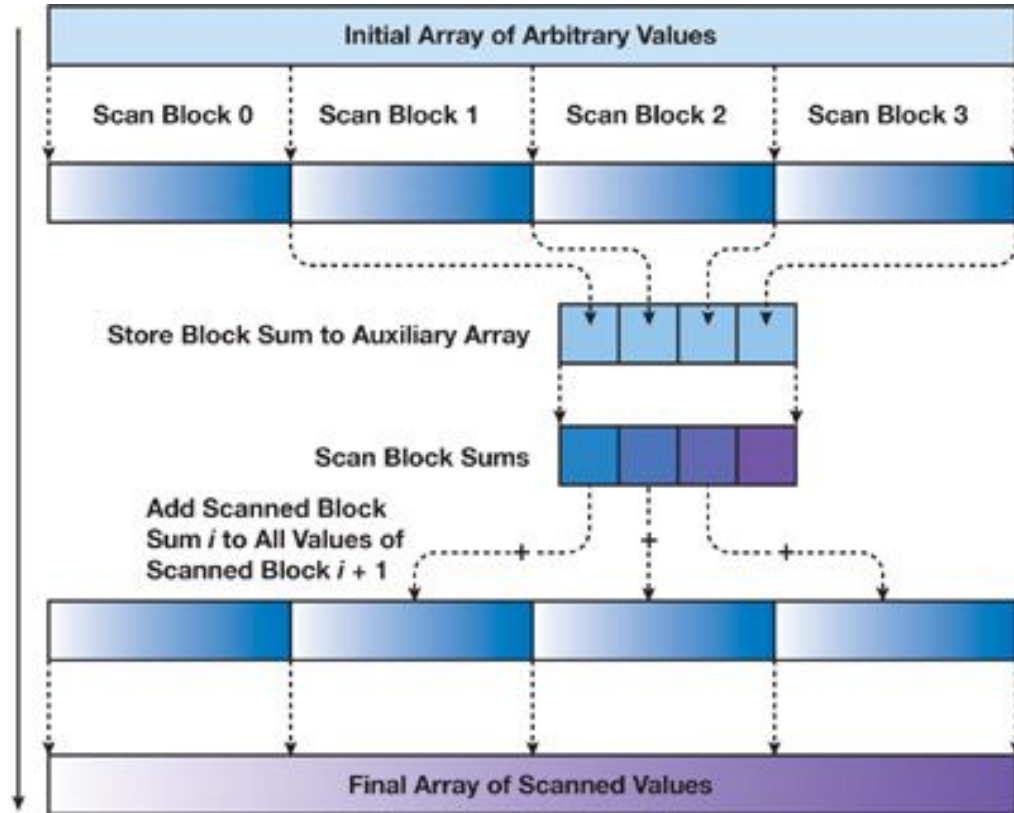
Fixing Bank Conflicts - реализация

```
19     for (int d = n >> 1; d > 0; d >= 1) // build sum in place up the tree
20     {
21         __syncthreads();
22         if (thid < d)
23         {
24             int ai = offset * (2 * thid + 1) - 1;
25             int bi = offset * (2 * thid + 2) - 1;
26             ai += CONFLICT_FREE_OFFSET(ai);
27             bi += CONFLICT_FREE_OFFSET(bi);
28
29             temp[bi] += temp[ai];
30         }
31         offset *= 2;
32     }
33
34     if (thid == 0)
35     {
36         temp[n - 1 + CONFLICT_FREE_OFFSET(n - 1)] = 0;
37     }
38
```

Fixing Bank Conflicts - реализация

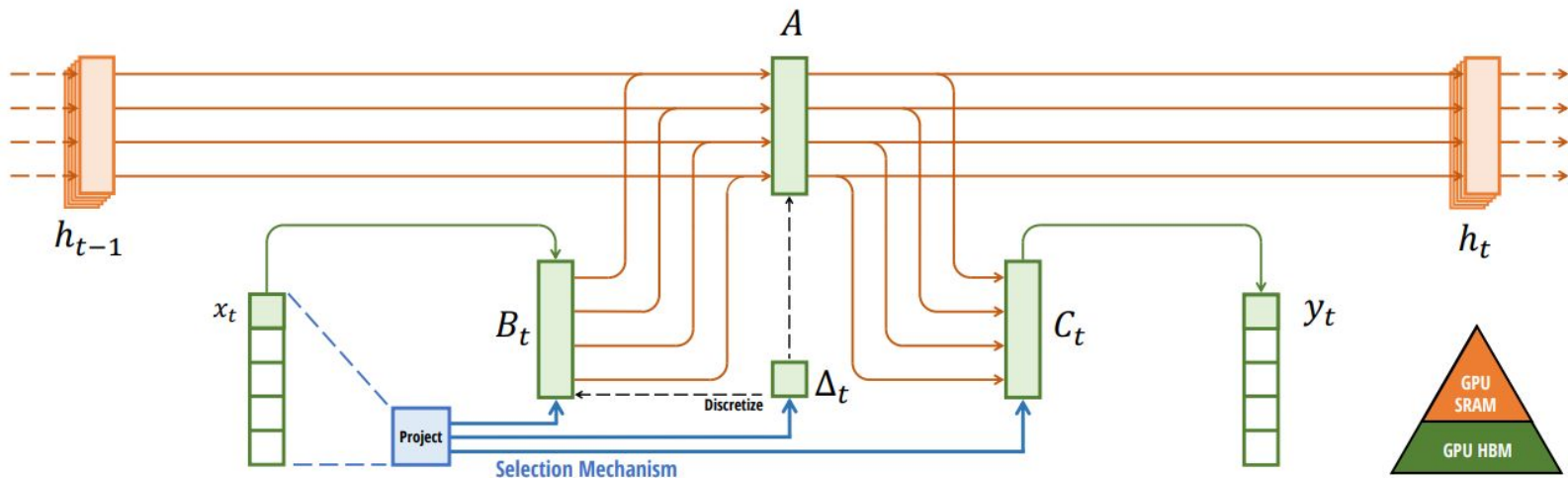
```
39     for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
40     {
41         offset >>= 1;
42         __syncthreads();
43         if (thid < d)
44         {
45             int ai = offset * (2 * thid + 1) - 1;
46             int bi = offset * (2 * thid + 2) - 1;
47             ai += CONFLICT_FREE_OFFSET(ai);
48             bi += CONFLICT_FREE_OFFSET(bi);
49
50             float t = temp[ai];
51             temp[ai] = temp[bi];
52             temp[bi] += t;
53         }
54     }
55     __syncthreads();
56
57     g_odata[ai] = temp[ai + bankOffsetA];
58     g_odata[bi] = temp[bi + bankOffsetB];
59 }
```

Arrays of Arbitrary Size



Associative Scan in Mamba

Selective State Space Model with Hardware-aware State Expansion



$$h'(t) = Ah(t) + Bx(t) \quad (1a)$$

$$y(t) = Ch(t) \quad (1b)$$

$$h_t = \bar{A}h_{t-1} + \bar{B}x_t \quad (2a)$$

$$y_t = Ch_t \quad (2b)$$

$$\bar{K} = (C\bar{B}, C\bar{A}\bar{B}, \dots, C\bar{A}^k\bar{B}, \dots) \quad (3a)$$

$$y = x * \bar{K} \quad (3b)$$

$$\bar{A} = \exp(\Delta A) \quad \bar{B} = (\Delta A)^{-1}(\exp(\Delta A) - I) \cdot \Delta B \quad (4)$$

Материалы

- [NVIDIA GPU Gems 3: Chapter 39 Parallel Prefix Sum \(Scan\) with CUDA](#)
- [Записи курса “Вычисления на видеокартах” в CS Space](#)
- [Лекция 3 “Исполнение потоков, иерархия памяти CUDA”](#)