Lecture 17:

Hardware Specialization and Algorithm Specific Programming

Parallel Computing

Stanford CS149, Fall 2023

Energy-constrained computing

Energy (Power x Time)-constrained computing

- Supercomputers are energy constrained
  - Due to shear scale of machine
  - Overall cost to operate (power for machine and for cooling)

- Datacenters are energy constrained
  - Reduce cost of cooling
  - Reduce physical space requirements

- Mobile devices are energy constrained
  - Limited battery life
  - Heat dissipation

Performance and Power

Performance

Energy efficiency

$$Power = \frac{Ops}{second} \times \frac{Joules}{Op}$$

FIXED ⬆ ⬇

What is the magnitude of improvement from specialization?

Specialization (fixed function) ⇒ better energy efficiency

Pursuing highly efficient processing…
(specializing hardware beyond just parallel CPUs and GPUs)

Why is a “general-purpose processor” so inefficient?

**Wait… this entire class we’ve been talking about making efficient use out of multi-core CPUs and GPUs…
and now you’re telling me these platforms are “inefficient”?**

Consider the complexity of executing an instruction on a modern processor…

Read instruction ⟶ Address translation, communicate with icache, access icache, etc.

Decode instruction ⟶ Translate op to uops, access uop cache, etc.

Check for dependencies/pipeline hazards

Identify available execution resource

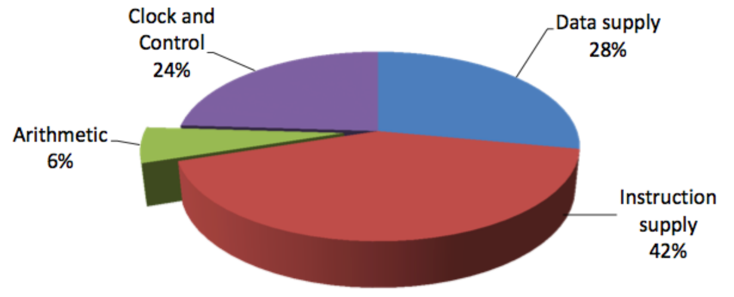
Use decoded operands to control register file SRAM (retrieve data)

Move data from register file to selected execution resource

Perform arithmetic operation

Move data from execution resource to register file

Use decoded operands to control write to register file SRAM



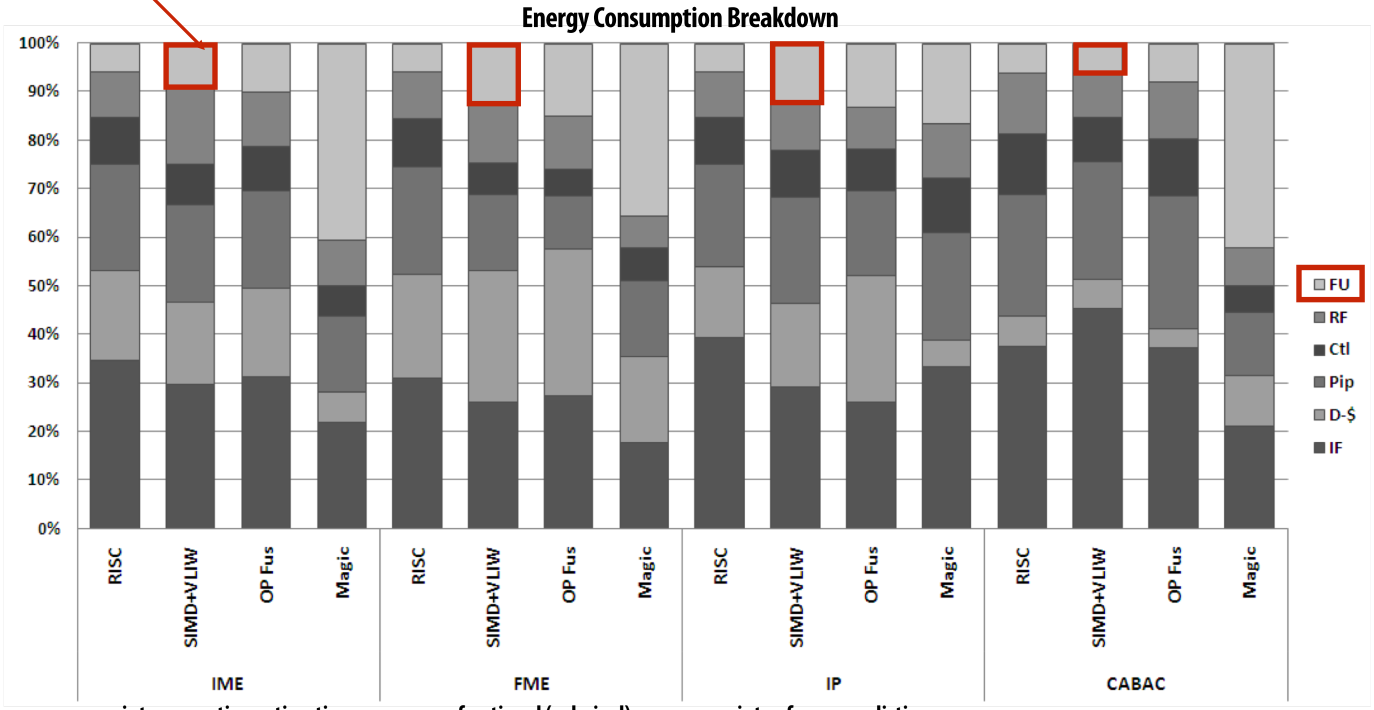
Efficient Embedded Computing [Dally et al. 08]
[Figure credit Eric Chung]

Review question:
How does SIMD execution reduce overhead of certain types of computations?
What properties must these computations have?

H.264 video encoding: fraction of energy consumed by functional units is small (even when using SIMD)

Even after encoding implemented with SIMD instruction

[Hameed et al. ISCA 2010]



integer motion estimation

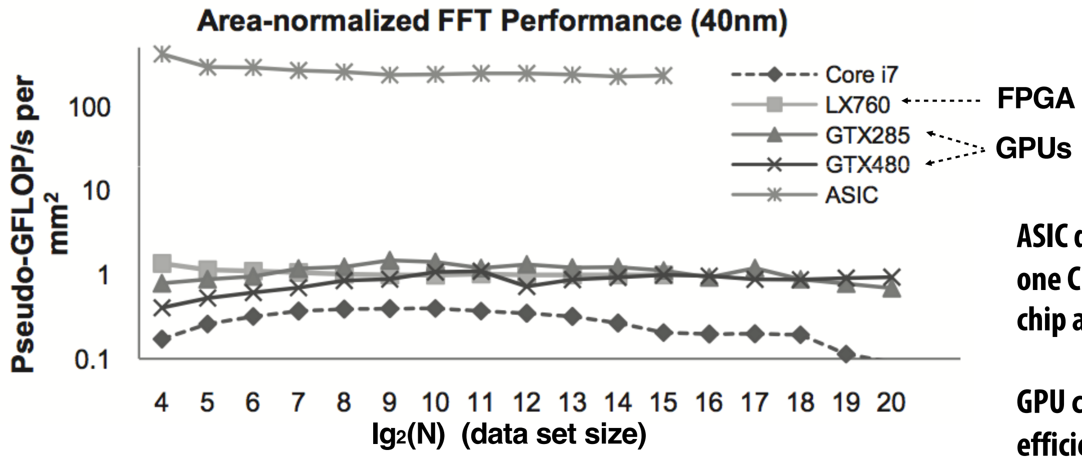
fractional (subpixel) motion estimation

intra-frame prediction, DTC, quantization

arithmetic encoding

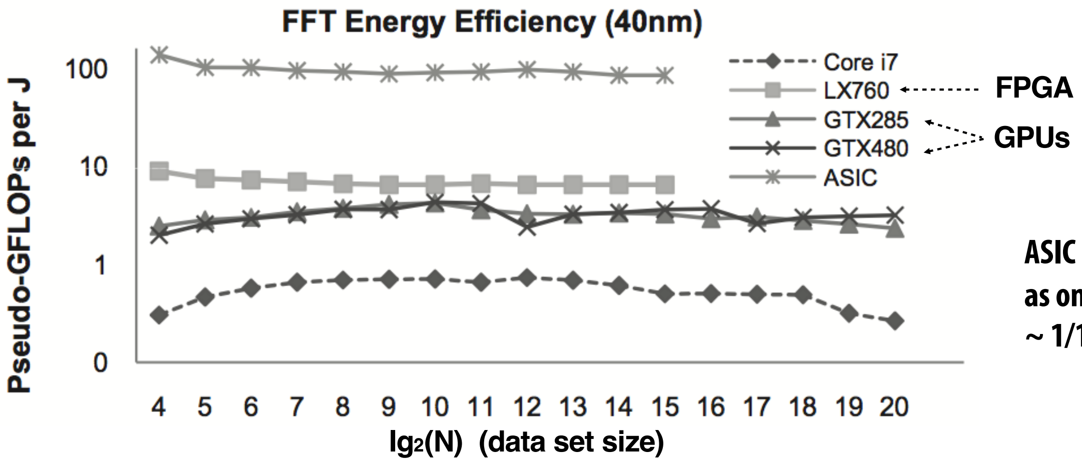
FU = functional units	Pip = pipeline registers (interstage)
RF = register fetch	D-$ = data cache
Ctrl = misc pipeline control	IF = instruction fetch + instruction cache

Fast Fourier transform (FFT): throughput and energy benefits of specialization



ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.



ASIC delivers same performance as one CPU core using only ~ 1/100th the power

[Chung et al. MICRO 2010]

Digital signal processors (DSPs)

Programmable processors, but simpler instruction stream control paths

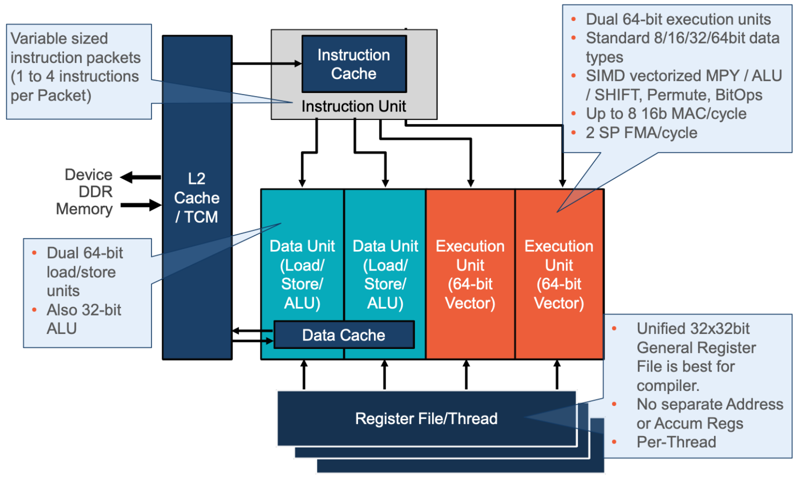
Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction (amortize cost of control)

Example: Qualcomm Hexagon DSP

Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors

VLIW: "very-long instruction word"
Single instruction specifies multiple different operations to do at once (contrast to SIMD)

Below: innermost loop of FFT
Hexagon DSP performs 29 "RISC" ops per cycle



Variable sized instruction packets (1 to 4 instructions per Packet)
Instruction Cache
Instruction Unit
Dual 64-bit execution units
Standard 8/16/32/64bit data types
SIMD vectorized MPY / ALU / SHIFT, Permute, BitOps
Up to 8 16b MAC/cycle
2 SP FMA/cycle
Device DDR Memory
L2 Cache / TCM
Dual 64-bit load/store units
Also 32-bit ALU
Data Unit (Load/ Store/ ALU)
Data Unit (Load/ Store/ ALU)
Execution Unit (64-bit Vector)
Execution Unit (64-bit Vector)
Data Cache
Register File/Thread
Unified 32x32bit General Register File is best for compiler.
No separate Address or Accum Regs
Per-Thread

64-bit Load and

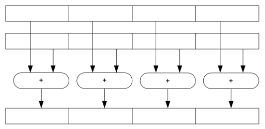
64-bit Store with post-update addressing

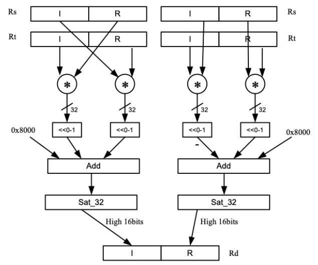
```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

Complex multiply with round and saturation

Zero-overhead loops
- Dec count
- Compare
- Jump top

Vector 4x16-bit Add





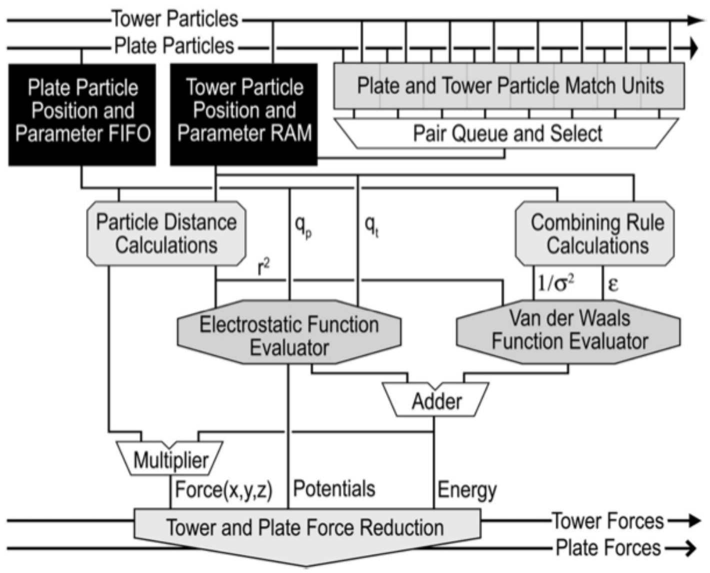
Hexagon DSP is in Google Pixel phone

Anton supercomputer for molecular dynamics

[Developed by DE Shaw Research]

- Simulates time evolution of proteins
- ASIC for computing particle-particle interactions (512 of them in machine)
- Throughput-oriented subsystem for efficient fast-fourier transforms
- Custom, low-latency communication

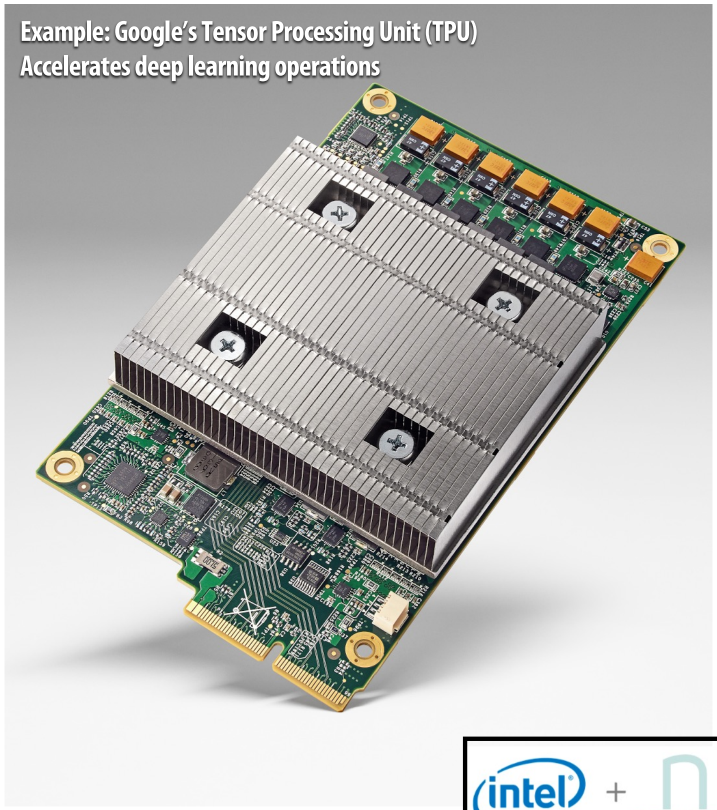
network designed for communication patterns of N-body simulations


Tower Particles
Plate Particles
Plate Particle Position and Parameter FIFO
Tower Particle Position and Parameter RAM
Plate and Tower Particle Match Units
Pair Queue and Select
Particle Distance Calculations
q_p
q_t
Combining Rule Calculations
r^2
$1/\sigma^2$
ε
Electrostatic Function Evaluator
Van der Waals Function Evaluator
Adder
Multiplier
Force(x,y,z)
Potentials
Energy
Tower and Plate Force Reduction
Tower Forces
Plate Forces

Specialized processors for evaluating deep networks

Countless recent papers at top computer architecture research conferences on the topic of ASICs or accelerators for deep learning or evaluating deep networks…

- Cambricon: an instruction set architecture for neural networks, Liu et al. ISCA 2016
- EIE: Efficient Inference Engine on Compressed Deep Neural Network, Han et al. ISCA 2016
- Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing, Albericio et al. ISCA 2016
- Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators, Reagen et al. ISCA 2016
- vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, Rhu et al. MICRO 2016
- Fused-Layer CNN Architectures, Alwani et al. MICRO 2016
- Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Network, Chen et al. ISCA 2016
- PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory, Chi et al. ISCA 2016
- DNNWEAVER: From High-Level Deep Network Models to FPGA Acceleration, Sharma et al. MICRO 2016

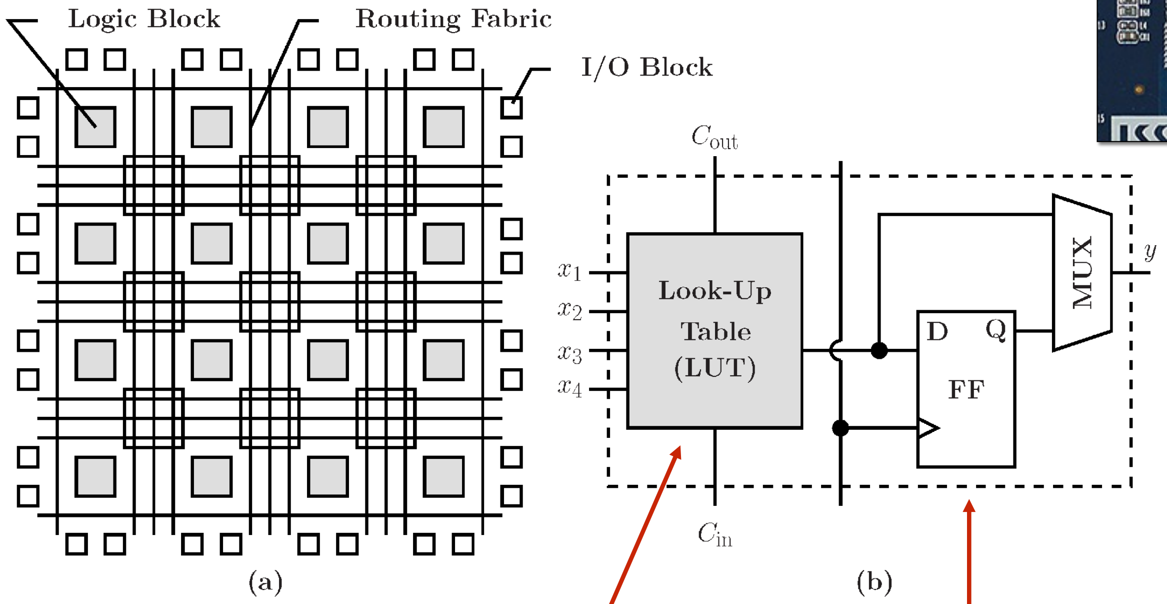

Example: Google's Tensor Processing Unit (TPU)
Accelerates deep learning operations

intel + nervana

Intel Lake Crest ML accelerator (formerly Nervana)

FPGAs (Field Programmable Gate Arrays)

- Middle ground between an ASIC and a processor
- FPGA chip provides array of logic blocks, connected by interconnect
- Programmer-defined logic implemented directly by FGPA



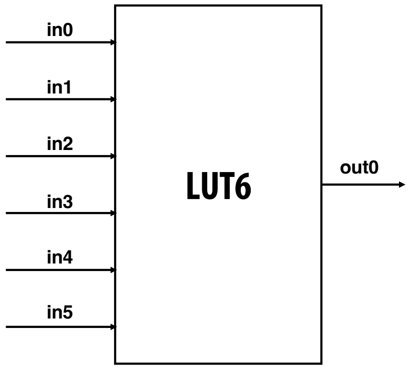
Flip flop (a register)

Programmable lookup table (LUT)

Image credit: Bai et al. 2014

Specifying combinatorial logic as a LUT

- **Example: 6-input, 1 output LUT in Xilinx Virtex-7 FPGAs**
  - **Think of a LUT6 as a 64 element table**



Example:
6-input AND

In	Out
0	0
1	0
2	0
3	0
⋮	⋮
63	1

40-input AND constructed by chaining outputs of eight LUT6's (delay = 3)

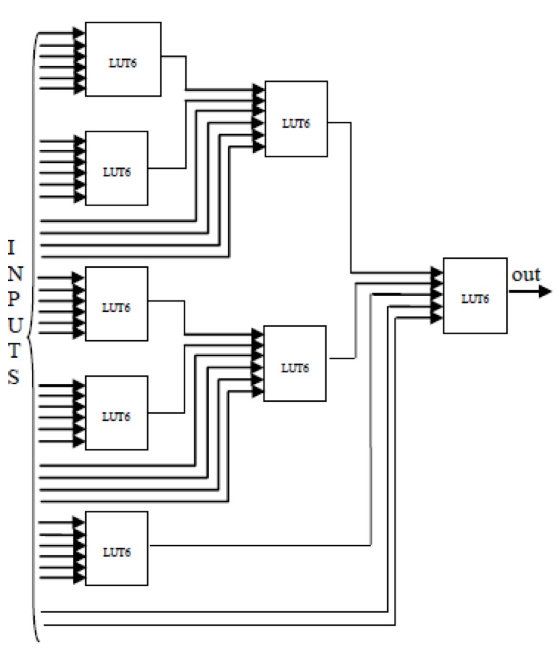
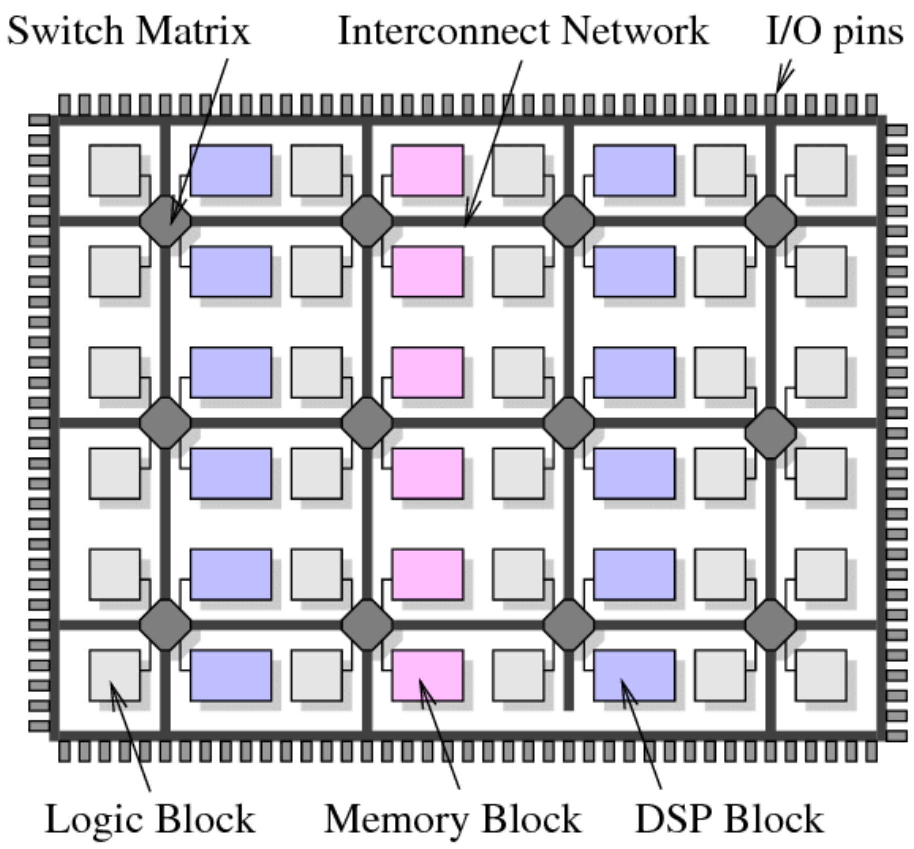


Image credit: [Zia 2013]

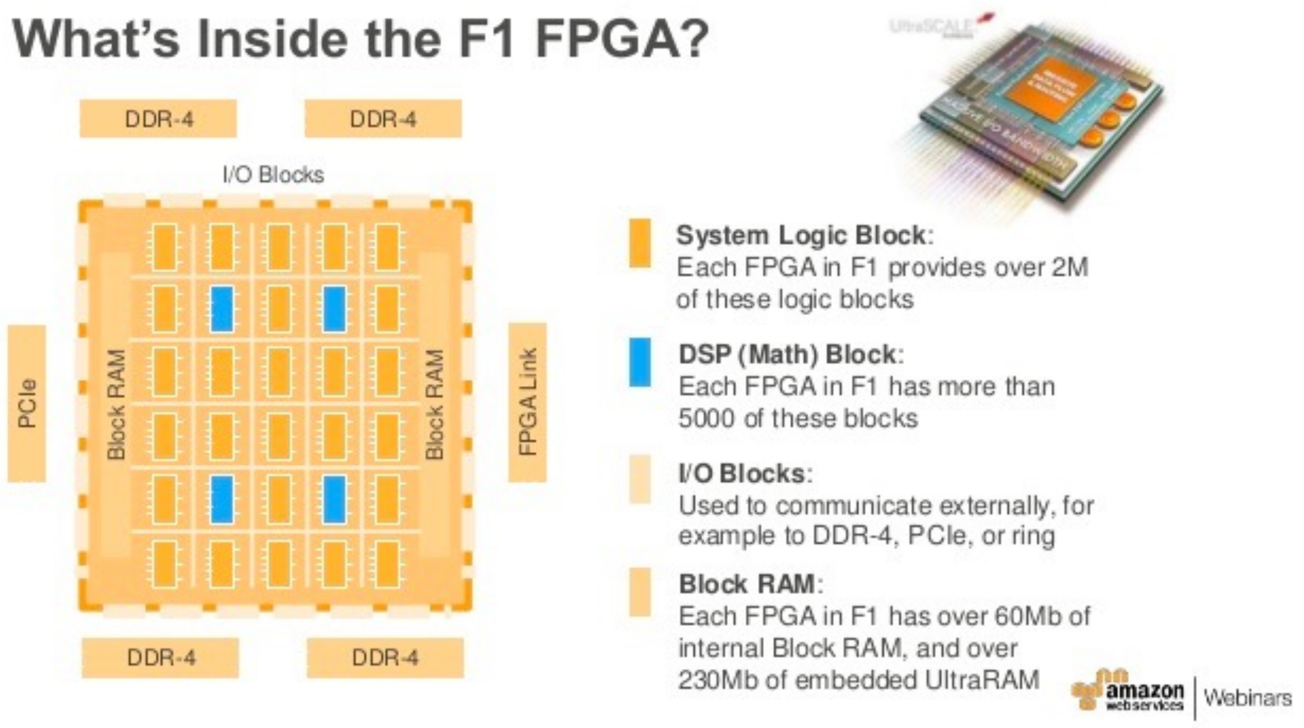
Modern FPGAs



- A lot of area devoted to hard gates
  - Memory blocks (SRAM)
  - DSP blocks (multiplier)

Amazon EC2 F1

- **FPGA's are now available on Amazon cloud services**



What's Inside the F1 FPGA?

🟧 **System Logic Block**: Each FPGA in F1 provides over 2M of these logic blocks

🟦 **DSP (Math) Block**: Each FPGA in F1 has more than 5000 of these blocks

▨ **I/O Blocks**: Used to communicate externally, for example to DDR-4, PCIe, or ring

🟧 **Block RAM**: Each FPGA in F1 has over 60Mb of internal Block RAM, and over 230Mb of embedded UltraRAM

amazon web services | Webinars

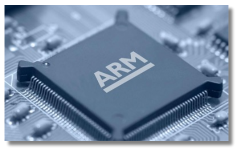
Efficiency benefits of compute specialization

- Rules of thumb: compared to high-quality C code on CPU...
- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf / watt
  - Assuming code maps well to wide data-parallel execution and is compute bound
- Fixed-function ASIC ("application-specific integrated circuit")
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound and is not floating-point math

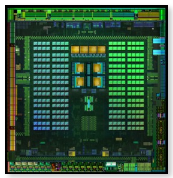
[Source: Chung et al. 2010 , Dally 08]

Choosing the right tool for the job

Energy-optimized CPU



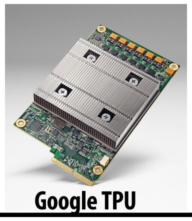
Throughput-oriented processor (GPU)



Programmable DSP

HEXAGON™

Domain Specific Accelerator



FPGA/ reconfigurable logic

ASIC

Video encode/decode, Audio playback, Camera RAW processing, neural nets (future?)

~10X more efficient

~20X

~50X??? (jury still out)

~100-1000X more efficient

Easiest to program

Limited domain of programmability with DSLs (e.g. DNN)

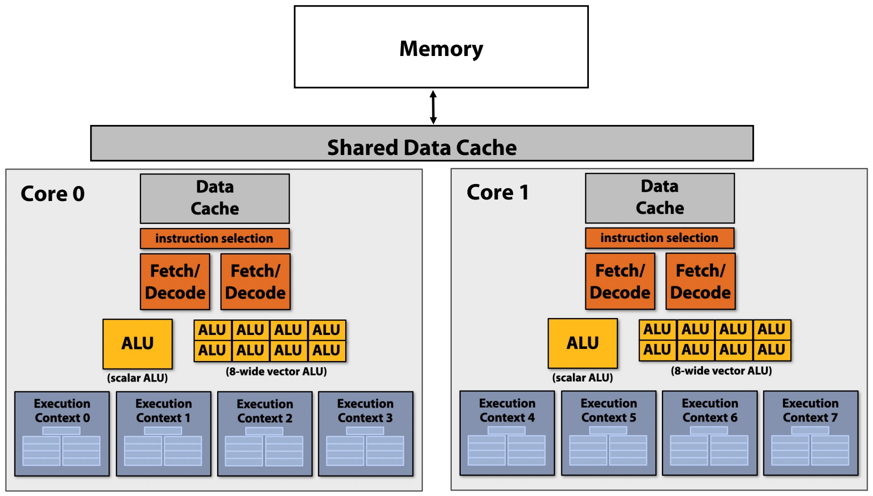
Difficult to program (making it easier is active area of research)

Not programmable + costs 10-100's millions of dollars to design / verify / create

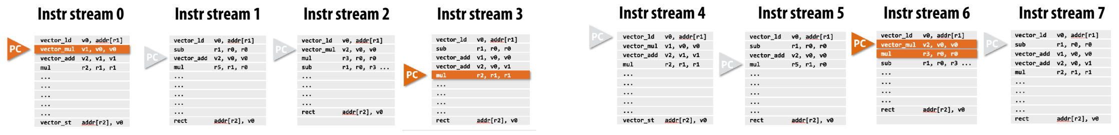
Credit: Pat Hanrahan for this slide design

Mapping Algorithms to Execution Resources

General Purpose Processor



Dual-core processor, multi-threaded cores (4 threads/core).
Two-way superscalar cores: each core can run up to two independent instructions per clock from any of its threads, provided one is scalar and the other is vector



Instr stream 0
```
vector_ld    v0, addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v1, v1
mul          r2, r1, r1
...
...
...
...
...
vector_st    addr[r2], v0
```

Instr stream 1
```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v2, v0, v0
mul          r5, r1, r0
...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 2
```
vector_ld    v0, addr[r1]
vector_mul   v2, v0, v0
mul          r3, r0, r0
sub          r1, r0, r3 ...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 3
```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v1, v0, v0
vector_add   v2, v0, v1
mul          r2, r1, r1
...
...
...
...
rect         addr[r2], v0
```

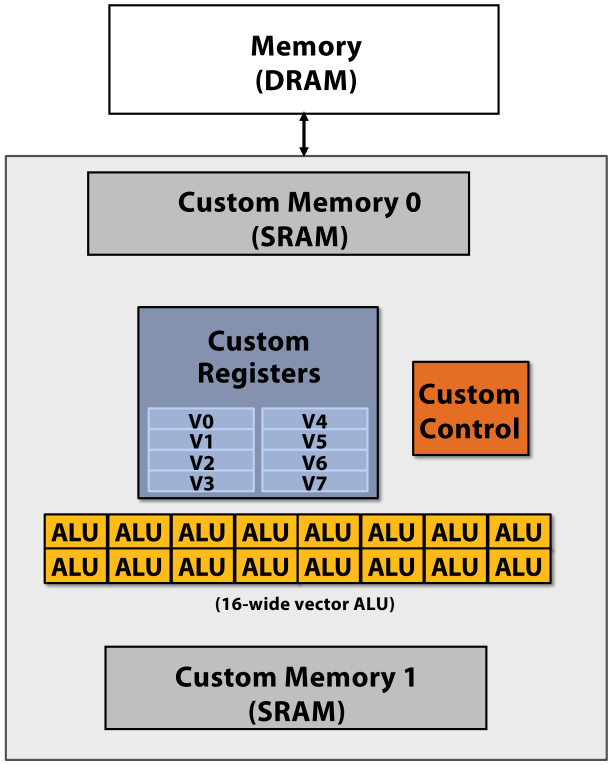
Instr stream 4
```
vector_ld    v0, addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v1, v1
mul          r2, r1, r1
...
...
...
...
...
vector_st    addr[r2], v0
```

Instr stream 5
```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v2, v0, v0
mul          r5, r1, r0
...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 6
```
vector_ld    v0, addr[r1]
vector_mul   v2, v0, v0
mul          r3, r0, r0
sub          r1, r0, r3 ...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 7
```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v1, v0, v0
vector_add   v2, v0, v1
mul          r2, r1, r1
...
...
...
...
rect         addr[r2], v0
```

Special Purpose Processor (Accelerator)



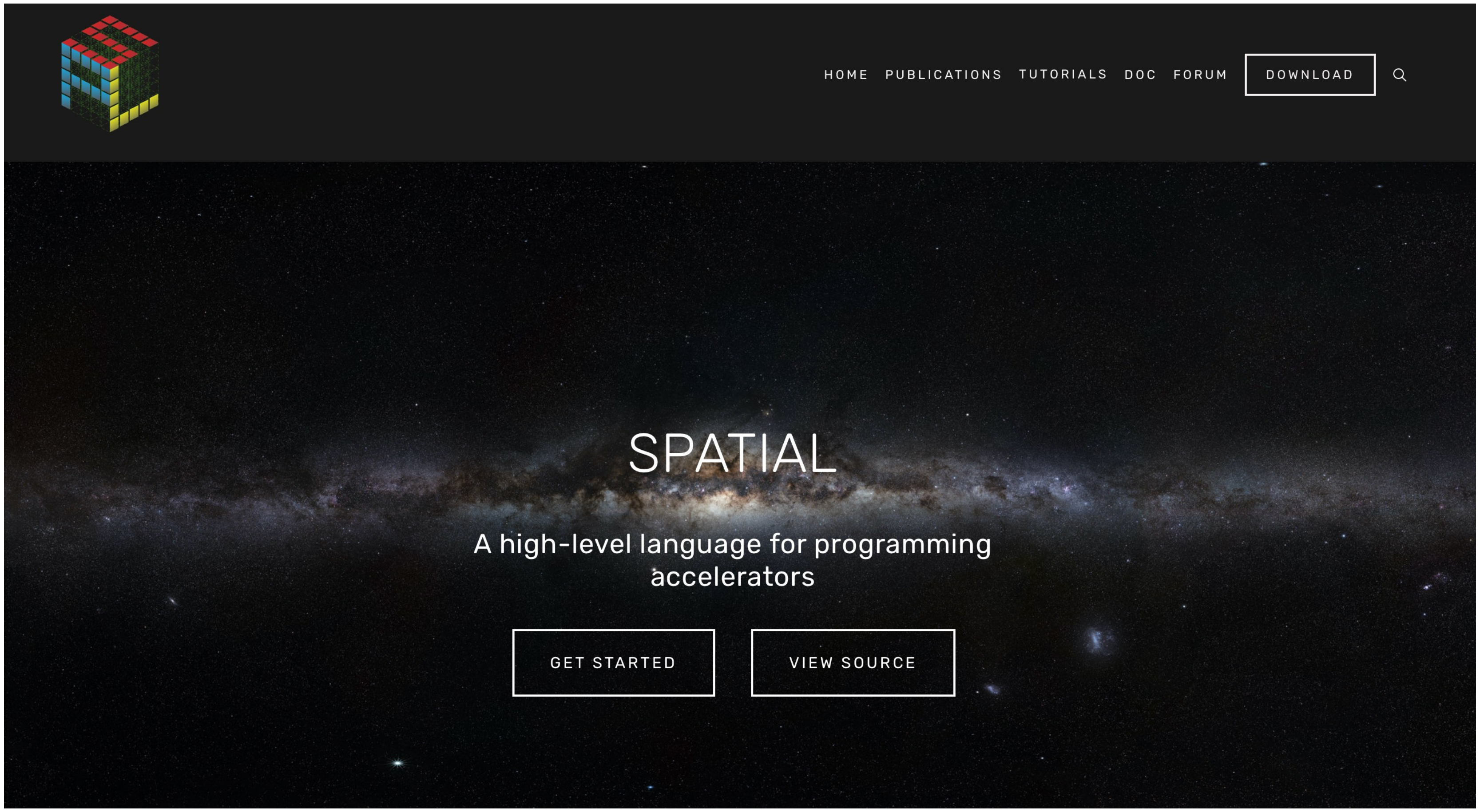
So You Want to Design an Accelerator for Your Algorithm

- **Traditionally, you must spend years becoming an expert in VHDL or Verilog, Chisel…**

- **High-Level Synthesis (HLS): Vivado HLS, Intel OpenCL, and Xilinx SDAccel**
  - **Restricted C with pragmas**
  - **These tools sacrifice performance and are difficult to use**

- **Spatial is a high-level language for designing hardware accelerators that was designed to enable performance-oriented programmers to specify**
  - **Parallelism: specialized compute**
  - **Locality: specialized memories and data movement**

Spatial-lang.org



SPATIAL

A high-level language for programming accelerators

Spatial: DSL for Accelerator Design

- Simplify configurable accelerator design
  - Constructs to express:
    - Parallel patterns as parallel and pipelined datapaths
      - Independent parallelism
      - Dependent parallelism
    - hierarchical control
    - explicit memory hierarchies
    - Explicit parameters
  - All parameters exposed to the compiler
  - Simple APIs to manage CPU $\Leftrightarrow$ Accelerator communication

- Allows programmers to focus on “interesting stuff”
  - Designed for performance-oriented programmers (parallelism and locality)
  - More intuitive than CUDA: dataflow instead of threads

The Spatial Language: Memory Templates

Explicit memory hierarchy
Typed storage templates

```
val buffer = SRAM[UInt8](C)
val image  = DRAM[UInt8](H,W)
```

Registers

```
val accum  = Reg[Double]
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int](R,C)
val pixels = ShiftReg[UInt8](R,C)
```

Explicit transfers across memory hierarchy
Dense and sparse access

```
buffer load image(i, j::j+C)
buffer gather image(a, 10)
```

Streaming abstractions

```
val videoIn  = StreamIn[RGB]
val videoOut = StreamOut[RGB]
```

The Spatial Language: Control Templates

Blocking/non-blocking interaction with CPU

```
Accel { … }

Accel(*) { … }
```

Arbitrary state machine / loop nesting with implicit control signals

```
FSM[Int]{s => s != DONE }{
  case STATE0 =>
    Foreach(C by 1){j => … }
  case STATE1 => …
    Reduce(0)(C by 1){i => … }

}{s => nextState(s) }
```

The Spatial Language: Design Parameters

Spatial templates capture a variety of design parameters:

Explicit parallelization factors

```
val P = 16 (1 → 32)
Reduce(0)(N by 1 par P){i =>
  data(i)
}{(a,b) => a + b}
```

Implicit/Explicit control schemes

```
Stream.Foreach(0 until N){i =>
  …
}
```

Explicit size parameters for stride and buffer sizes

```
val B = 64 (64 → 1024)
val buffer = SRAM[Float](B)
Foreach(N by B){i =>
  …
}
```

Implicit memory banking and buffering schemes for parallelized access

```
Foreach(64 par 16){i =>
  buffer(i) // Parallel read
}
```

Inner Product

Code

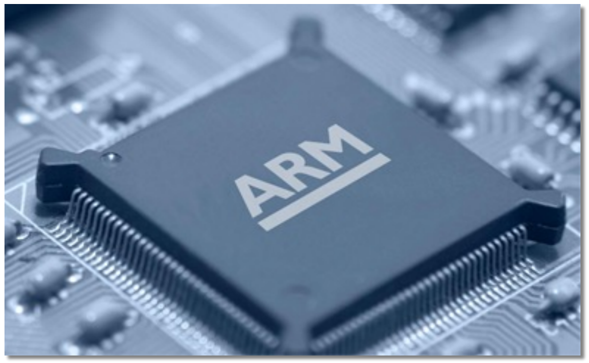
Let’s build an accelerator to see how Spatial works

Sketch of generated hardware

Inner Product in C

```
// Set up accumulator and memory pointers
int output = 0;
int* vec1 = (int*)malloc(N * sizeof(int));
int* vec2 = (int*)malloc(N * sizeof(int));

// Iterate through data and accumulate
for (int i = 0; i < N; i++) {
  output = output + (vec1(i) * vec2(i));
}
```

Here is inner product written in C for a CPU

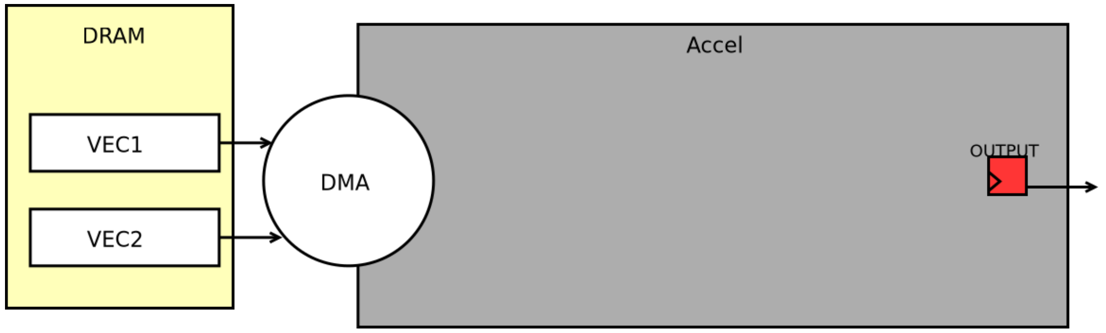


Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator (instantiate hardware)
Accel {














}
```

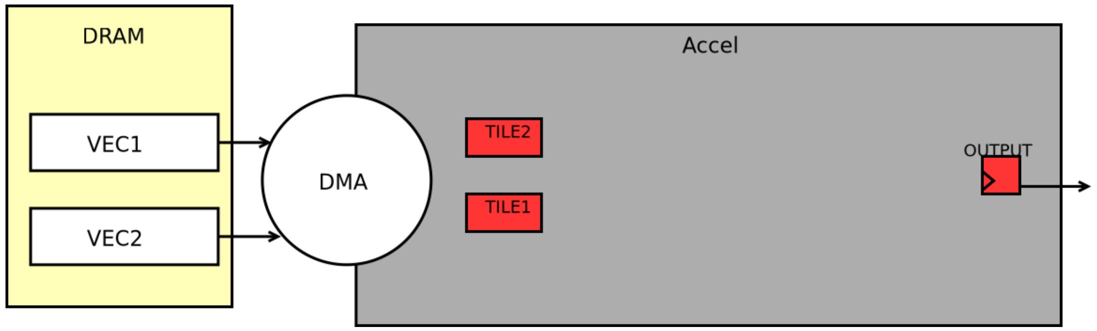
Inner product in Spatial allows the programmer to build a hardware accelerator

- Start of code looks like C example
- Accel instantiates "for" loop in hardware



Inner Product in Spatial

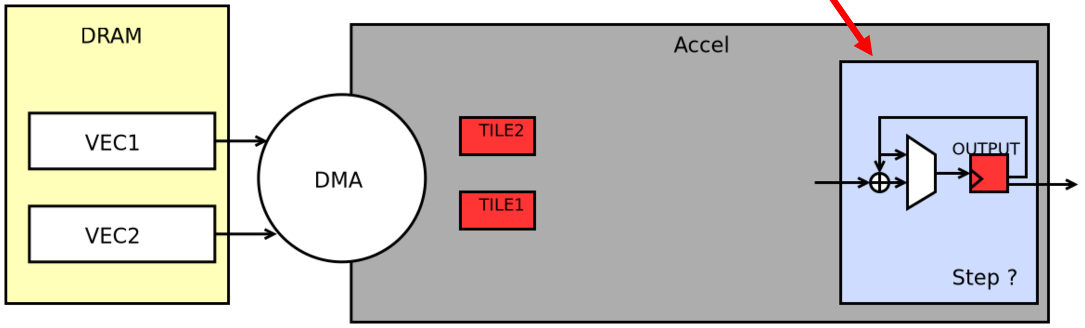
```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)










}
```



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // More controllers coming...







  }{a, b => a + b}
}
```

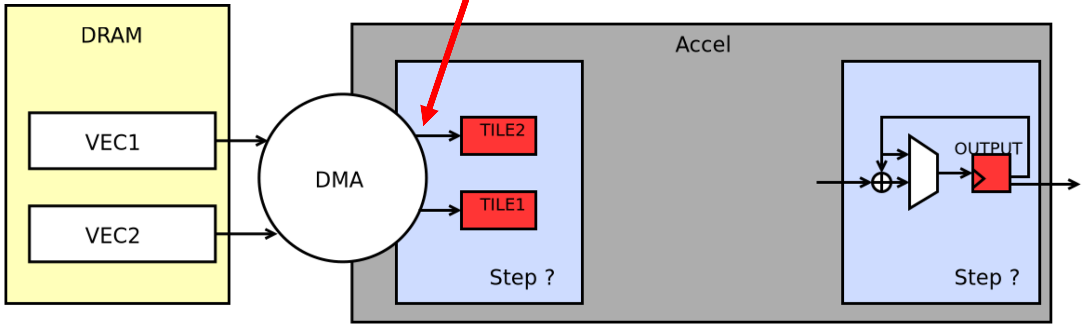
- **Spatial generates multi-step controllers**
  (This Reduce controller's final step will handle the accumulation)



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)





  }{a, b => a + b}
}
```

- **Spatial generates multi-step controllers**
- **Spatial manages communication with DRAM**



Inner Product in Spatial

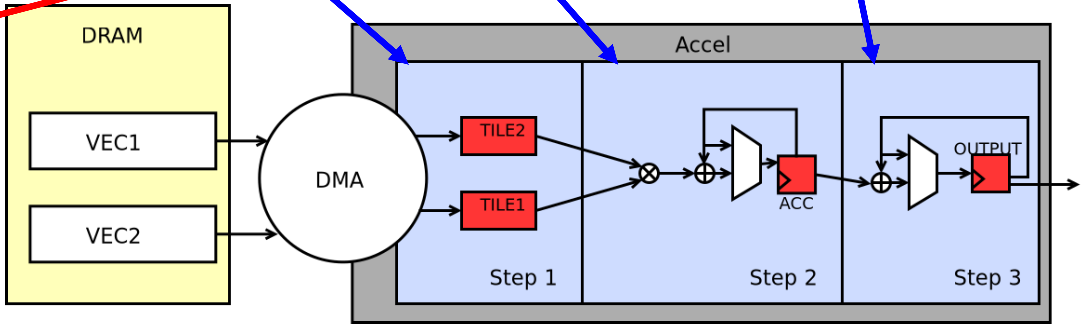
```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par 1){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

- **Spatial generates multi-step controllers**
- **Spatial manages communication with DRAM**

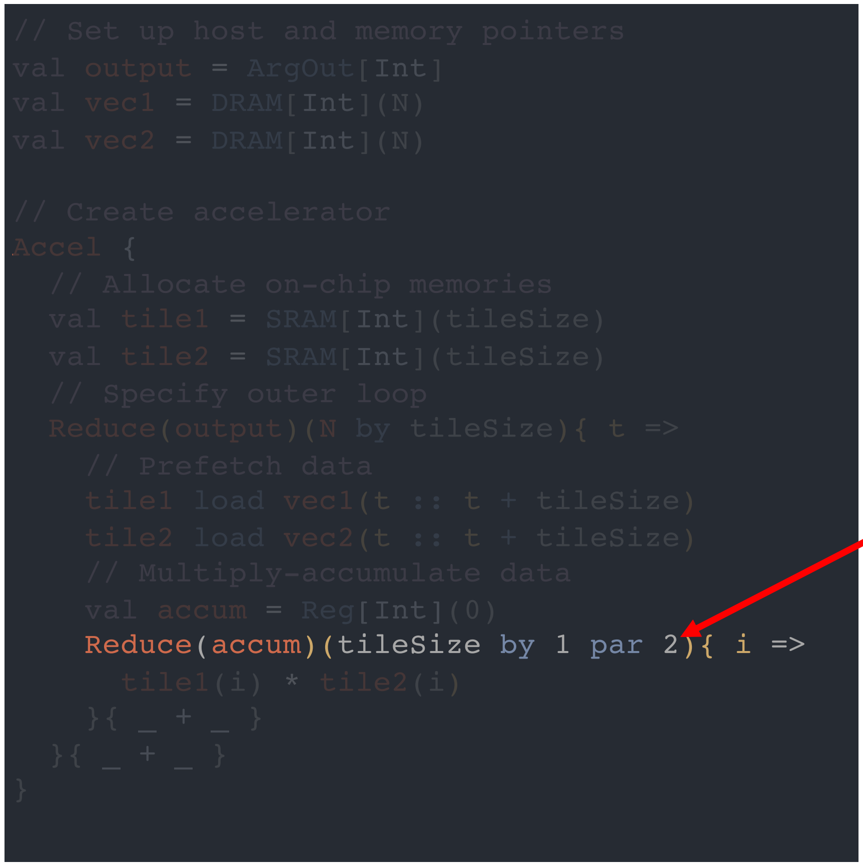
The complete app generates a three-step control

Load ⟶ intra-tile accumulate ⟶ full accumulate

Where is the parallelism?

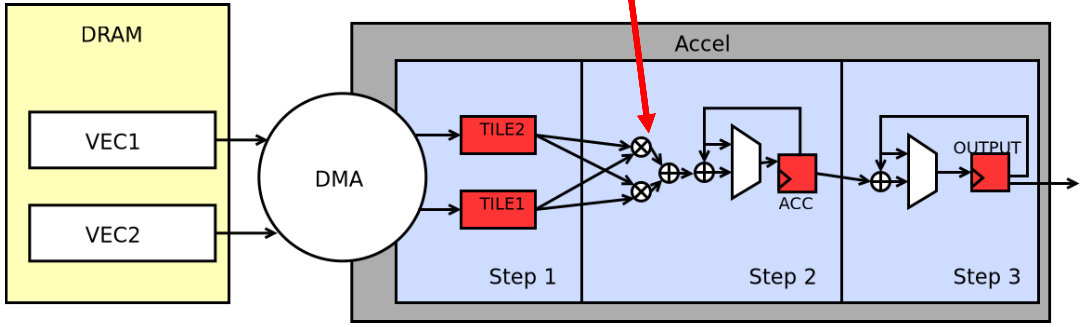


Inner Product in Spatial

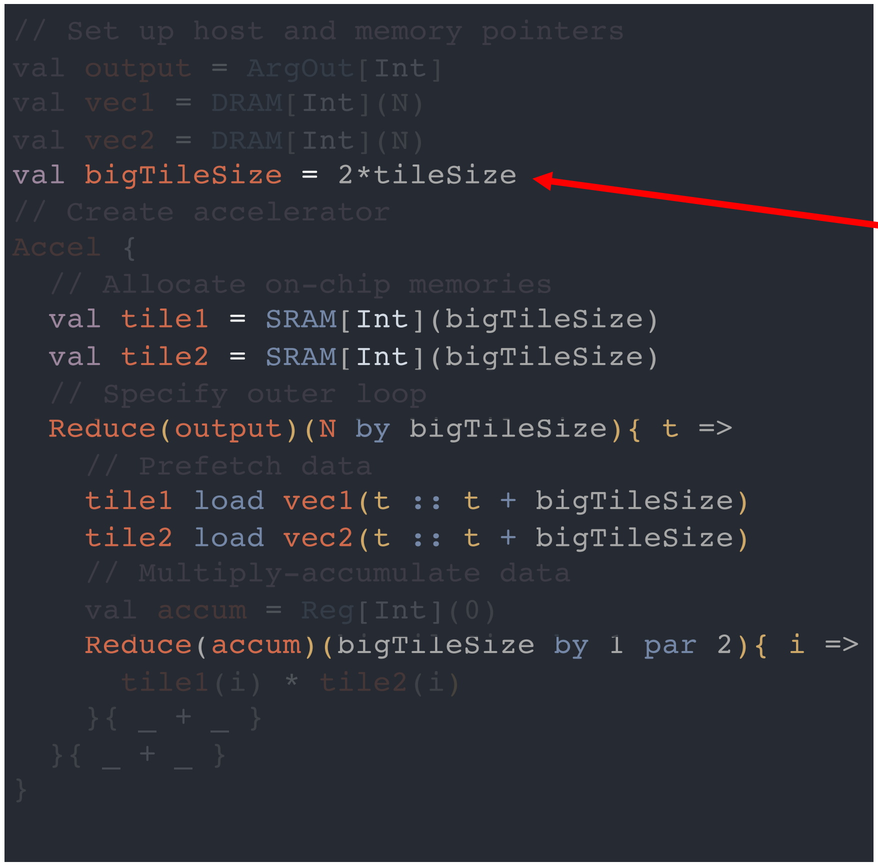


```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par 2){ i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```

- Spatial generates multi-step controllers
- Spatial manages communication with DRAM
- Spatial helps express hardware datapaths

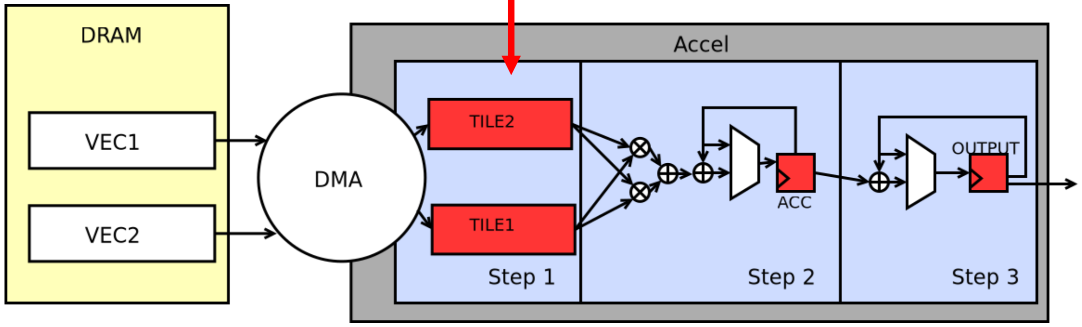


Inner Product in Spatial

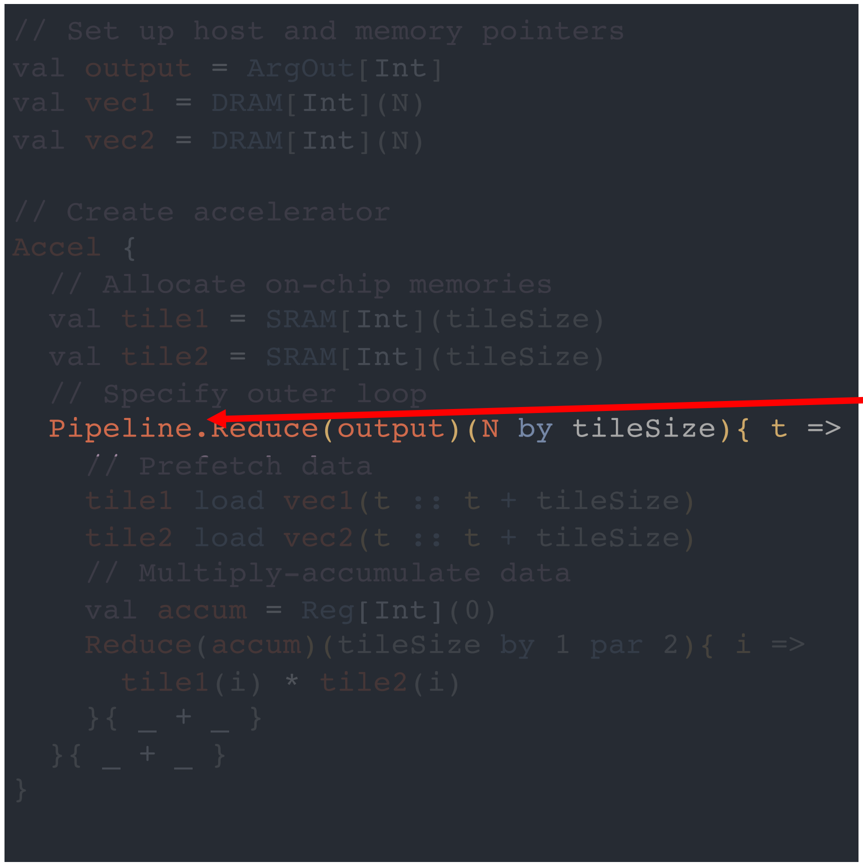


```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)
val bigTileSize = 2*tileSize
// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](bigTileSize)
  val tile2 = SRAM[Int](bigTileSize)
  // Specify outer loop
  Reduce(output)(N by bigTileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + bigTileSize)
    tile2 load vec2(t :: t + bigTileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(bigTileSize by 1 par 2){ i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```

- **Spatial generates multi-step controllers**
- **Spatial manages communication with DRAM**
- **Spatial helps express hardware datapaths**
- **Spatial makes it easy to tile**

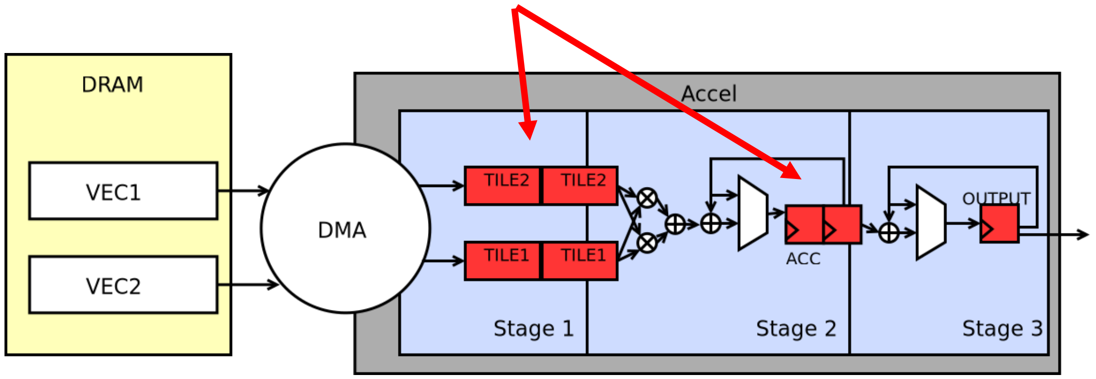


Inner Product in Spatial



```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Pipeline.Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par 2){ i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```

- Spatial generates multi-step controllers
- Spatial manages communication with DRAM
- Spatial helps express hardware datapaths
- Spatial makes it easy to tile and stream
- Spatial lets the user manage scheduling
  - With annotation, steps (stages) execute in pipelined fashion. "Buffering" of memories is inferred

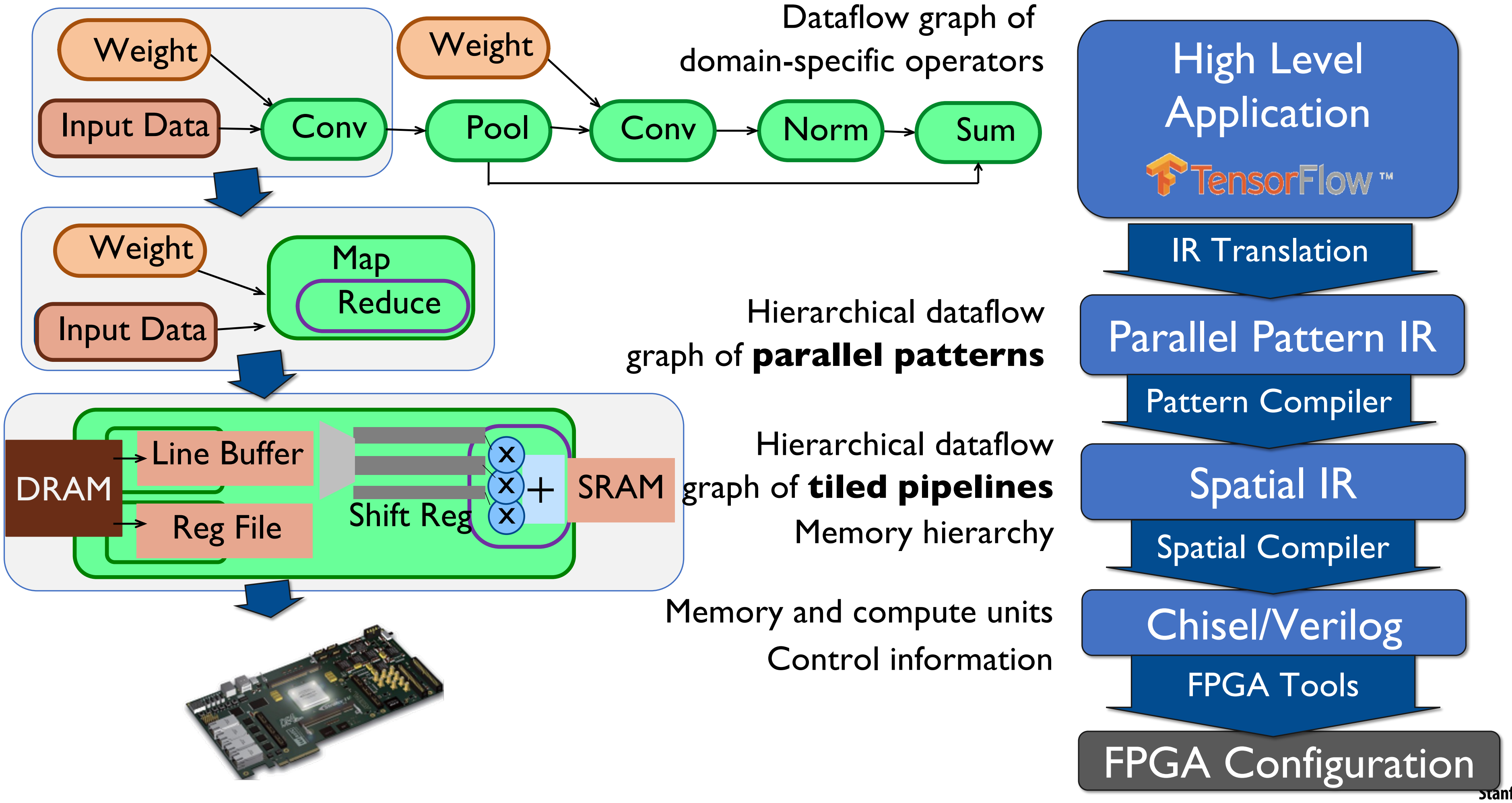


Spatial Question

- **Spatial programmer's responsibility**

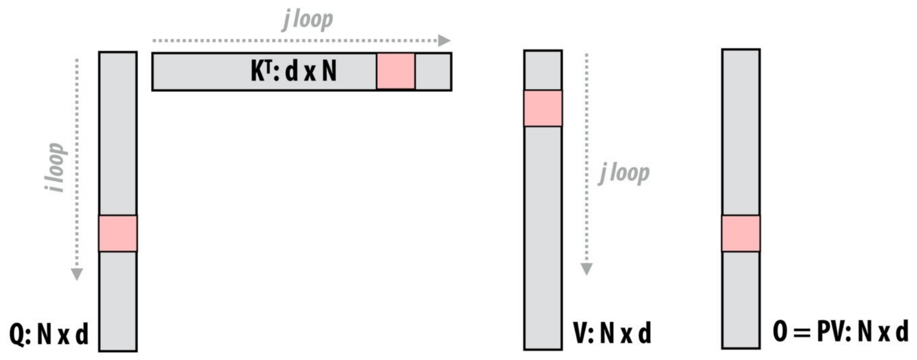
- **Spatial compiler's responsibility**

TensorFlow to FPGA



Recap: Why was Flash Attention powerful?

Fused attention



```
for each j:
  for each i:
     Load block $Q_i$, $K^T{}_j$, $V_j$, $O_i$
     Compute $S_{ij} = Q_i K^T{}_j$
     Compute $M_{ij} = m(S_{ij})$, $P_{ij} = f(S_{ij})$, and $l_{ij} = l(S_{ij})$      (all functions operate row-wise on row-vectors)
     Multiply $P_{ij}V_j$ and accumulate into $O_i$ with appropriate scalings (see previous slide for math)
```

Save memory footprint:
Never materialize N^2 matrix

Save memory bandwidth:
(high arithmetic intensity)
- Read 3 blocks (from Q, K, V)
- Do two matrix multiplies + a few row summations
- Accumulate into O block (which is resident in cache)

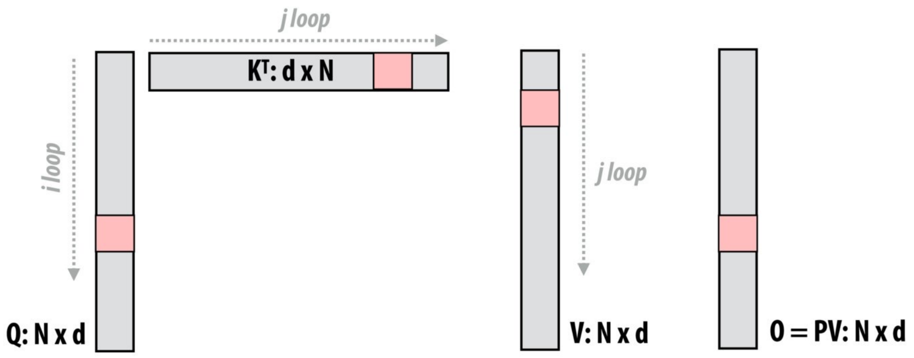
Note there is additional computation vs. the original version (must re-scale prior values of O each step of i-loop)

Recap: Why was FlashAttention Powerful?

Fusion!

With streaming execution, we get these benefits for free! (Free Fusion!)

Fused attention



Save memory footprint:
Never materialize N^2 matrix

Save memory bandwidth:
(high arithmetic intensity)

- Read 3 blocks (from Q, K, V)
- Do two matrix multiplies + a few row summations
- Accumulate into O block (which is resident in cache)

Note there is additional computation vs. the original version (must re-scale prior values of O each step of i-loop)

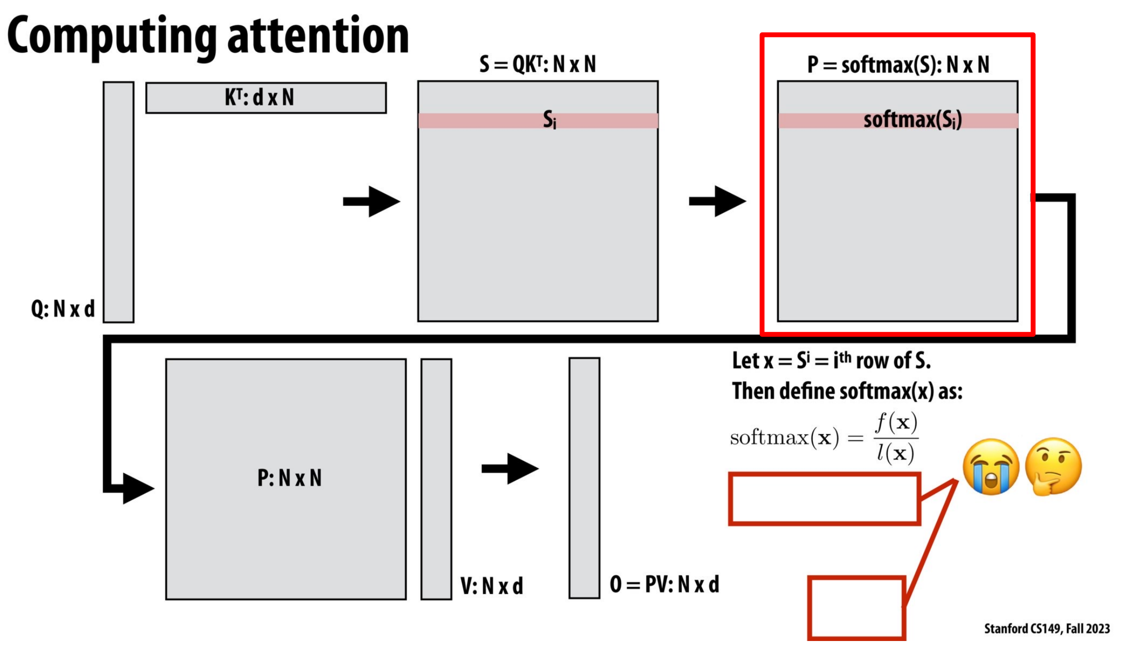
```
for each j:
  for each i:
     Load block $Q_i$, $K^T_j$, $V_j$, $O_i$
     Compute $S_{ij} = Q_i K^T_j$
     Compute $M_{ij} = m(S_{ij})$, $P_{ij} = f(S_{ij})$, and $l_{ij} = l(S_{ij})$      (all functions operate row-wise on row-vectors)
     Multiply $P_{ij}V_j$ and accumulate into $O_i$ with appropriate scalings (see previous slide for math)
```

Streaming execution model: Free Fusion!

- **Kernel-based Execution Model:**
  - FlashAttention prevents the materialization of the N x N matrix
  - However, it requires modifying the algorithm and extra computation
- **Streaming execution model:**
  - Avoids materialization of the N x N matrix
  - **Without algorithmic changes & extra computation**

Preliminary: Softmax

- Softmax is actually a 3-step operation



① Exponential

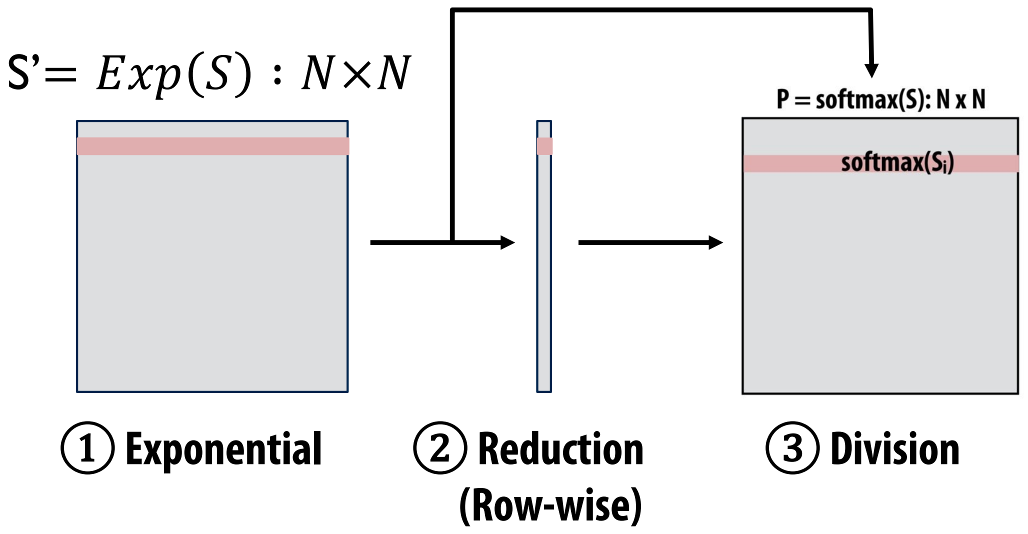
$$P_{ij} = \frac{\boxed{e^{S_{ij}}}}{\boxed{\sum_j e^{S_{ij}}}}$$

③ Division

② Reduction (Row-wise)

Preliminary: Softmax

- Softmax is actually a 3-step operation



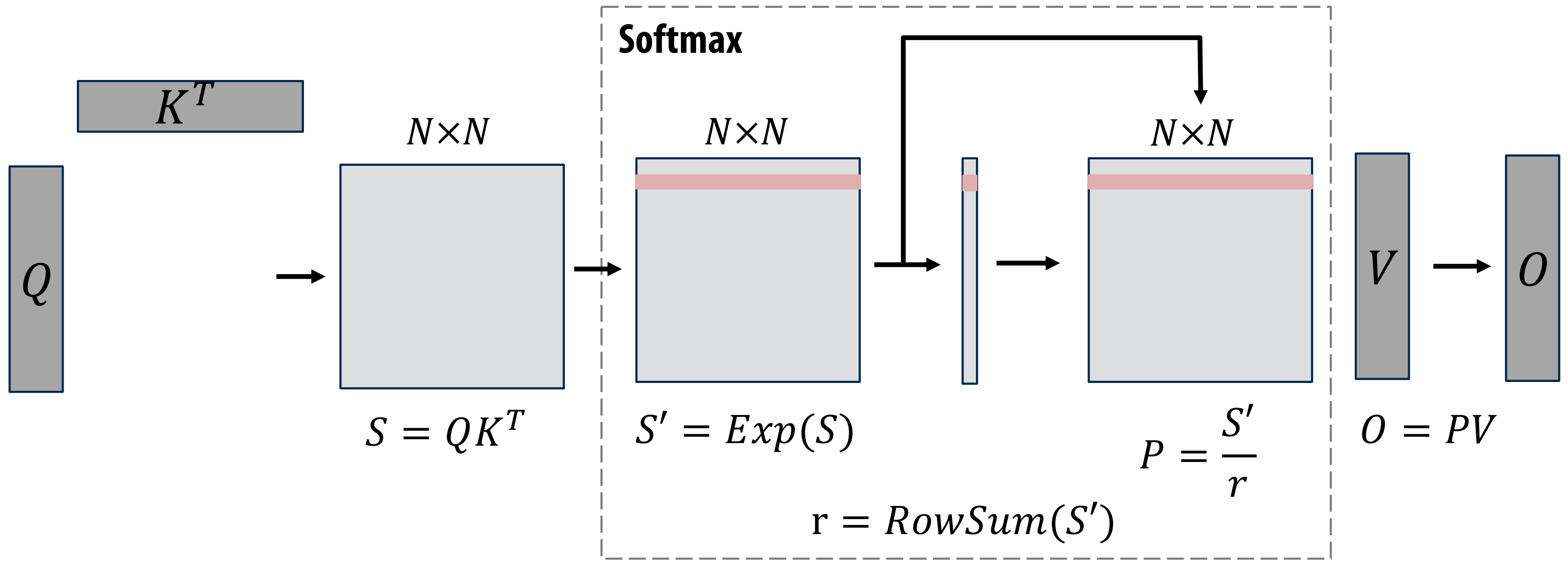
$$P_{ij} = \frac{\boxed{e^{S_{ij}}}}{\boxed{\sum_j e^{S_{ij}}}}$$

① Exponential

② Reduction (Row-wise)

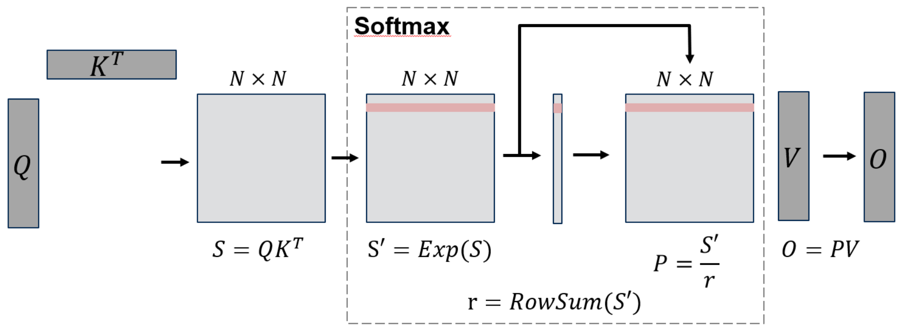
③ Division

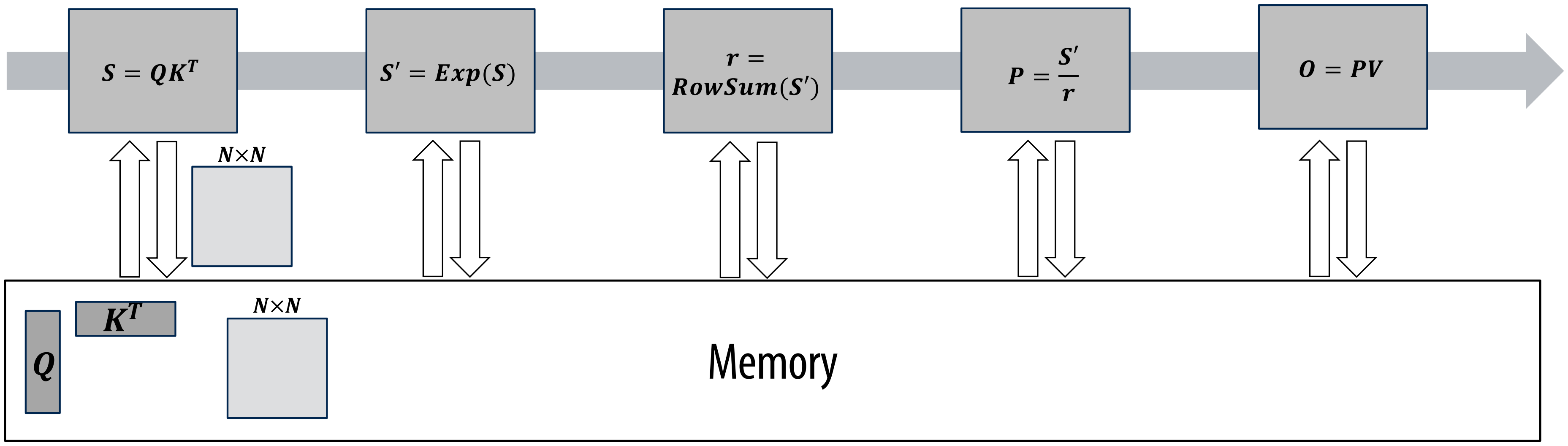
Attention



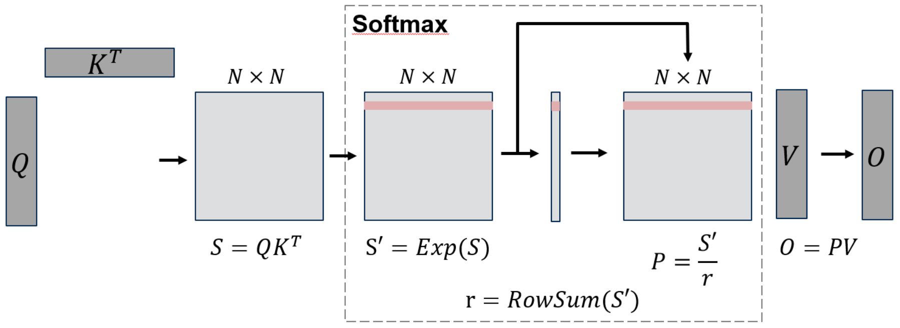
K^T
Q
$N \times N$
$S = QK^T$
Softmax
$N \times N$
$S' = Exp(S)$
$N \times N$
$P = \frac{S'}{r}$
r $= RowSum(S')$
V
O
$O = PV$

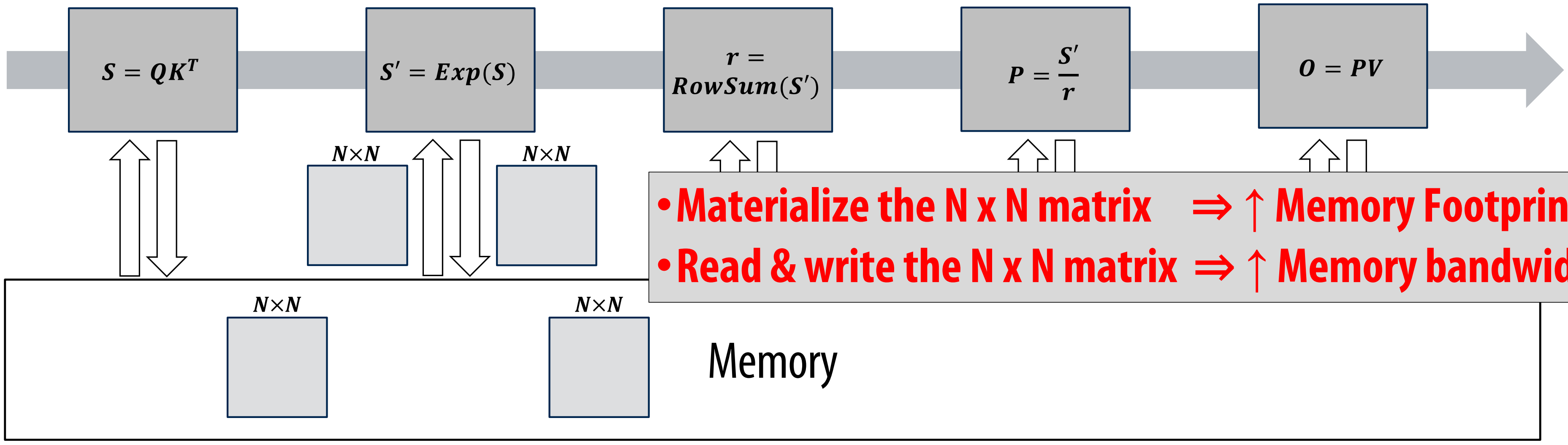
Kernel-based Execution Model





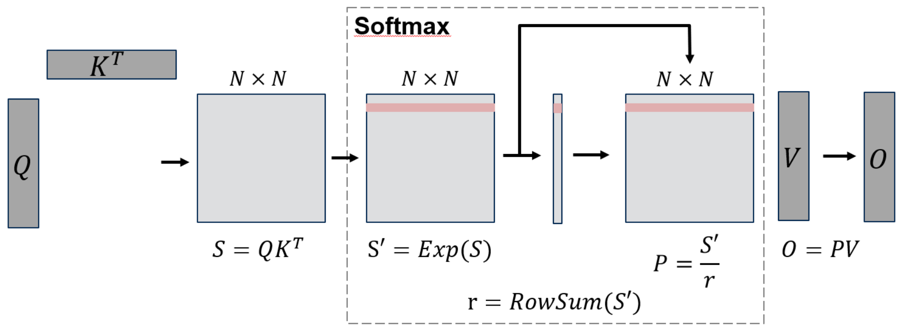
Kernel-based Execution Model

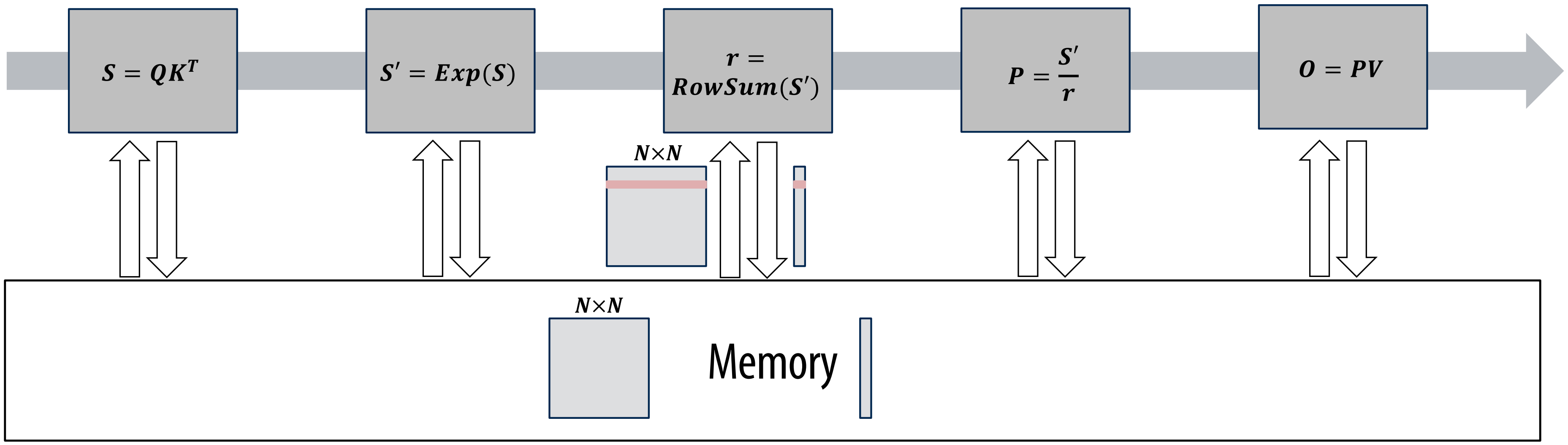




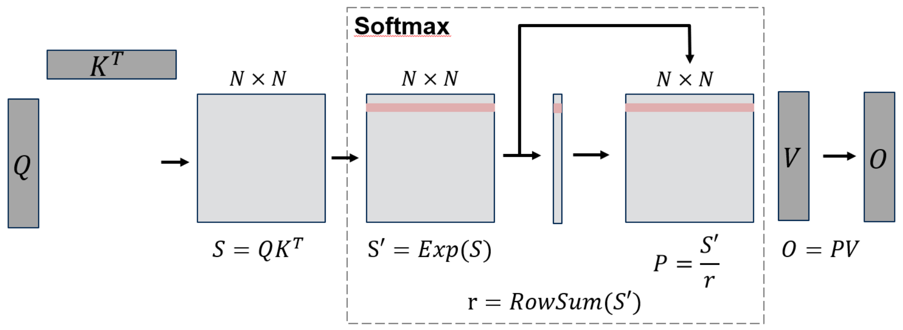
- Materialize the N x N matrix $\Rightarrow \uparrow$ Memory Footprint
- Read & write the N x N matrix $\Rightarrow \uparrow$ Memory bandwidth

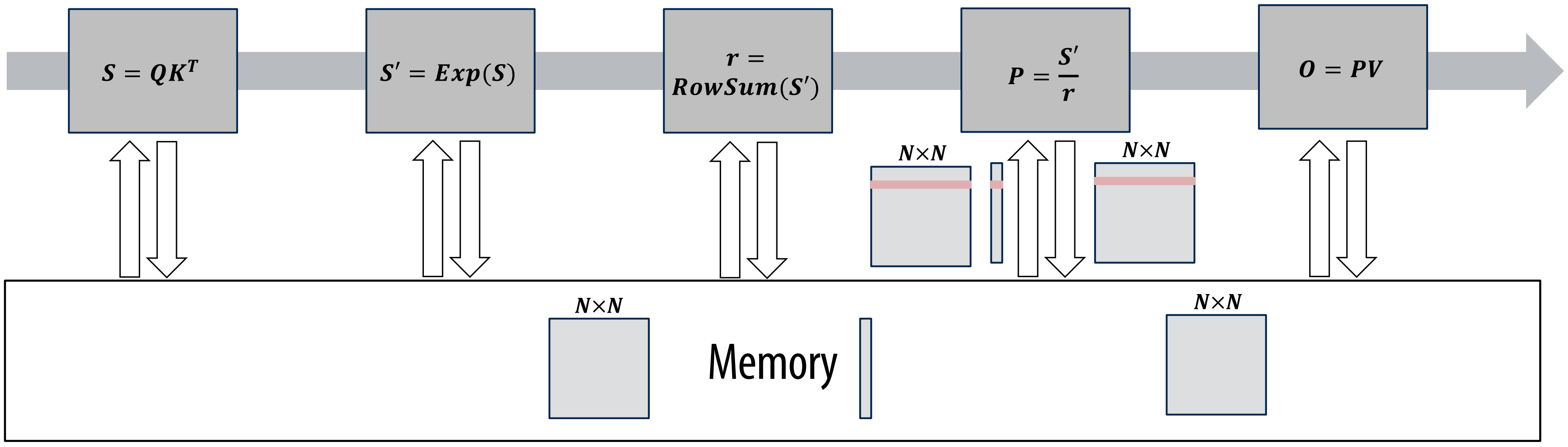
Kernel-based Execution Model



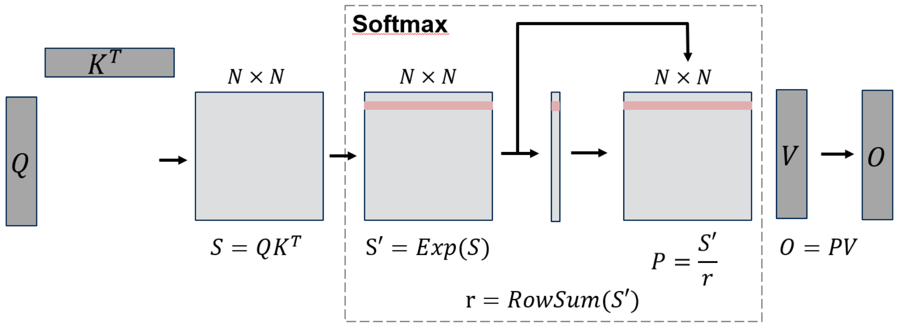


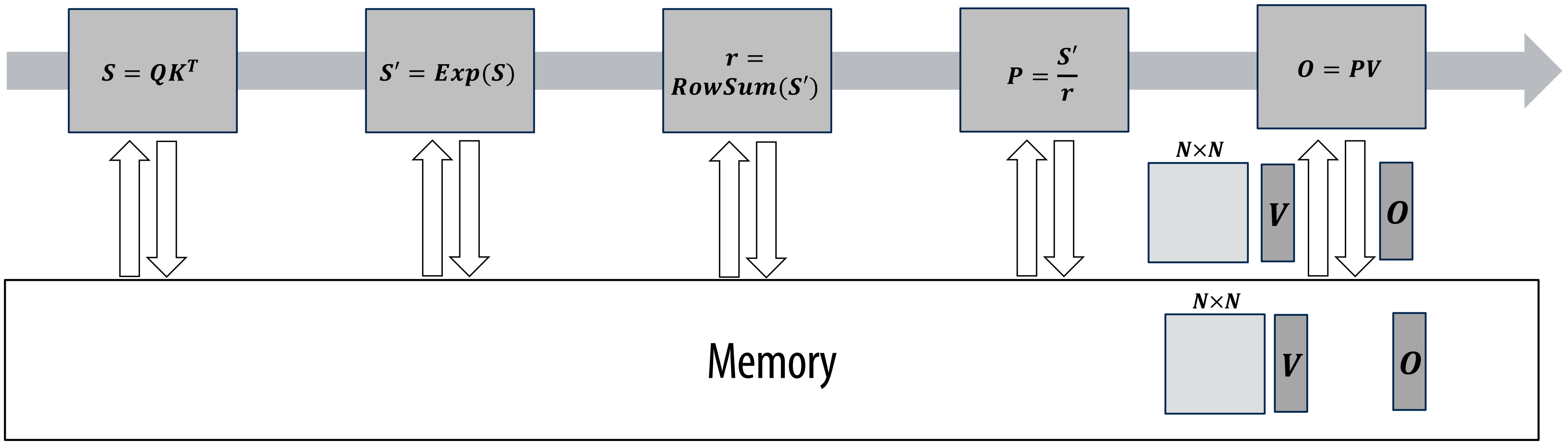
Kernel-based Execution Model



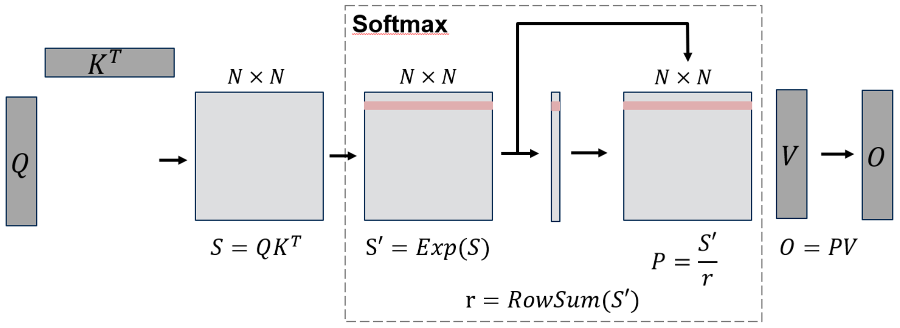


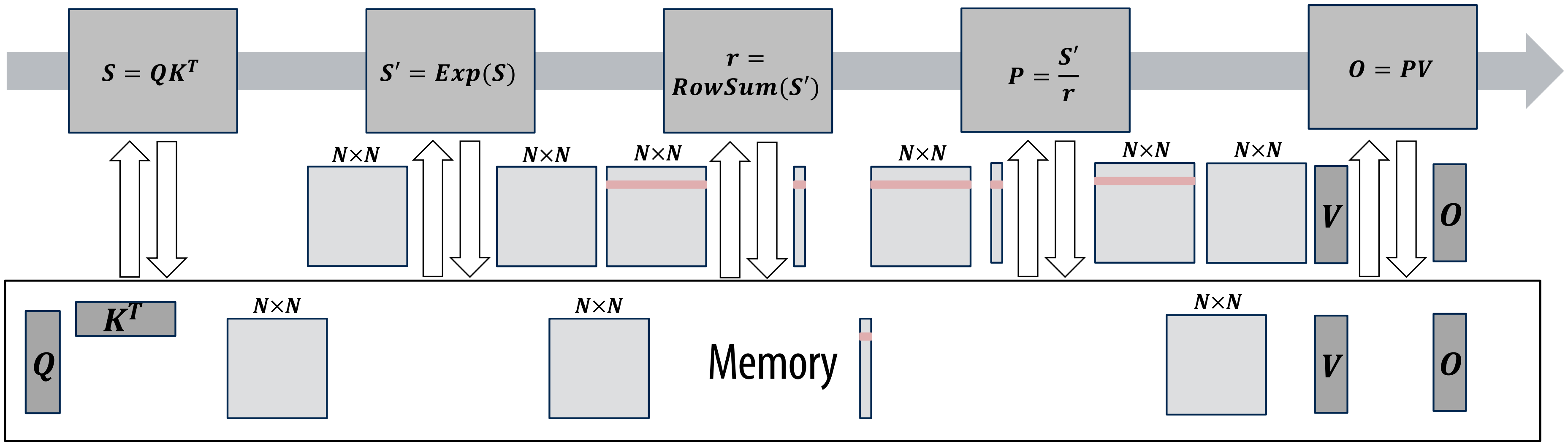
Kernel-based Execution Model





Kernel-based Execution Model (Overview)





Streaming Execution Model

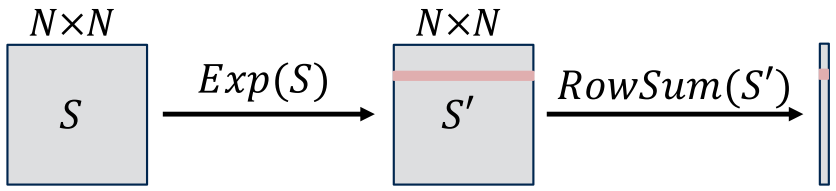
With the streaming execution model, we get fusion for free which means:

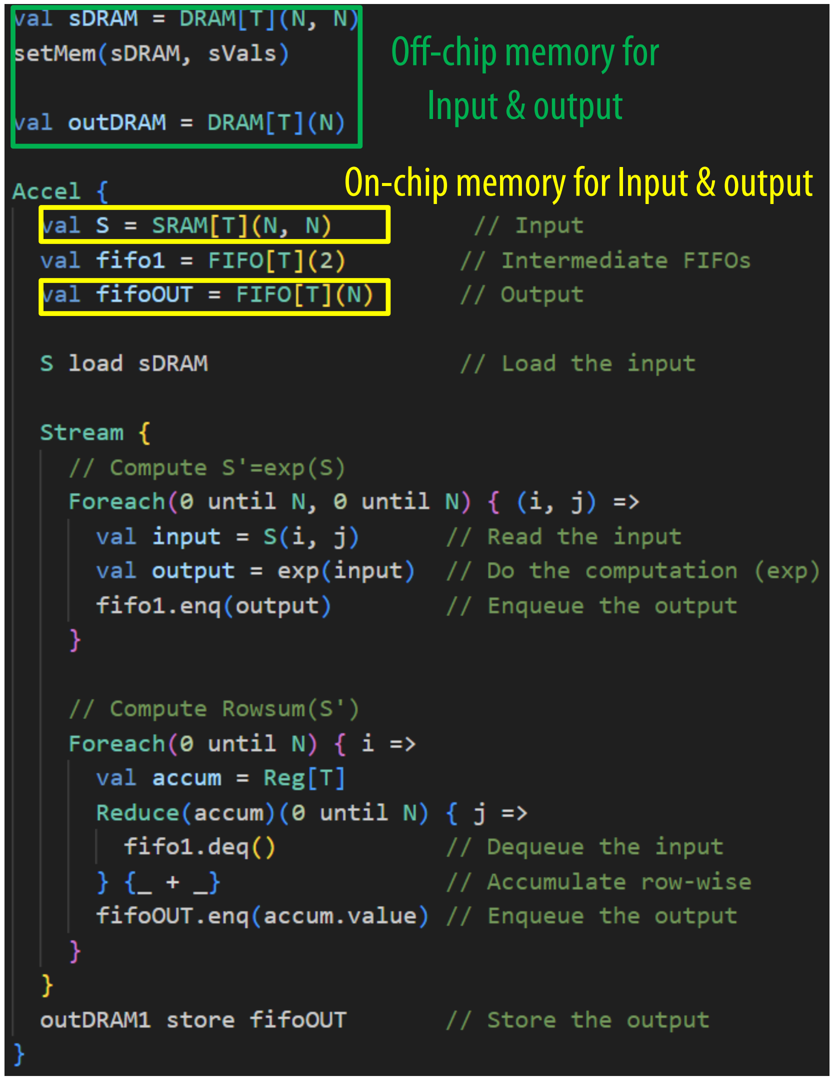
- Avoid materializing the N x N matrix ⇒ **↓Memory Footprint**
- Avoid reading & writing the intermediate N x N matrices ⇒ **↓Memory bandwidth**

Streaming execution Model

An example program in a streaming execution model

- **Computation: Exponential & Rowsum**





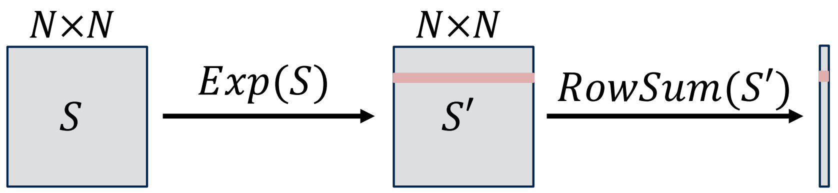
Off-chip memory for Input & output
On-chip memory for Input & output

```
val sDRAM = DRAM[T](N, N)
setMem(sDRAM, sVals)

val outDRAM = DRAM[T](N)

Accel {
  val S = SRAM[T](N, N)          // Input
  val fifo1 = FIFO[T](2)        // Intermediate FIFOs
  val fifoOUT = FIFO[T](N)      // Output

  S load sDRAM                  // Load the input

  Stream {
    // Compute S'=exp(S)
    Foreach(0 until N, 0 until N) { (i, j) =>
      val input = S(i, j)      // Read the input
      val output = exp(input)  // Do the computation (exp)
      fifo1.enq(output)        // Enqueue the output
    }

    // Compute Rowsum(S')
    Foreach(0 until N) { i =>
      val accum = Reg[T]
      Reduce(accum)(0 until N) { j =>
        fifo1.deq()            // Dequeue the input
      } {_ + _}                // Accumulate row-wise
      fifoOUT.enq(accum.value) // Enqueue the output
    }
  }
  outDRAM1 store fifoOUT       // Store the output
}
```

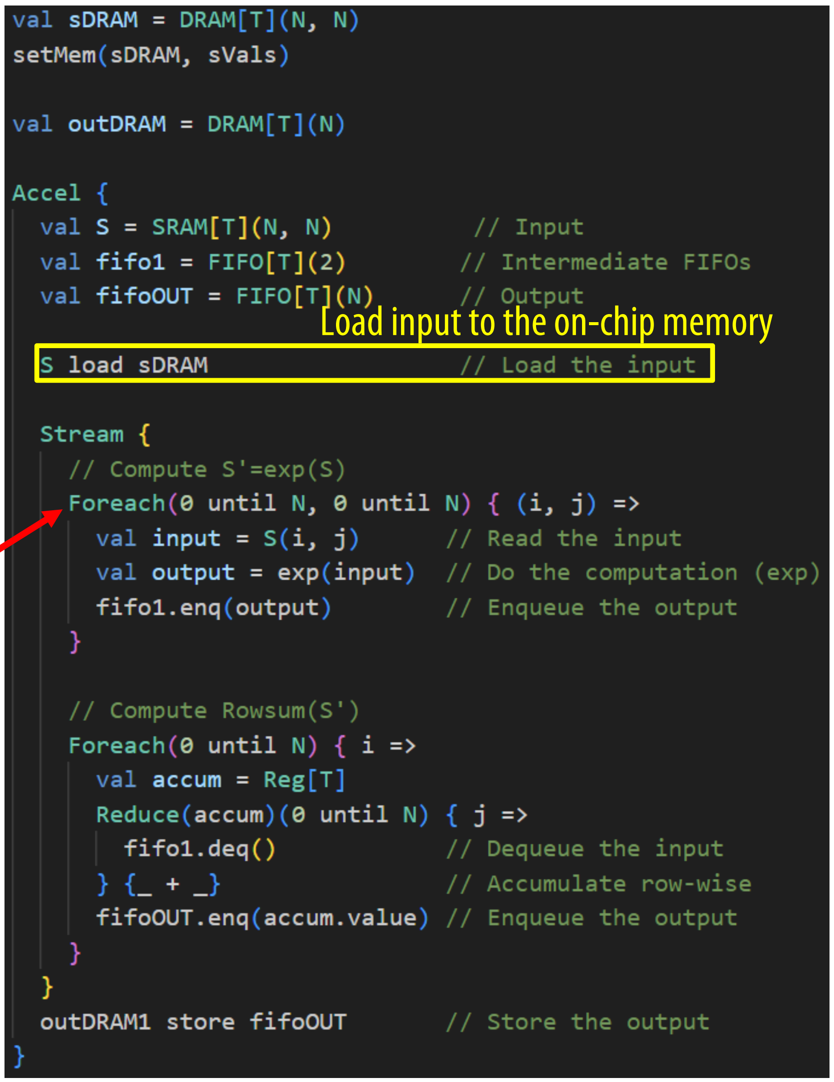
Streaming execution Model

An example program in a streaming execution model

- **Computation: Exponential & Rowsum**



Doing the computation piece-wise



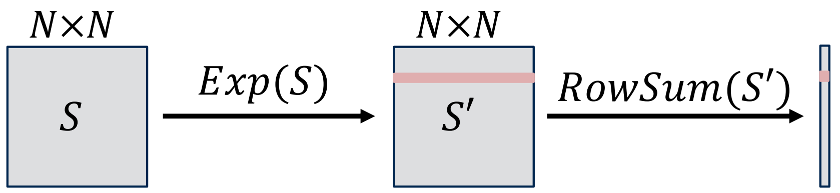
```
val sDRAM = DRAM[T](N, N)
setMem(sDRAM, sVals)

val outDRAM = DRAM[T](N)

Accel {
  val S = SRAM[T](N, N)           // Input
  val fifo1 = FIFO[T](2)          // Intermediate FIFOs
  val fifoOUT = FIFO[T](N)        // Output

  S load sDRAM                    // Load the input

  Stream {
    // Compute S'=exp(S)
    Foreach(0 until N, 0 until N) { (i, j) =>
      val input = S(i, j)        // Read the input
      val output = exp(input)    // Do the computation (exp)
      fifo1.enq(output)          // Enqueue the output
    }

    // Compute Rowsum(S')
    Foreach(0 until N) { i =>
      val accum = Reg[T]
      Reduce(accum)(0 until N) { j =>
        fifo1.deq()              // Dequeue the input
      } {_ + _}                  // Accumulate row-wise
      fifoOUT.enq(accum.value)   // Enqueue the output
    }
  }
  outDRAM1 store fifoOUT         // Store the output
}
```

Load input to the on-chip memory

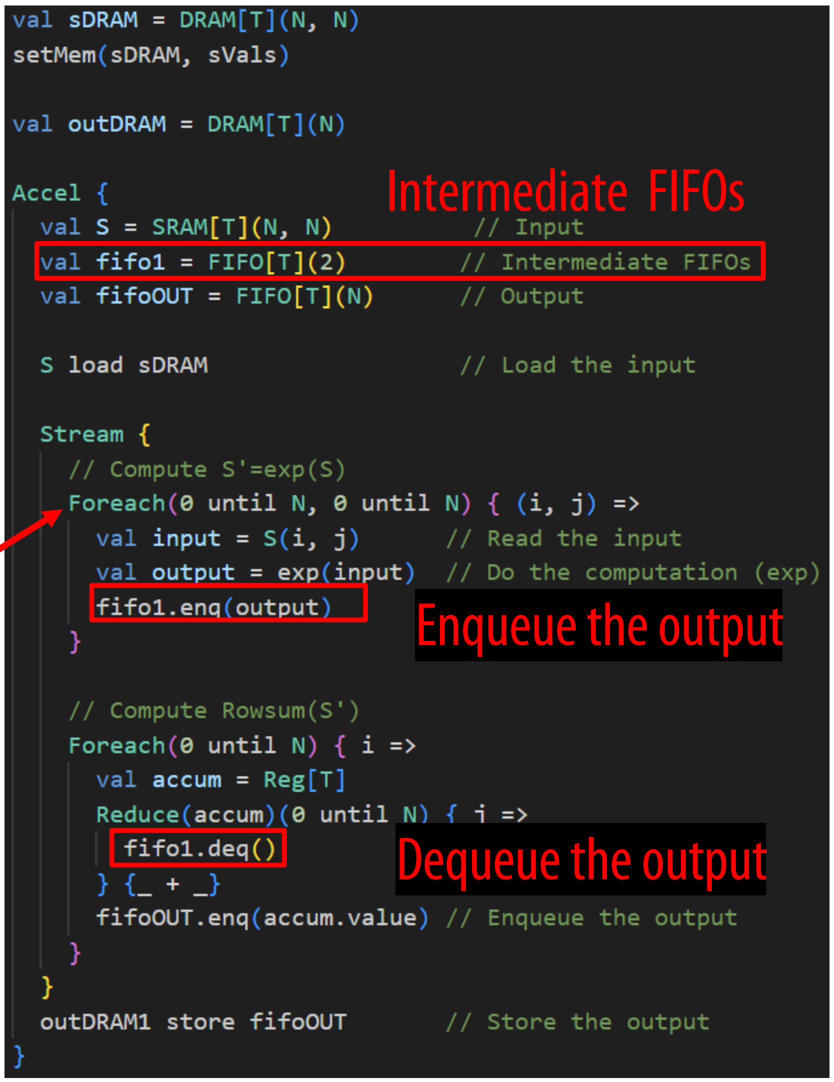
Streaming execution Model

An example program in a streaming execution model

- **Computation: Exponential & Rowsum**



Doing the computation piece-wise

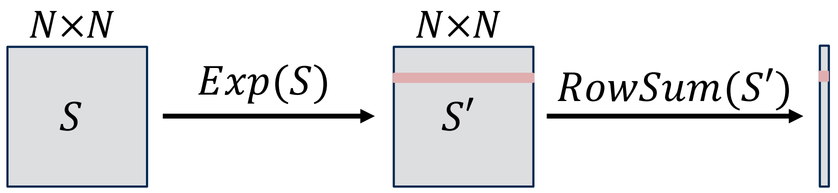


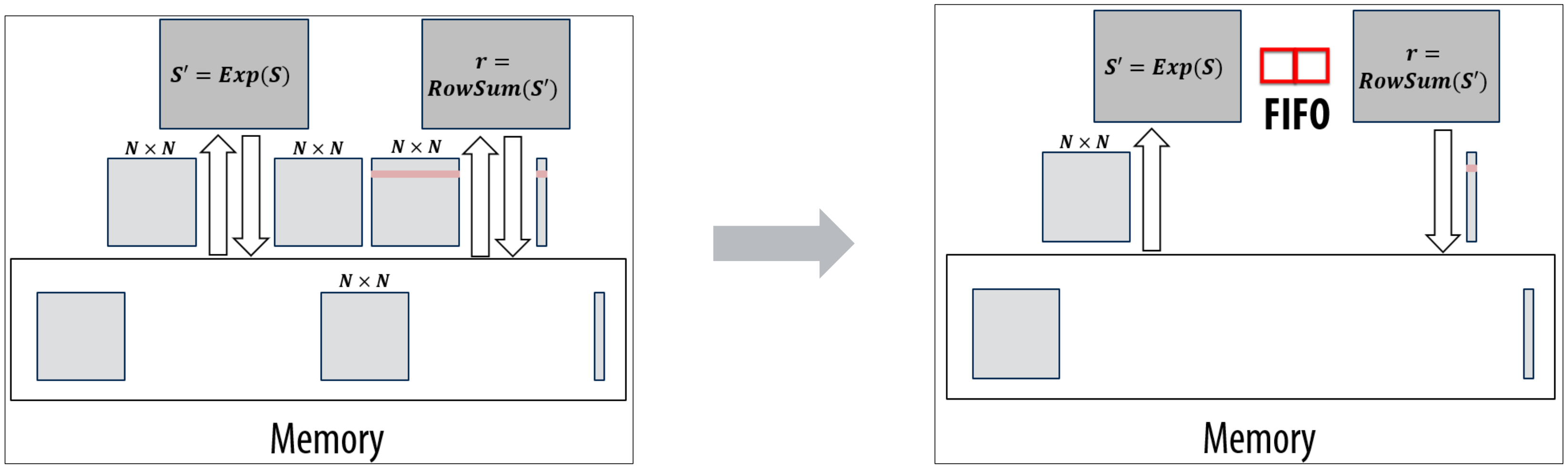
```
val sDRAM = DRAM[T](N, N)
setMem(sDRAM, sVals)

val outDRAM = DRAM[T](N)

Accel {
  val S = SRAM[T](N, N)          // Input
  val fifo1 = FIFO[T](2)         // Intermediate FIFOs
  val fifoOUT = FIFO[T](N)       // Output

  S load sDRAM                   // Load the input

  Stream {
    // Compute S'=exp(S)
    Foreach(0 until N, 0 until N) { (i, j) =>
      val input = S(i, j)       // Read the input
      val output = exp(input)   // Do the computation (exp)
      fifo1.enq(output)
    }

    // Compute Rowsum(S')
    Foreach(0 until N) { i =>
      val accum = Reg[T]
      Reduce(accum)(0 until N) { j =>
        fifo1.deq()
      } {_ + _}
      fifoOUT.enq(accum.value) // Enqueue the output
    }
  }
  outDRAM1 store fifoOUT        // Store the output
}
```

Streaming Execution Model Summary

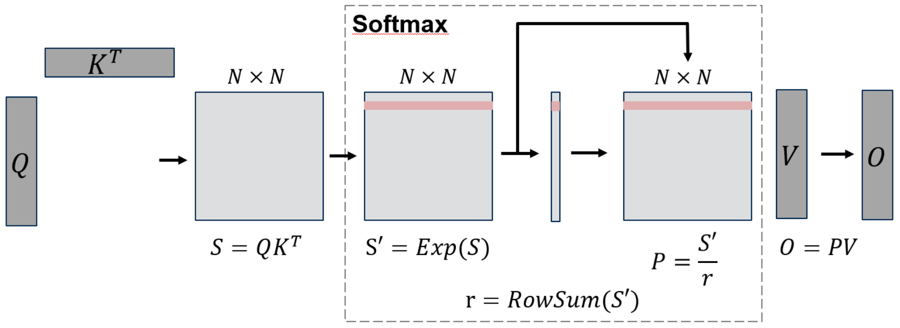
Example using a streaming execution model

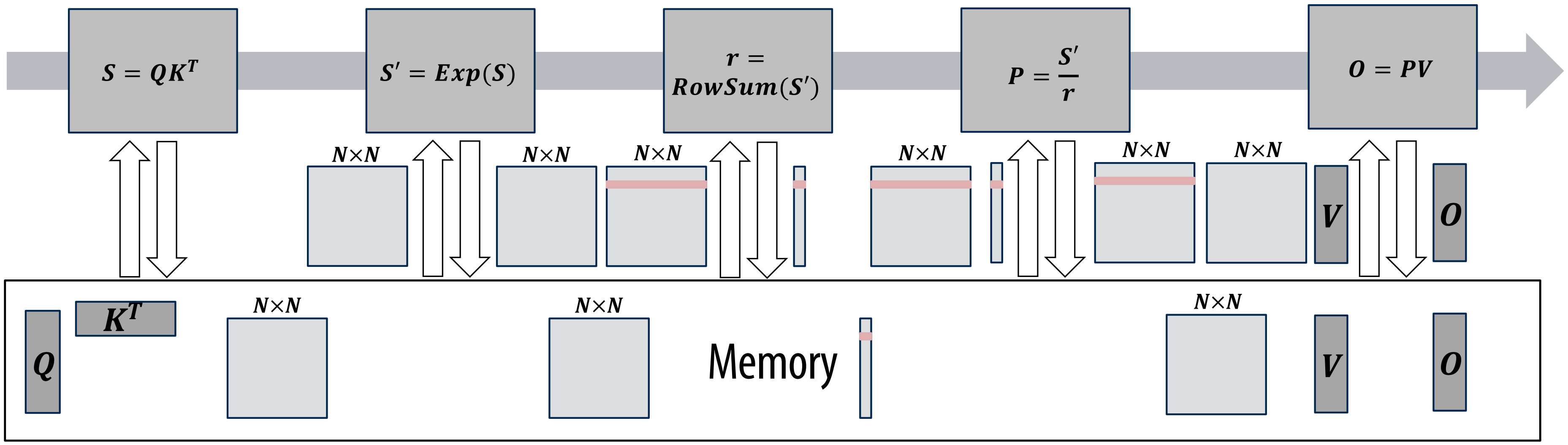
- **Computation: Exponential & Rowsum**



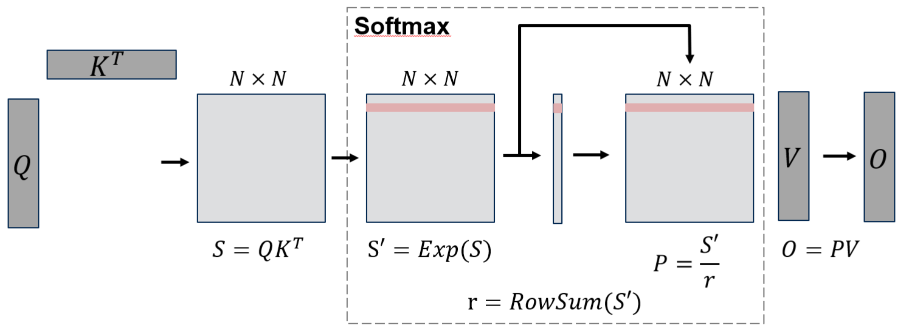


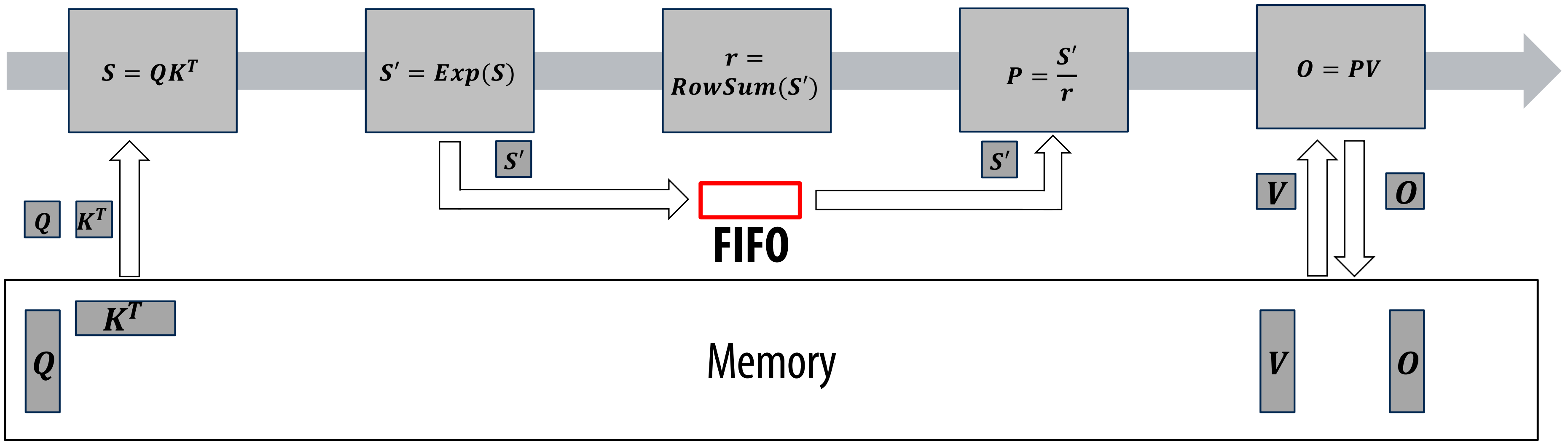
Kernel-based Execution Model (Overview)



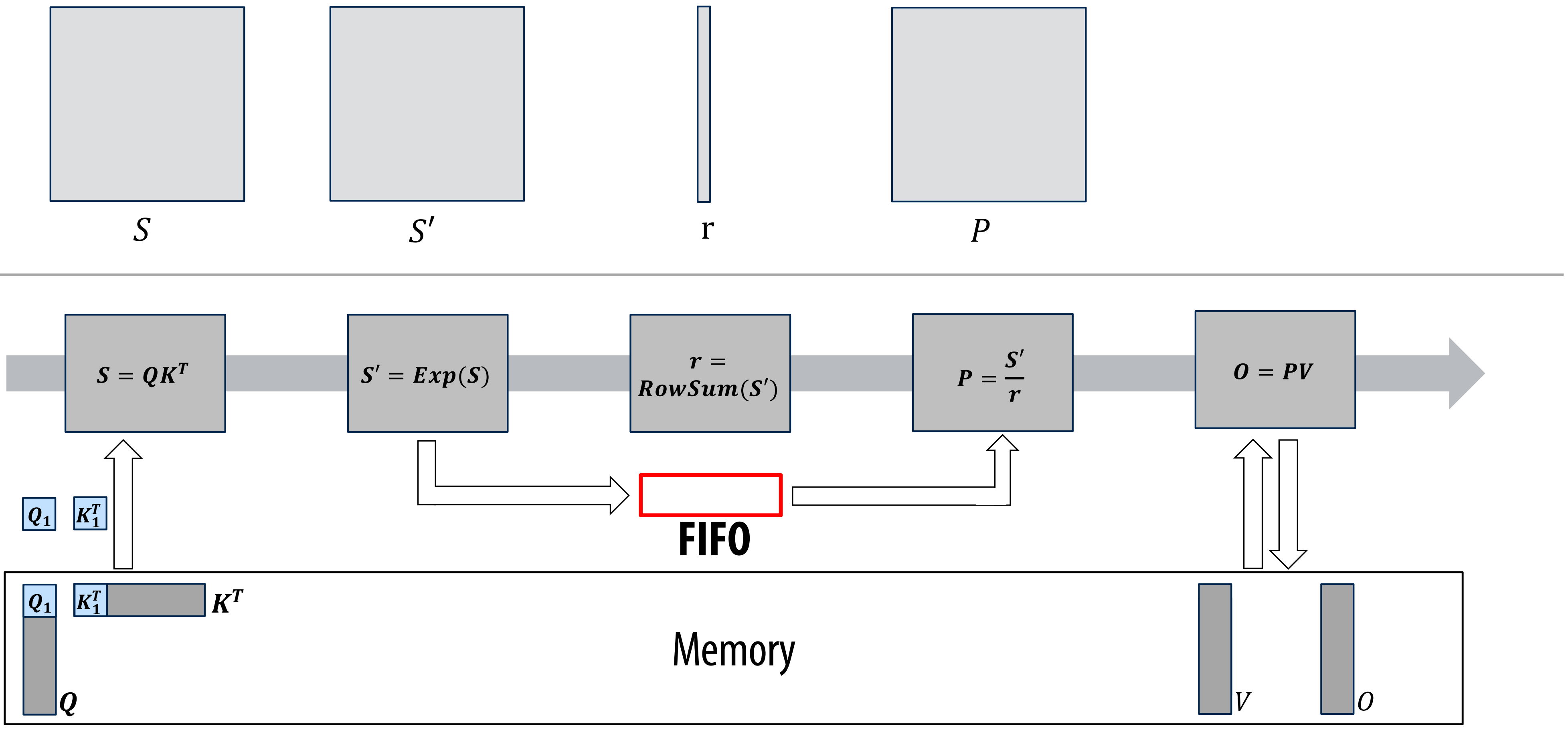


Streaming Execution Model (Overview)

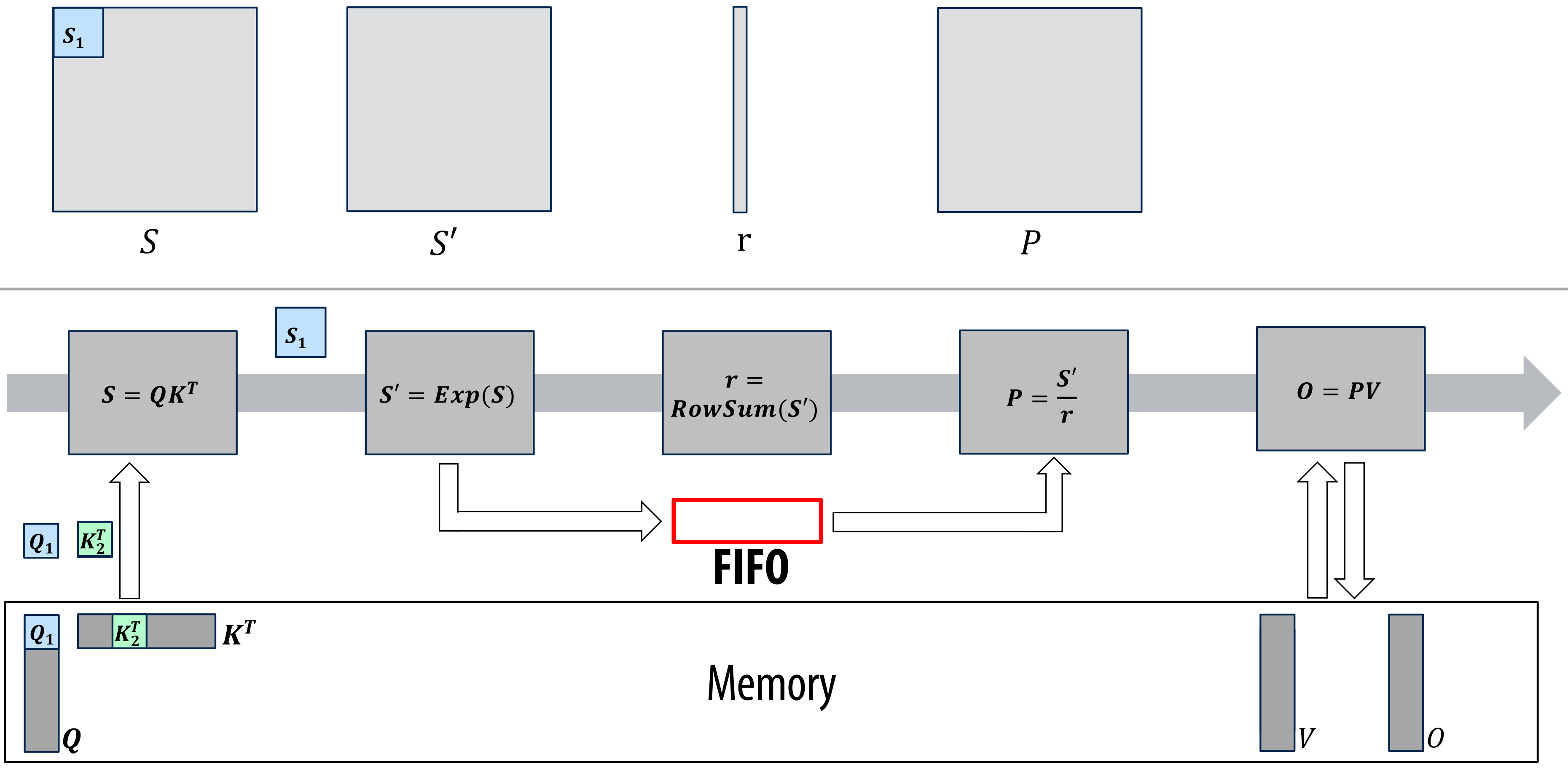




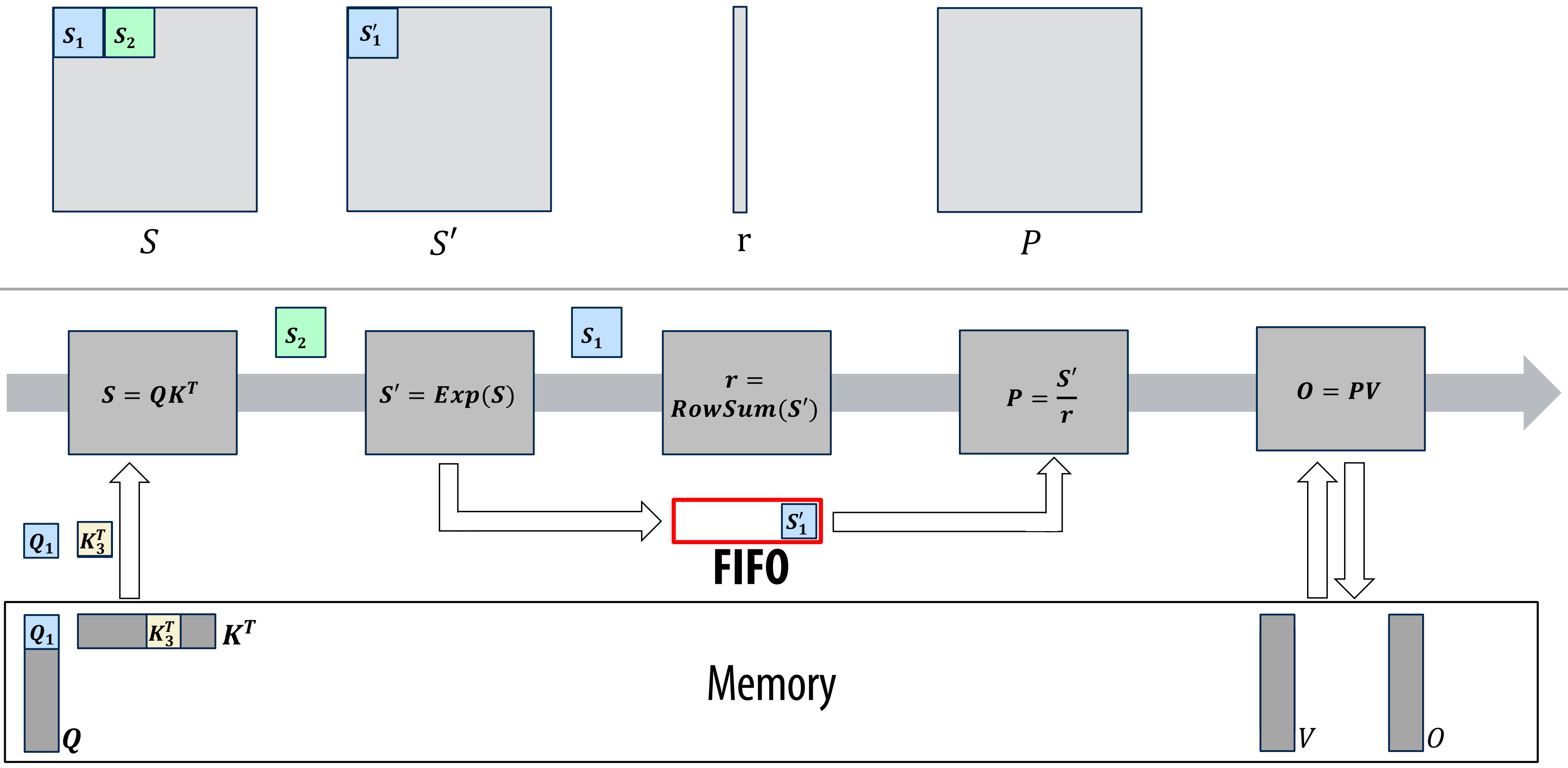
Streaming Execution Model



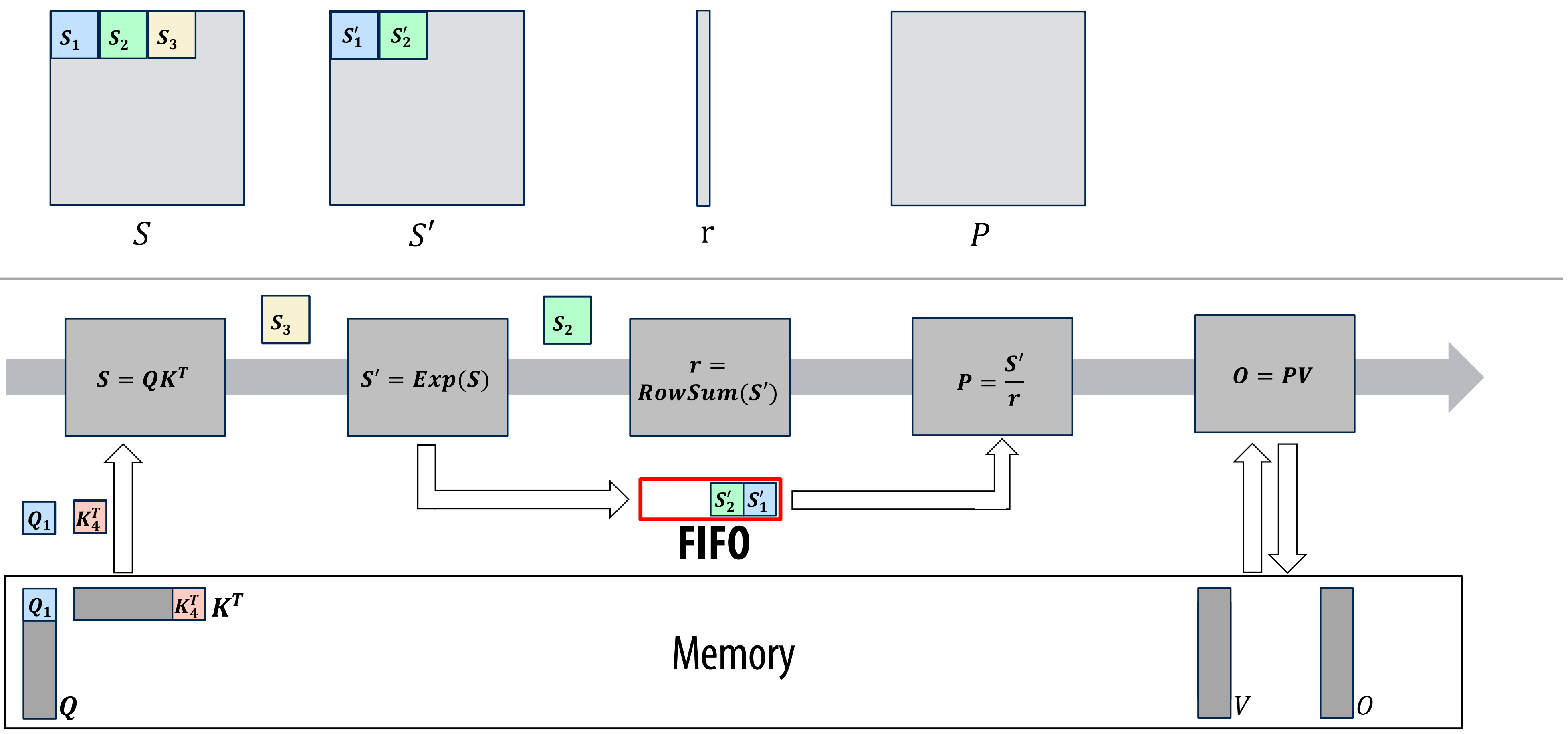
Streaming Execution Model



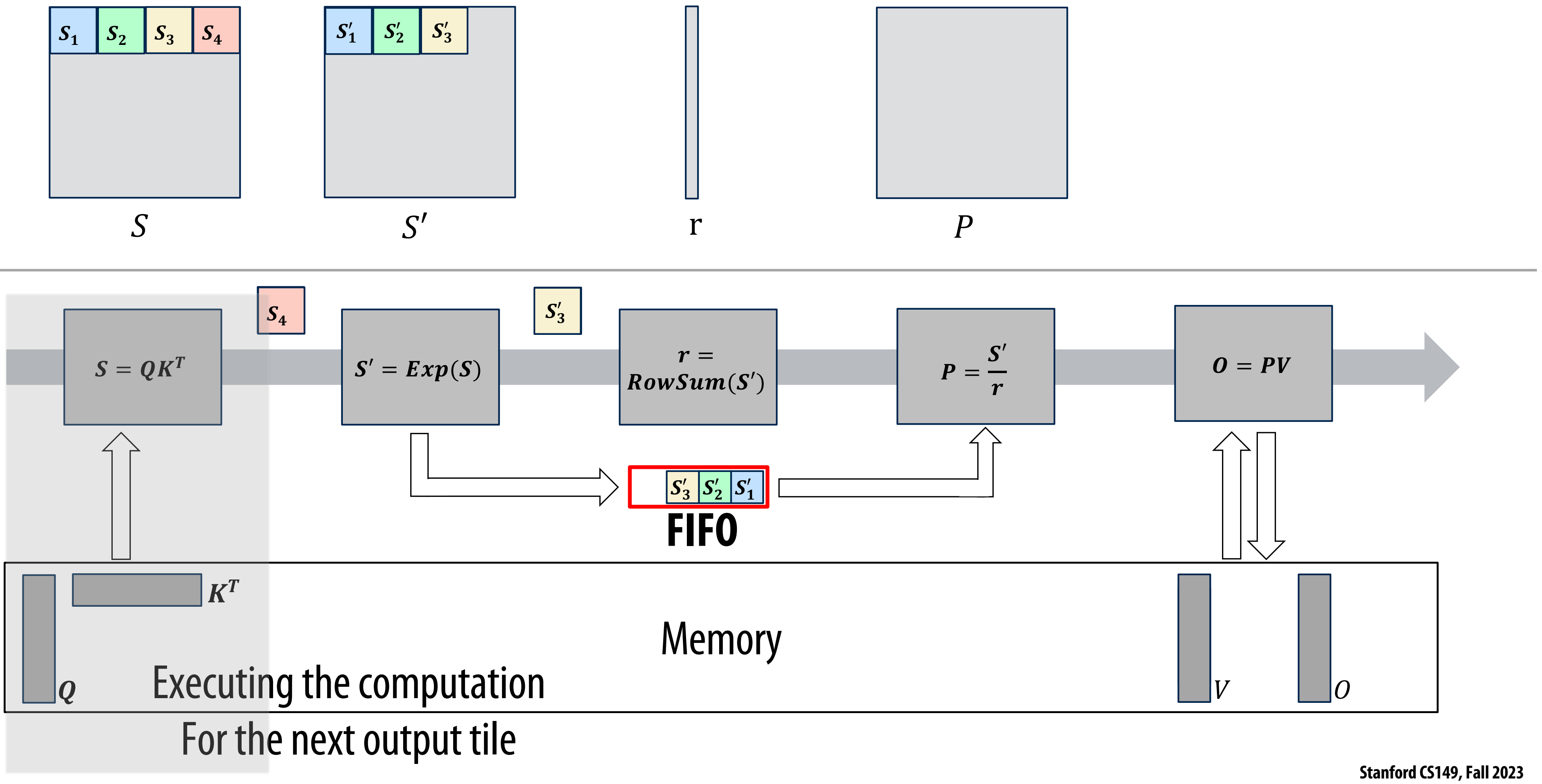
Streaming Execution Model



Streaming Execution Model



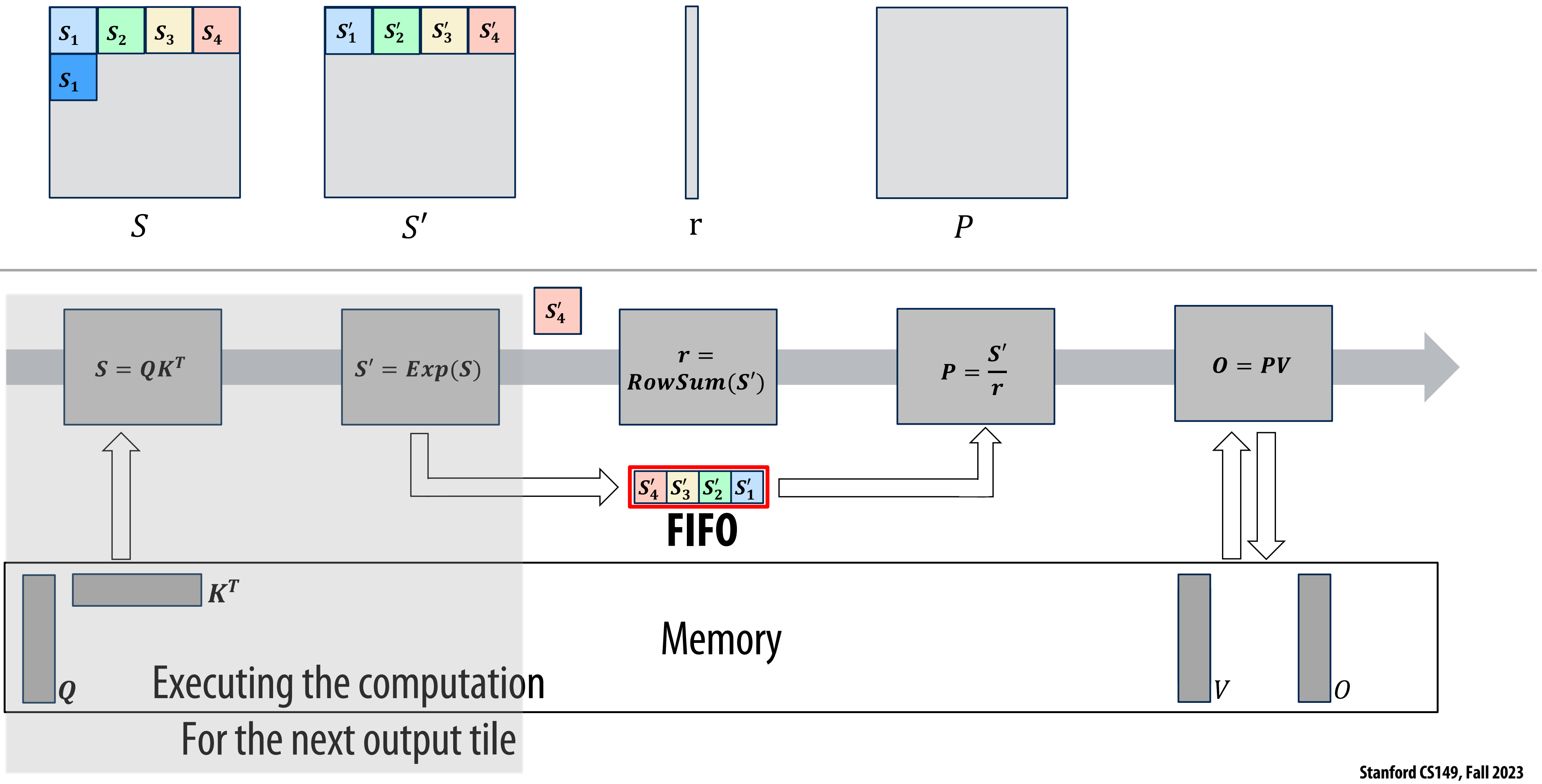
Streaming Execution Model



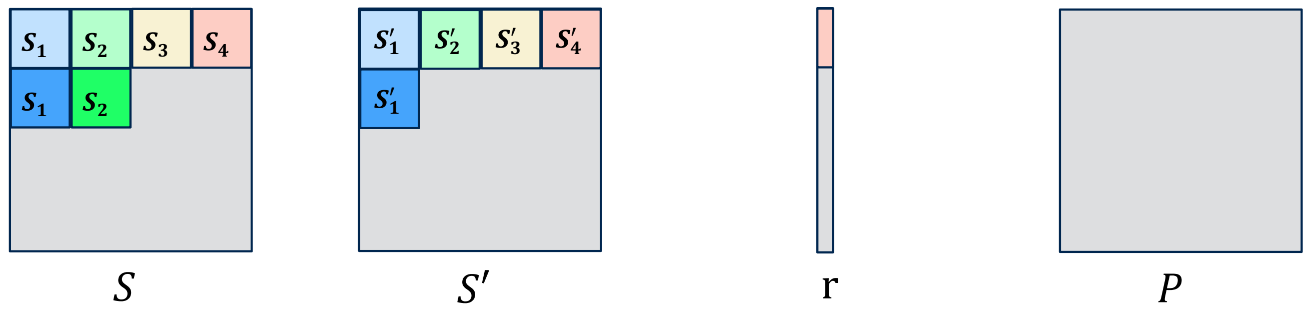
Executing the computation

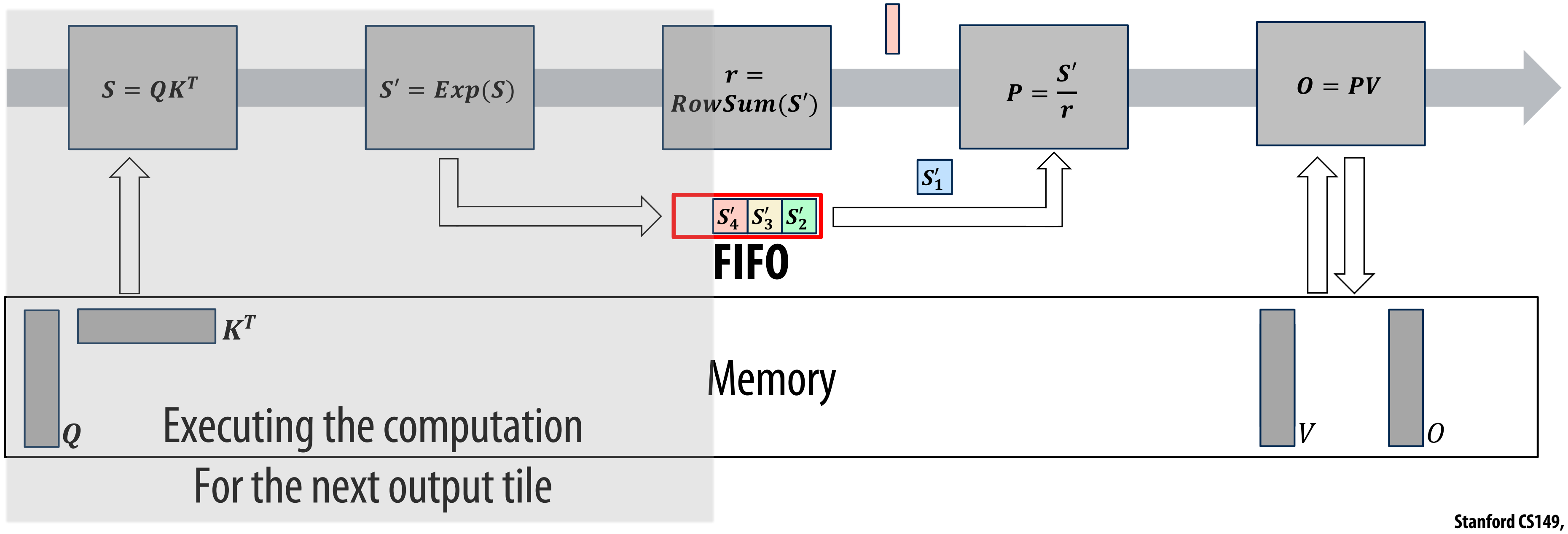
For the next output tile

Streaming Execution Model

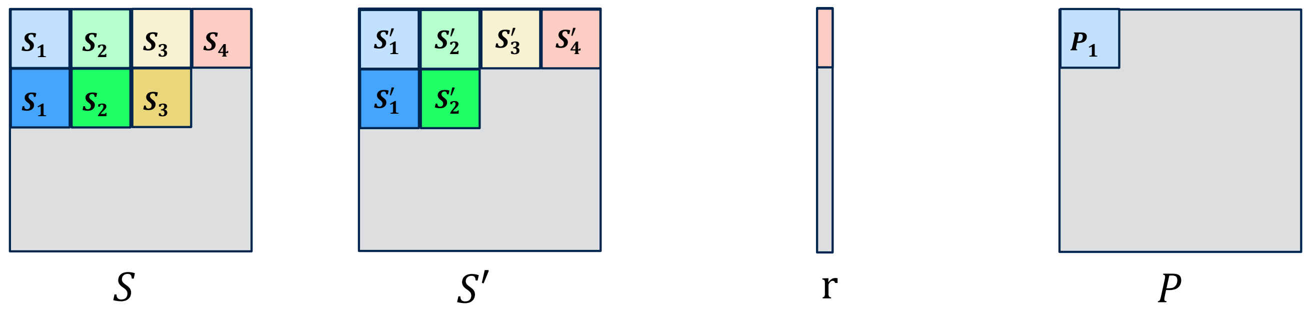


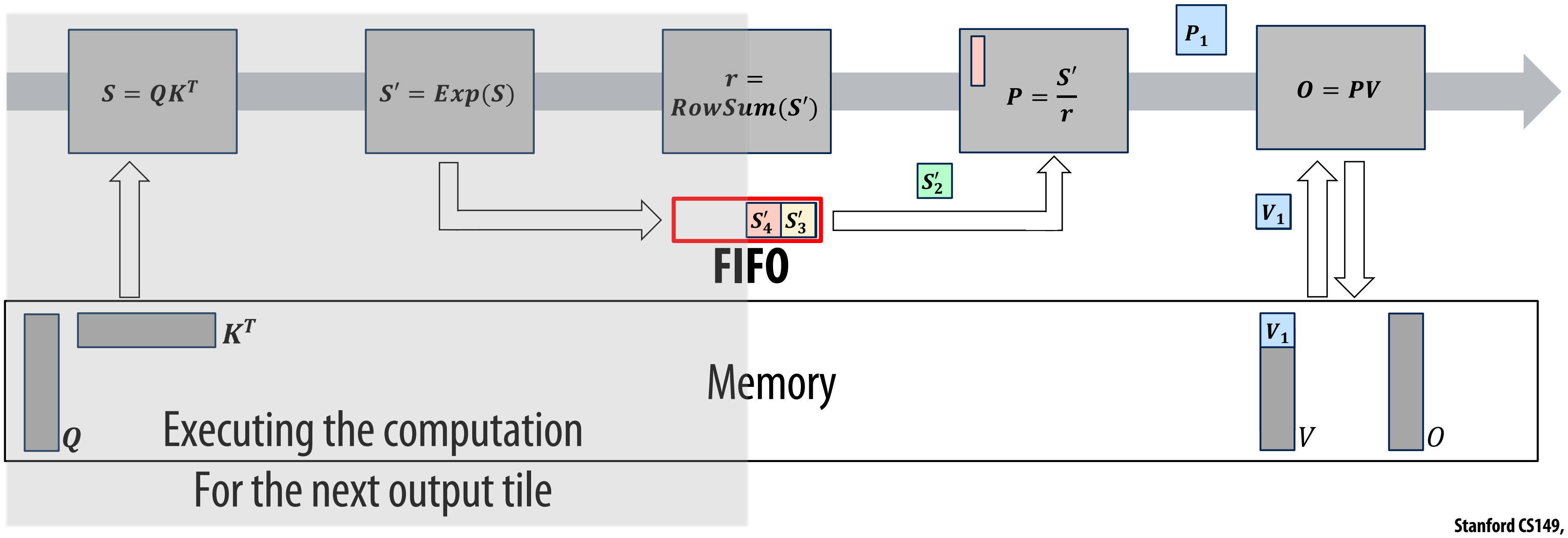
Streaming Execution Model





Streaming Execution Model

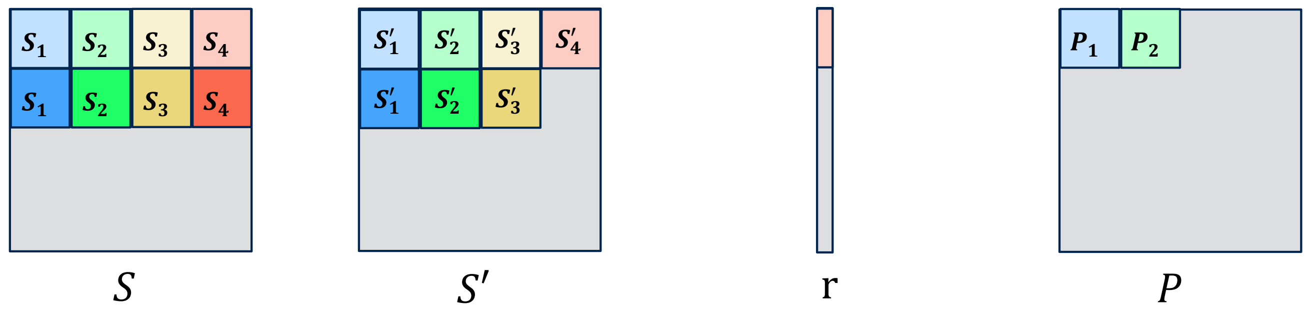


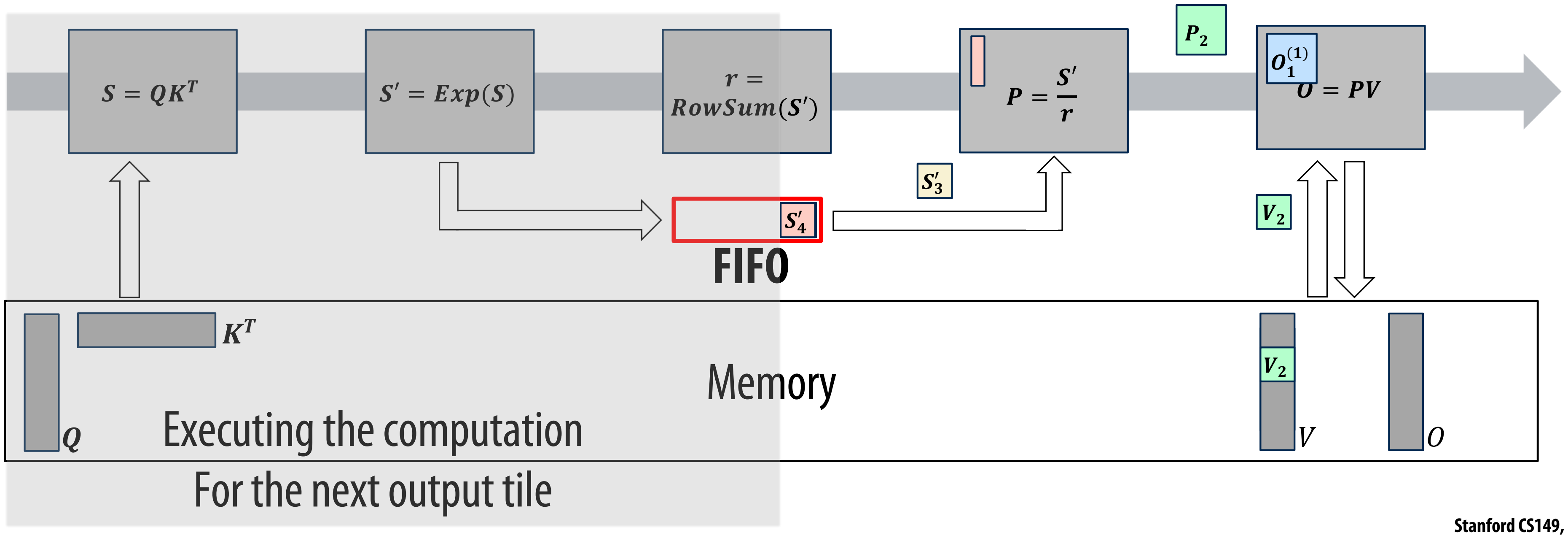


Executing the computation

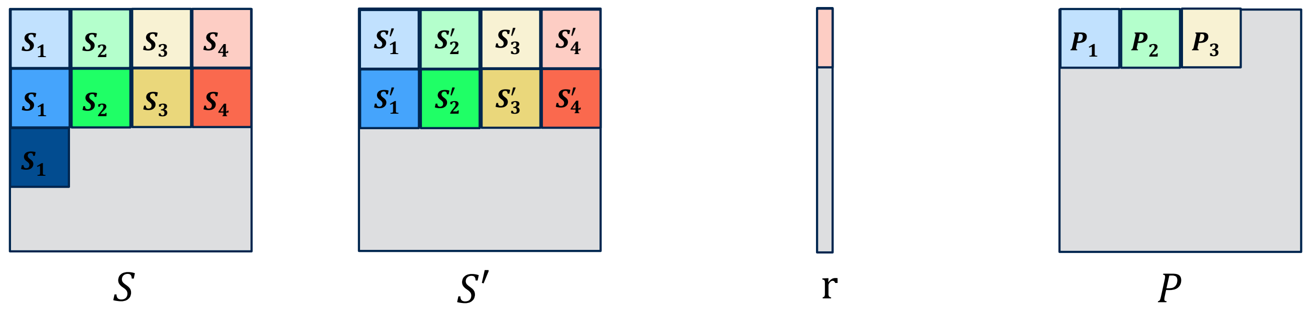
For the next output tile

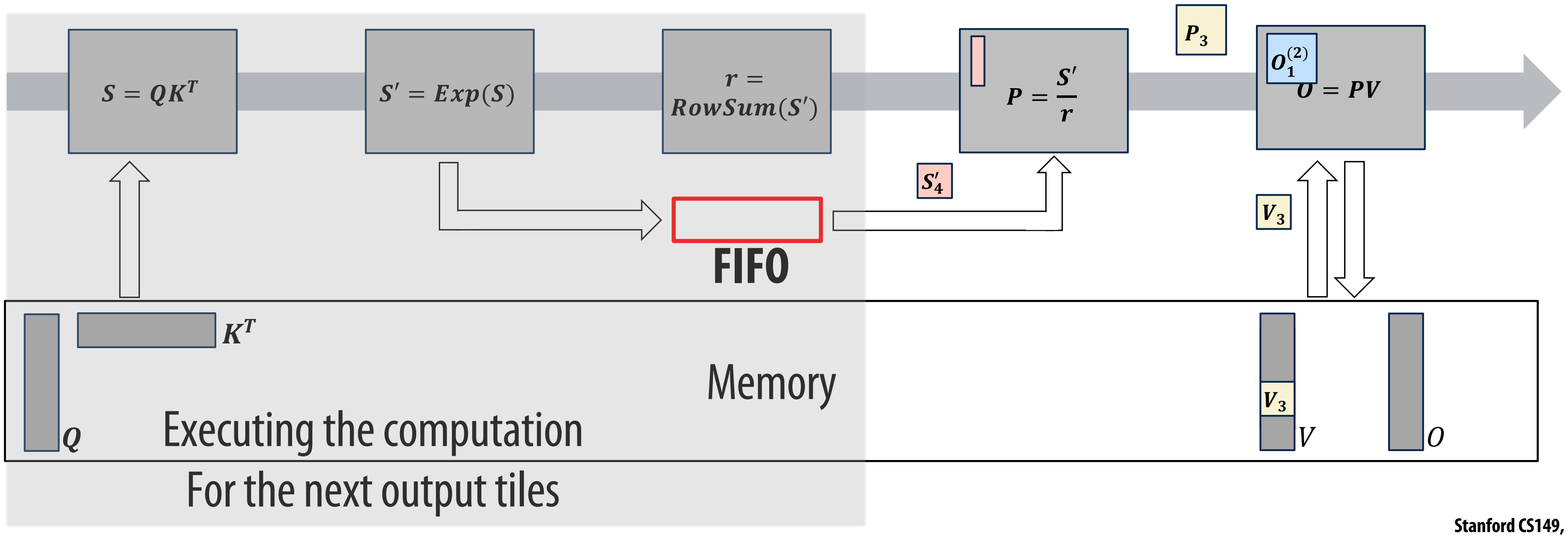
Streaming Execution Model



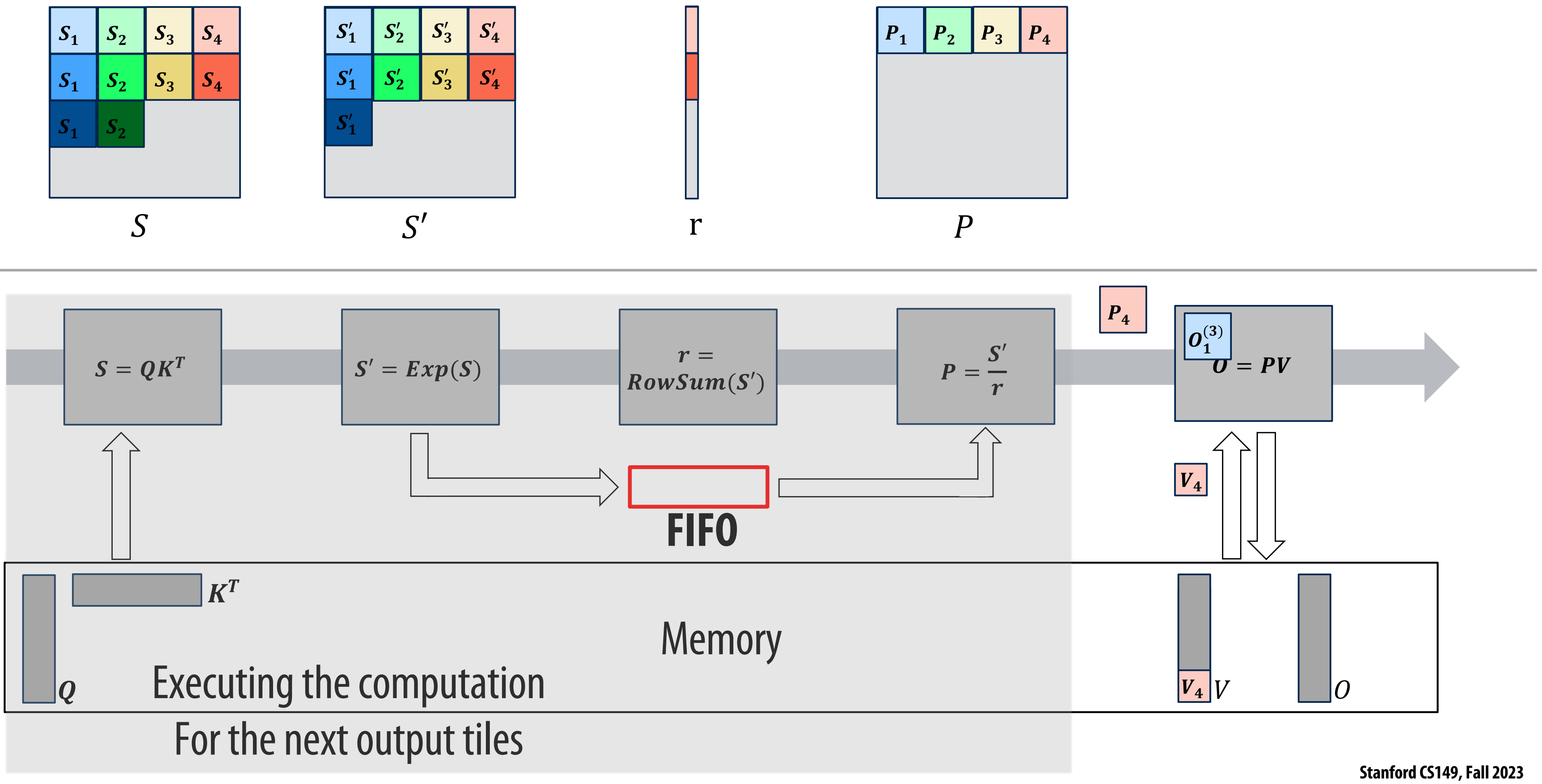


Streaming Execution Model

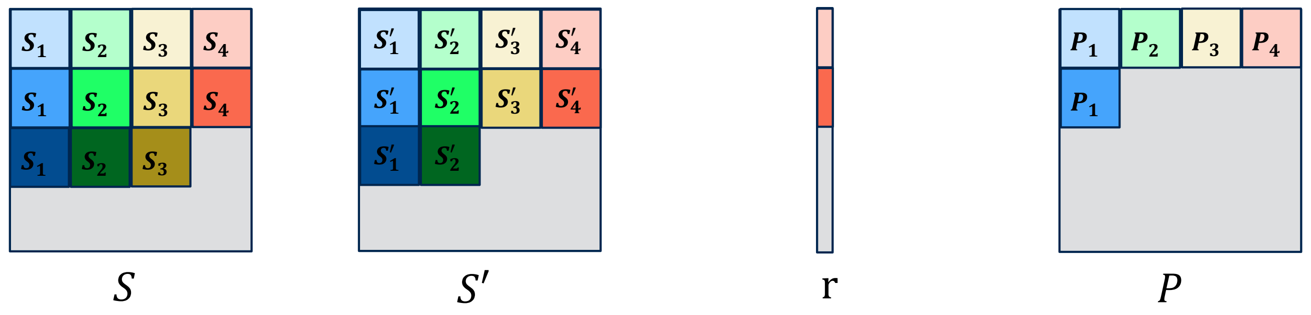


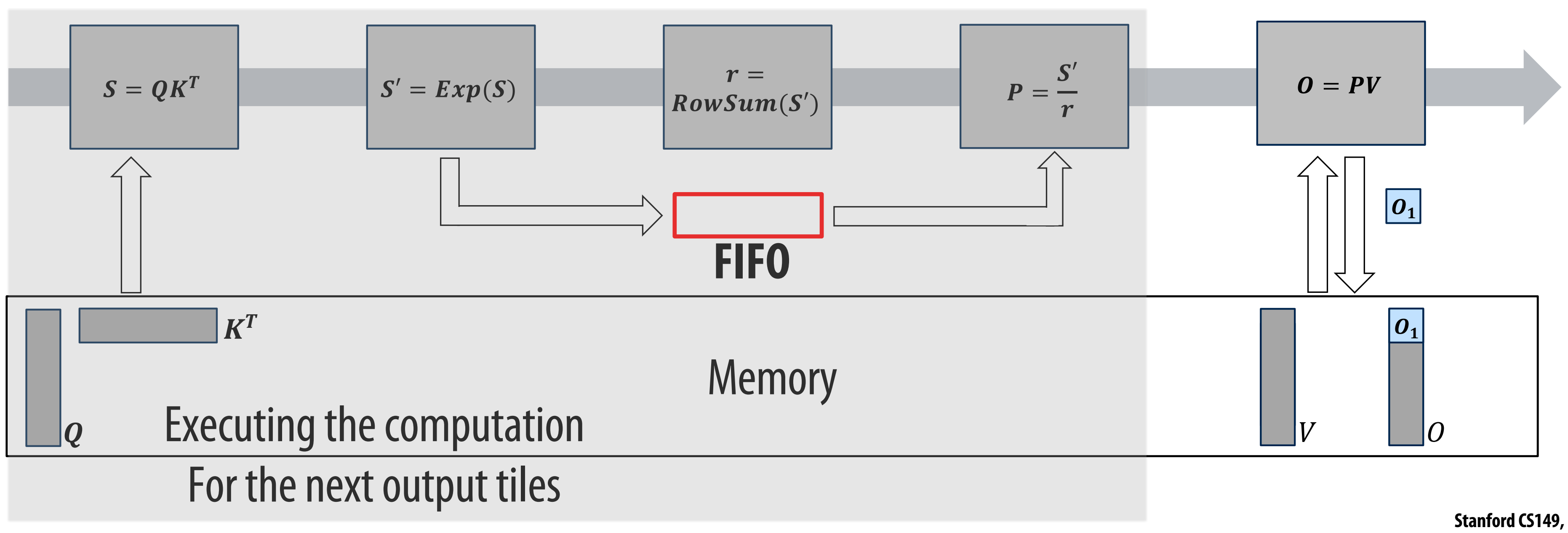


Streaming Execution Model



Streaming Execution Model



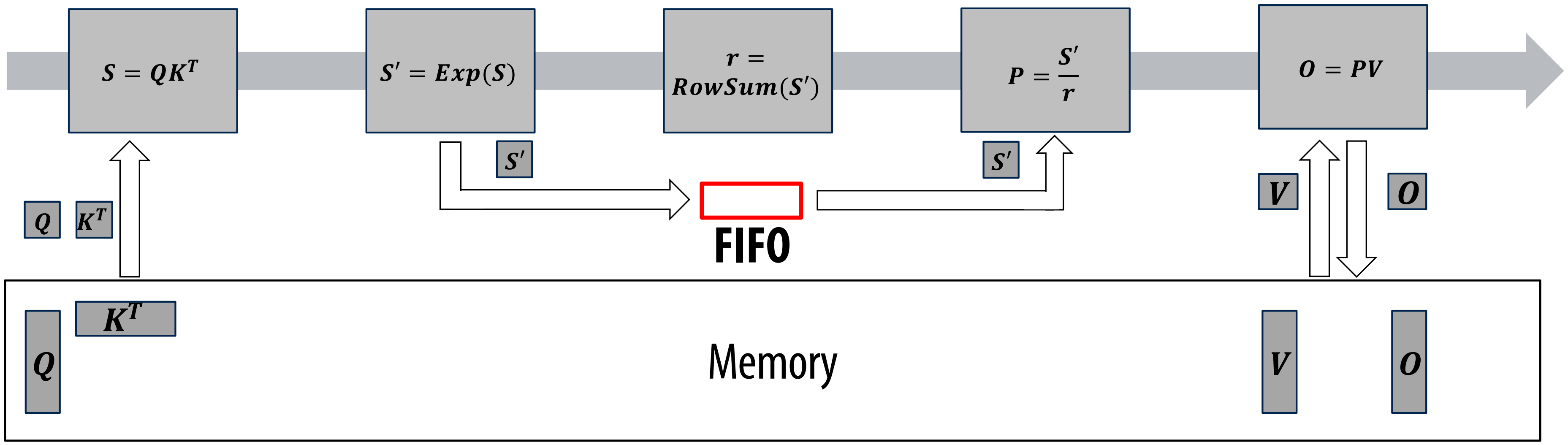


Executing the computation

For the next output tiles

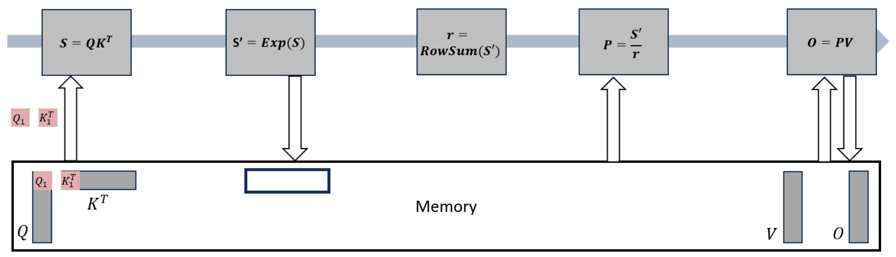
Can we do better with FlashAttention?

- Yes!
- By paying a bit more computation cost as Flash Attention does, we can eliminate the FIFO in the middle

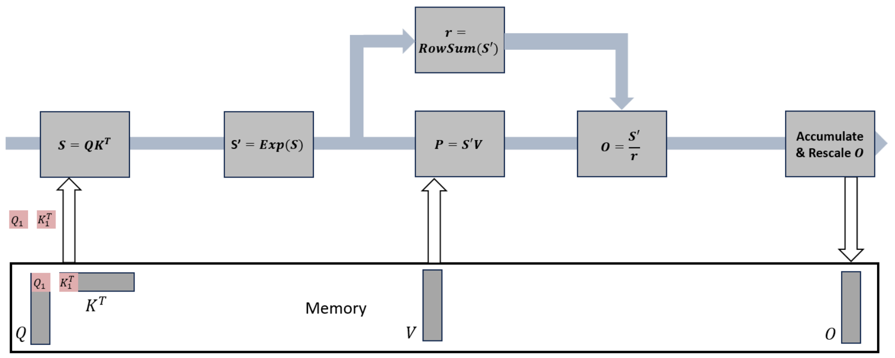


Can we do better with FlashAttention?

- We needed this sequence-length (N) sized FIFO to buffer the output of $S' = Exp(S)$.
- This is because in softmax, we have to wait until the row-wise reduction (row sum) is calculated to divide the output of $S' = Exp(S)$ with the row sum.
- Flash Attention breaks this dependency by:
  - Reordering operations
  - Using a running sum & rescaling instead of the naïve reduction (row sum)



W/O Flash Attention



W/ Flash Attention

Kernel versus Stream Execution

- **More parallelism**
  - **FlashAttention with kernel-based execution model:**
    - Cannot overlap the computation for different output tiles
  - **Streaming execution model**
    - Spatially maps each computation with pipeline communication
    - **Can overlap (pipeline) the computation for different output tiles!**
- **Don't have to manually create fused kernels**
  - **FlashAttention with kernel-based execution model:**
    - Have to manually write fused kernels in CUDA
    - Often challenging to fuse deeply due to the limit in (# of registers / SM)
  - **Streaming execution model**
    - Operations gets fused automatically if we write the program using FIFOs
    - Compiler can automatically generate fused executioin

Accelerator Design Summary

- **Significant energy efficiency improvements from specialized accelerators (100x–1000x)**
- **Designing an accelerator is a tradeoff between performance and resource utilization**
  - **Parallelism**
  - **Locality**
- **It requires the programmer to have insight into the application**
  - **Where is the bottleneck**
  - **Is the implementation compute or memory-bound**
- **Spatial helps you understand the trade-off between performance and resource utilization**
  - **Allows rapid exploration of your algorithm**
  - **Enables high-level accelerator design**

Reducing energy consumption idea 1: use specialized processing

(use the right processor for the job)

Reducing energy consumption idea 2: move less data

Data movement has high energy cost

- Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory
  - Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption
- "Ballpark" numbers [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

Suggests that recomputing values, rather than storing and reloading them, is a better answer when optimizing code for energy efficiency!

- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

Moving data is costly!

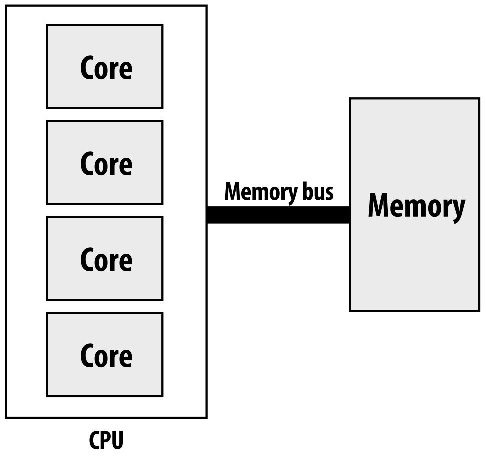
Data movement limits performance

Many processing elements…

= higher overall rate of memory requests

= need for more memory bandwidth

(result: bandwidth-limited execution)

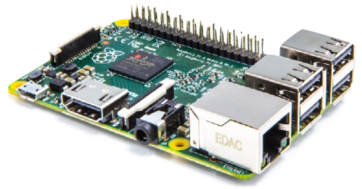


Data movement has high energy cost

~ 0.9 pJ for a 32-bit floating-point math op *

~ 5 pJ for a local SRAM (on chip) data access

~ 640 pJ to load 32 bits from LPDDR memory

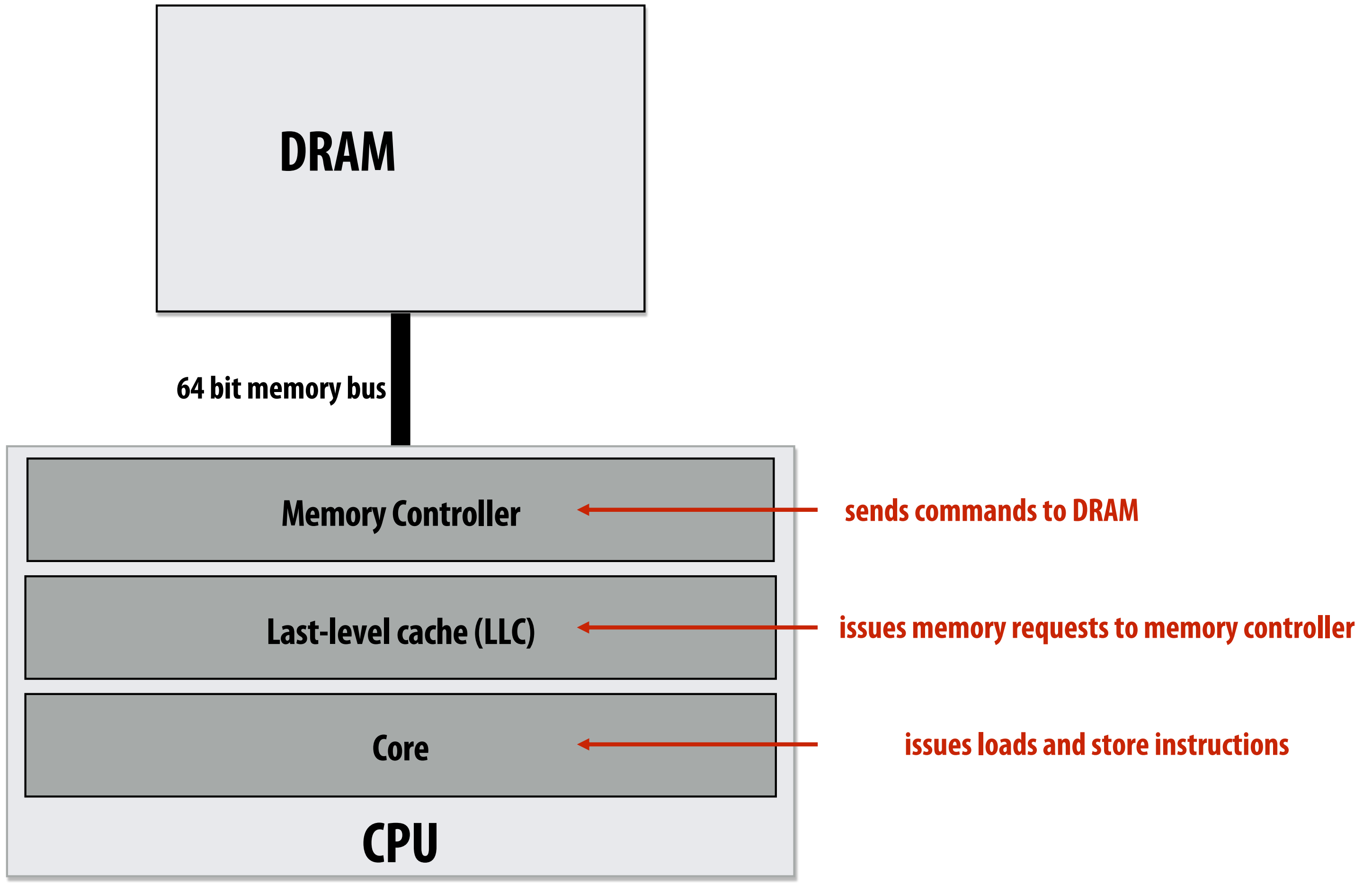


* Source: [Han, ICLR 2016], 45 nm CMOS assumption

Accessing DRAM

(a basic tutorial on how DRAM works)

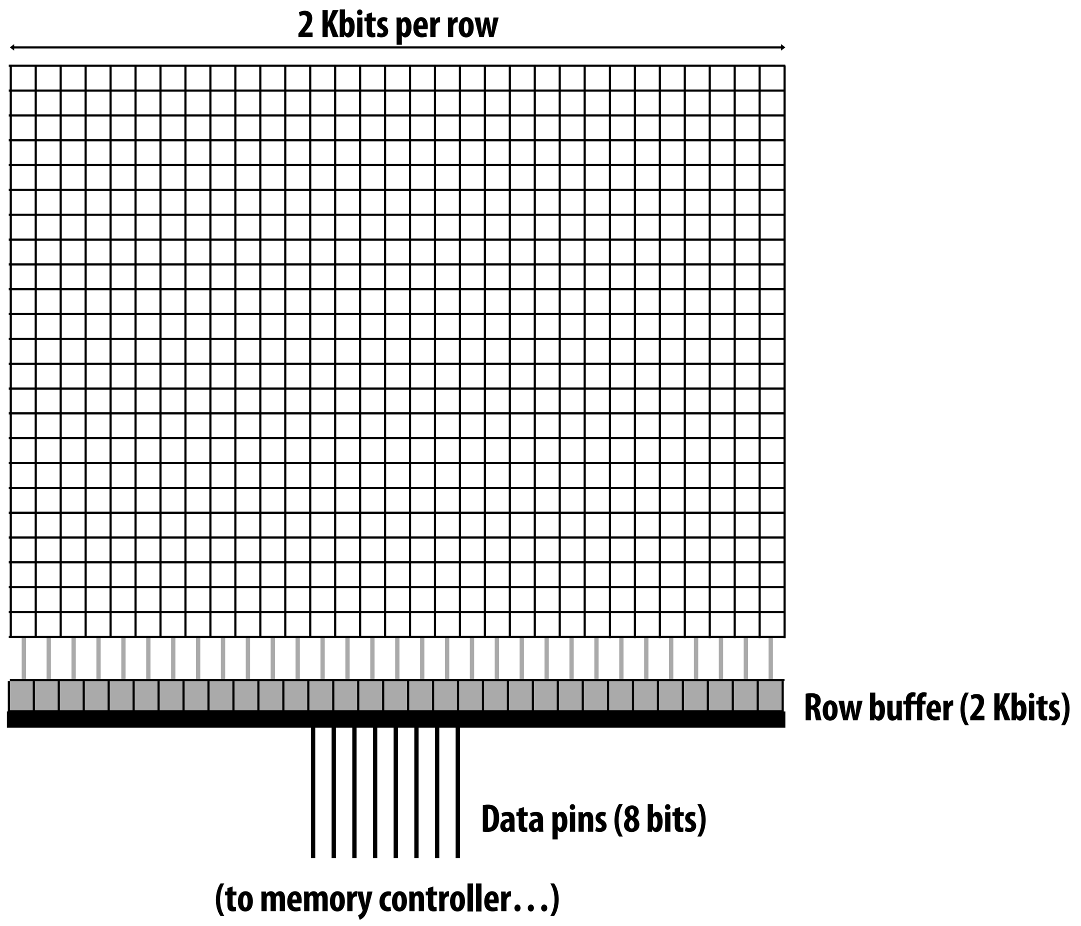
The memory system



DRAM array

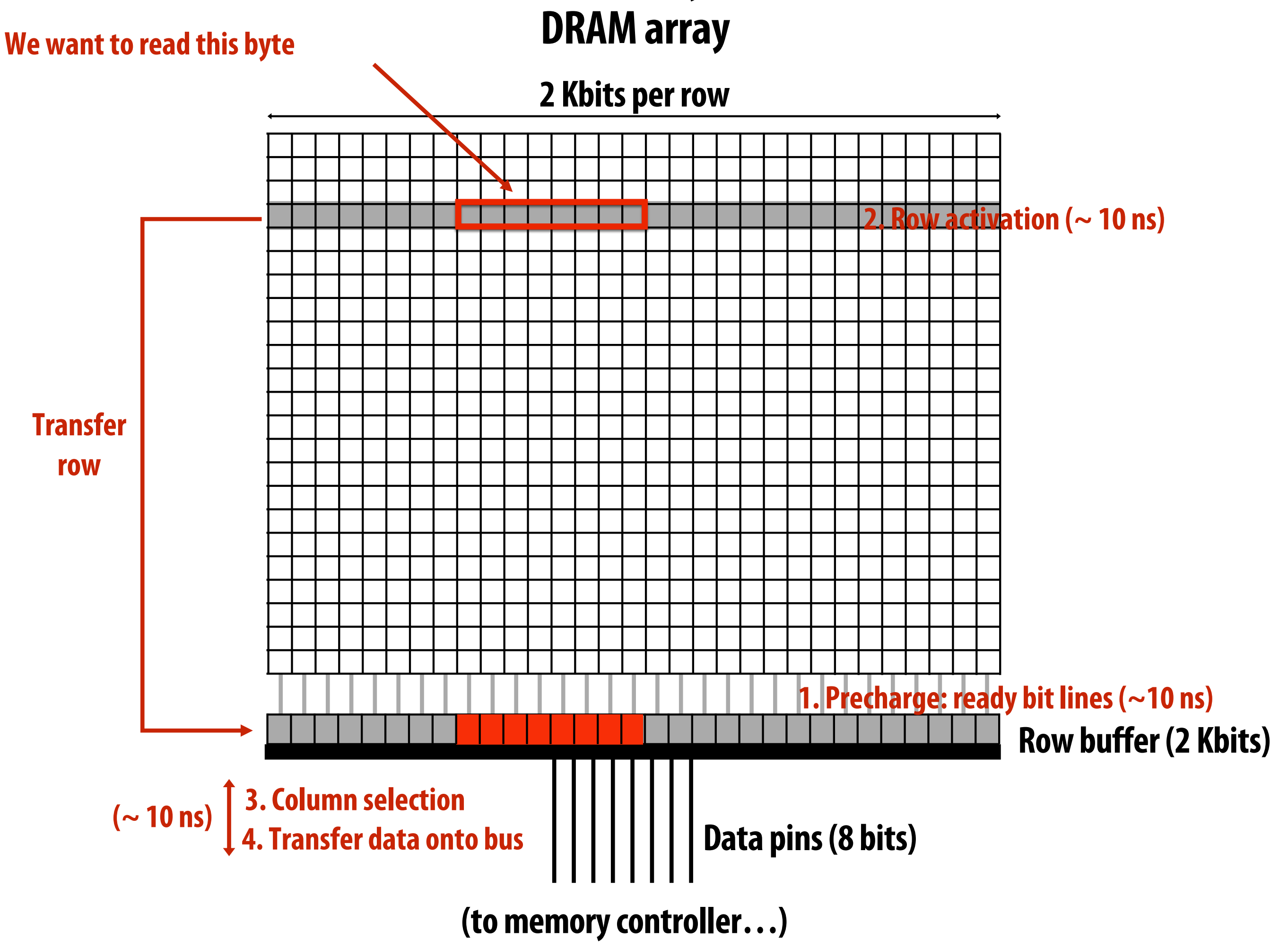
1 transistor + capacitor per "bit"

(Recall: a capacitor stores charge)



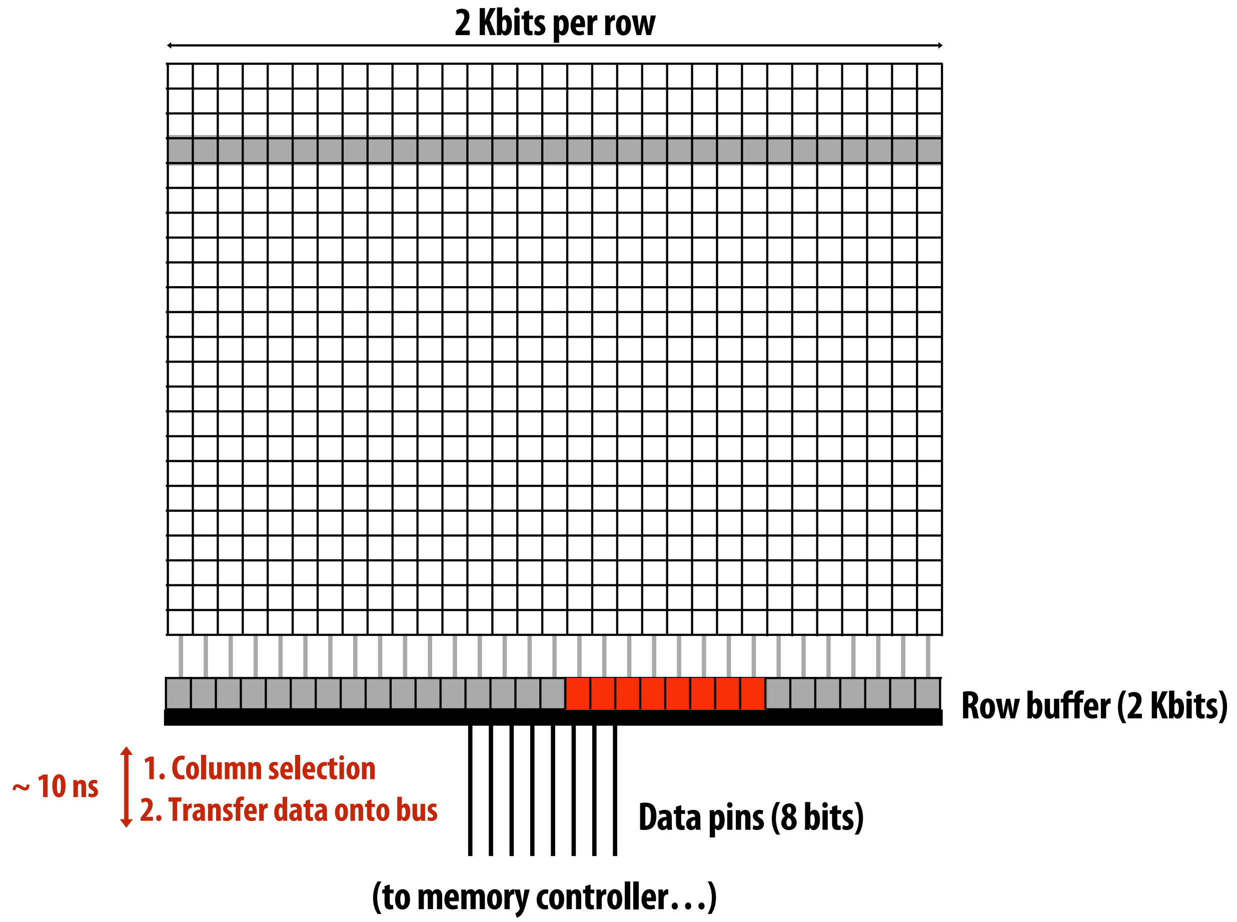
DRAM operation (load one byte)

Estimated latencies are in units of memory clocks: DDR3-1600 (Kayvon's laptop)



Load next byte from (already active) row

Lower latency operation: can skip precharge and row activation steps



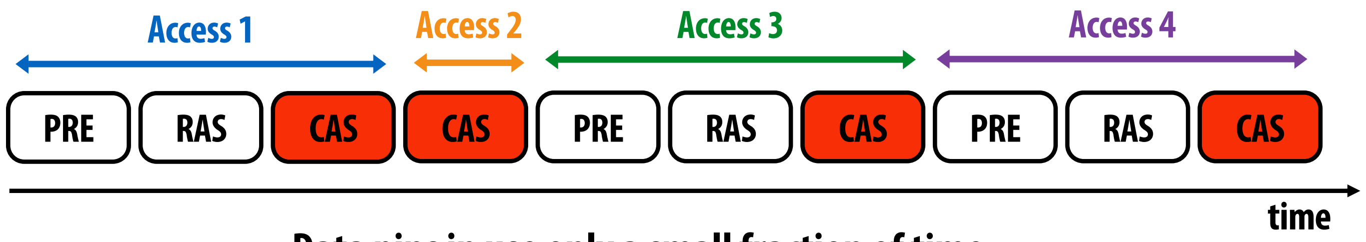
DRAM access latency is not fixed

- **Best case latency: read from active row**
  - Column access time (CAS)
- **Worst case latency: bit lines not ready, read from new row**
  - Precharge (PRE) + row activate (RAS) + column access (CAS)

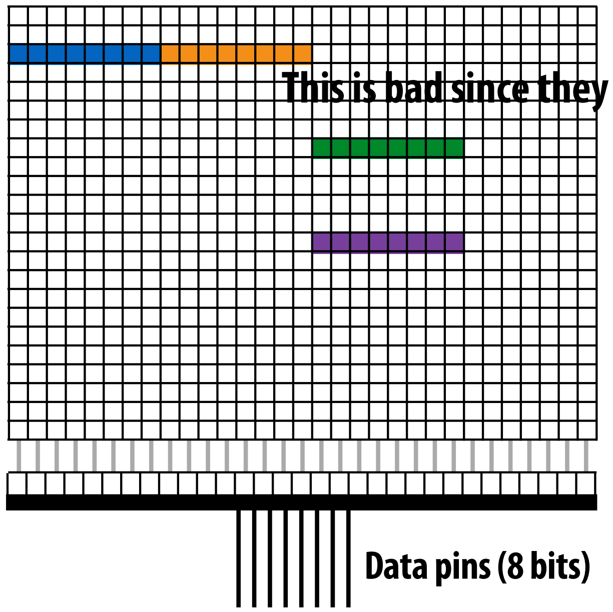
Precharge readies bit lines and writes row buffer contents back into DRAM array (read was destructive)

- **Question 1: when to execute precharge?**
  - After each column access?
  - Only when new row is accessed?
- **Question 2: how to handle latency of DRAM access?**

Problem: low pin utilization due to latency of access

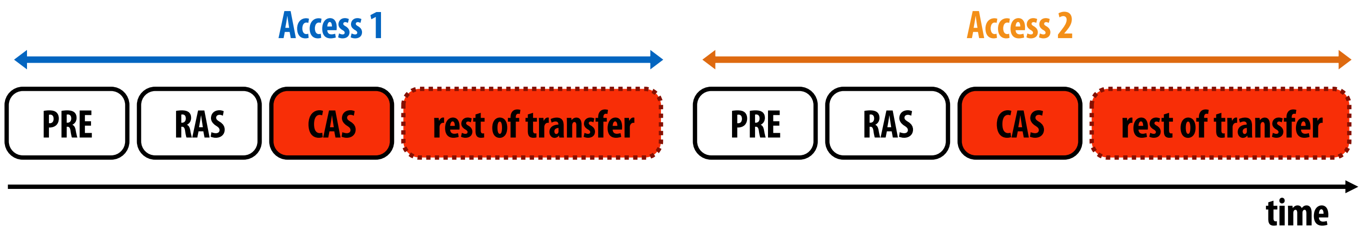


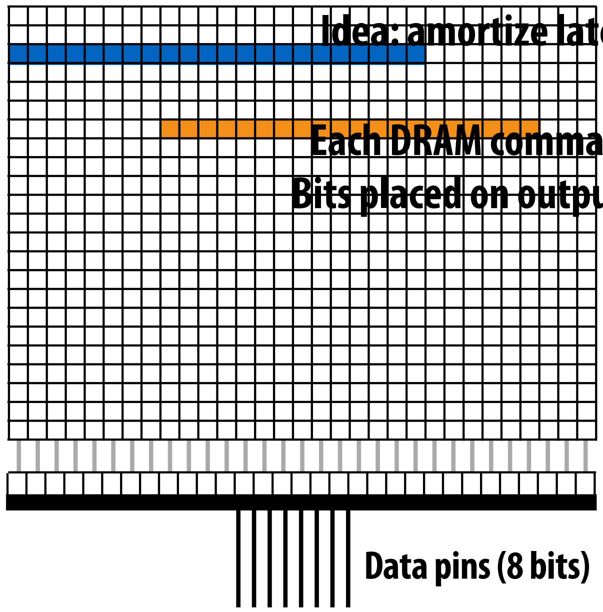
Data pins in use only a small fraction of time
(red = data pins busy)



This is bad since they are the scarcest resource!

DRAM burst mode





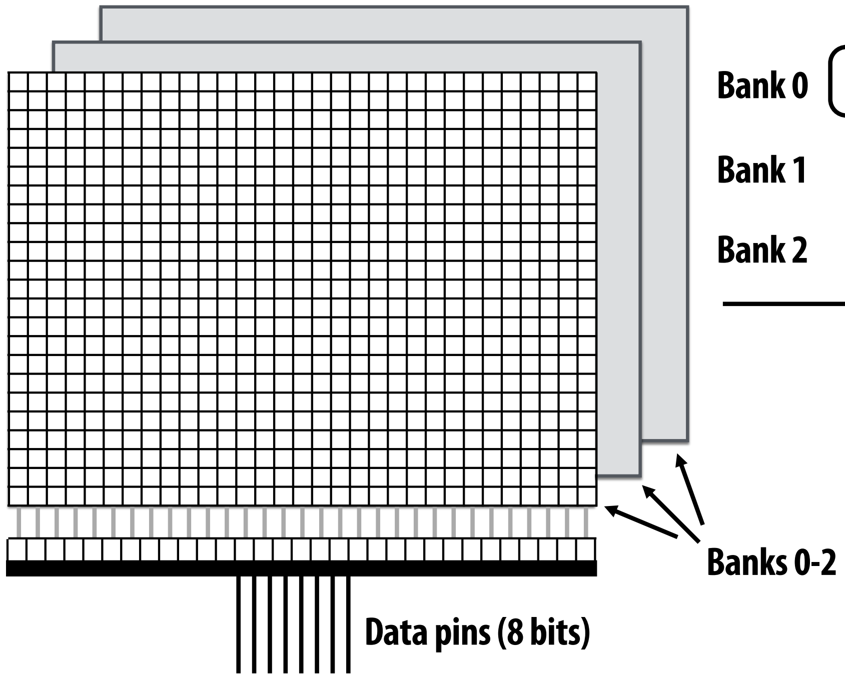
Idea: amortize latency over larger transfers

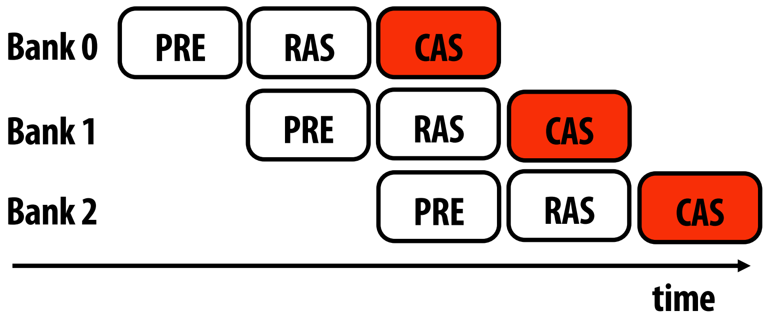
Each DRAM command describes bulk transfer

Bits placed on output pins in consecutive clocks

DRAM chip consists of multiple banks

- All banks share same pins (only one transfer at a time)
- Banks allow for pipelining of memory requests
  - Precharge/activate rows/send column address to one bank while transferring data from another
  - Achieves high data pin utilization

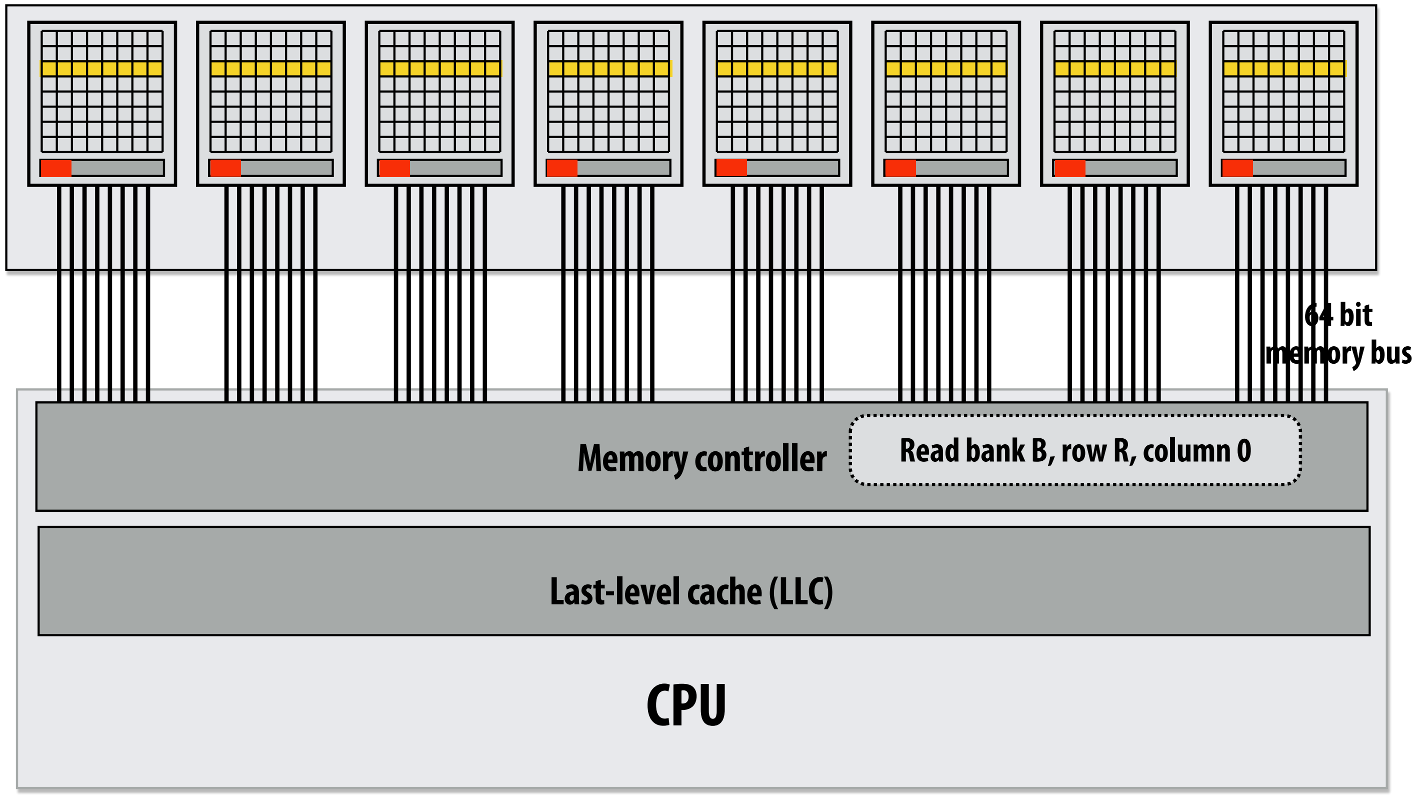




Organize multiple chips into a DIMM

Example: Eight DRAM chips (64-bit memory bus)

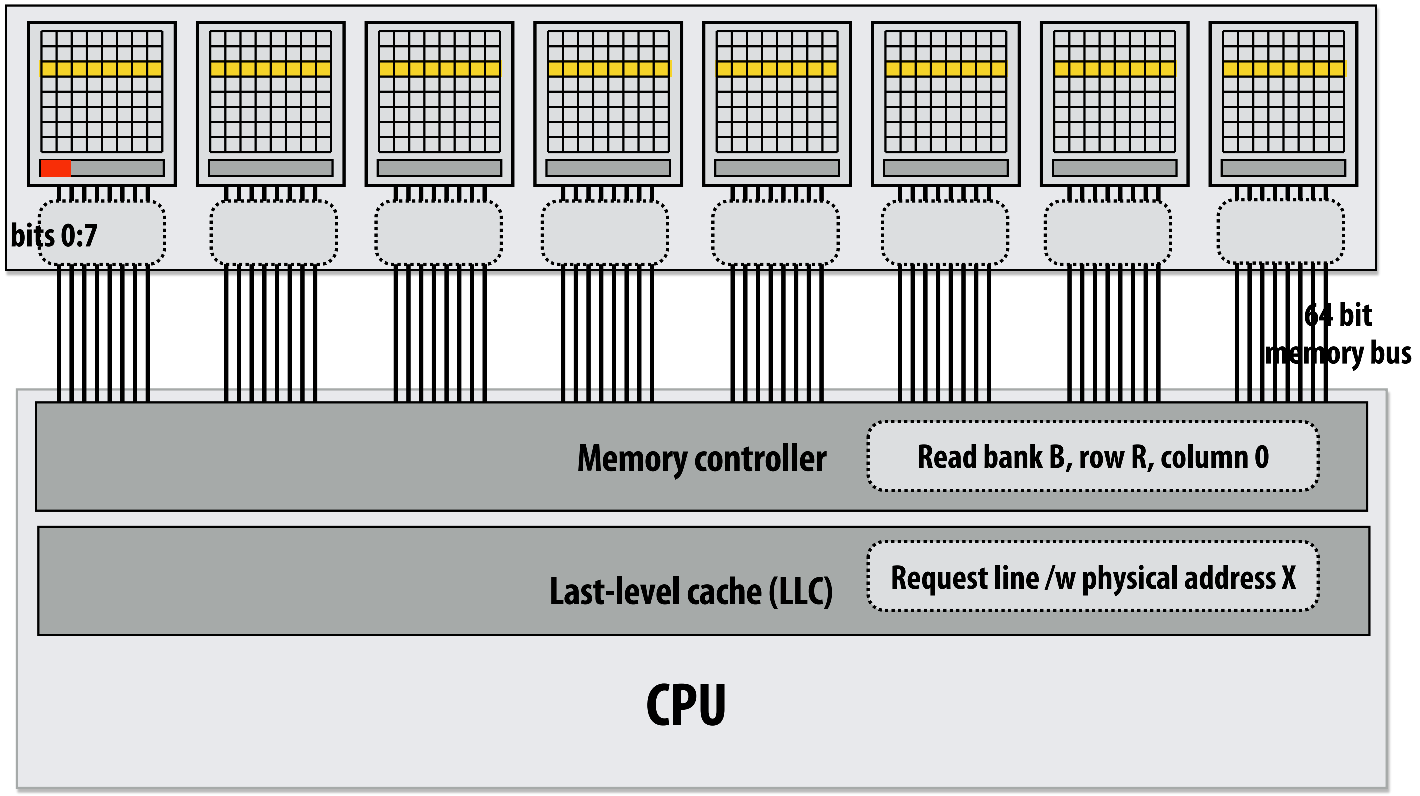
Note: DIMM appears as a single, higher capacity, wider interface DRAM module to the memory controller. Higher aggregate bandwidth, but minimum transfer granularity is now 64 bits.



Reading one 64-byte (512 bit) cache line (the wrong way)

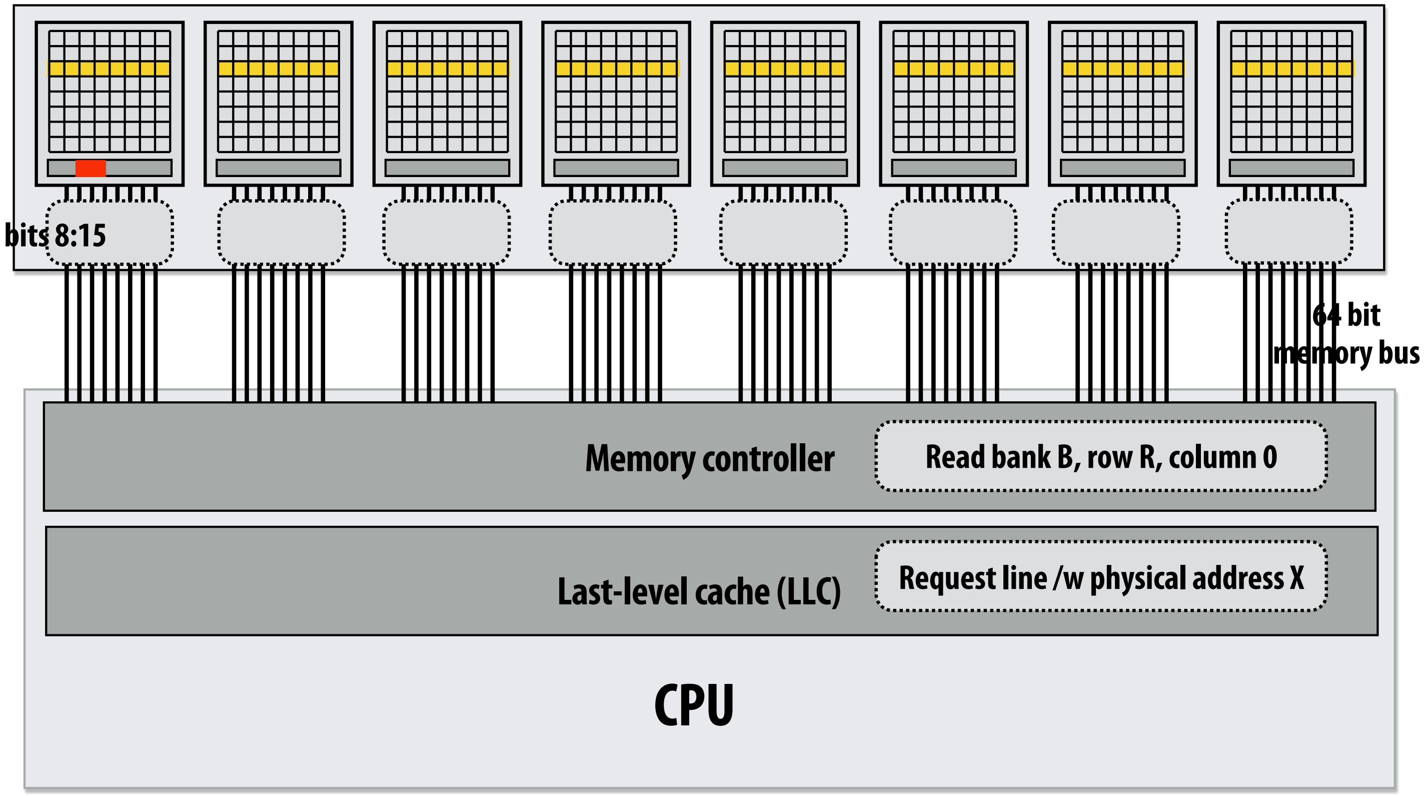
Assume: consecutive physical addresses mapped to same row of same chip

Memory controller converts physical address to DRAM bank, row, column



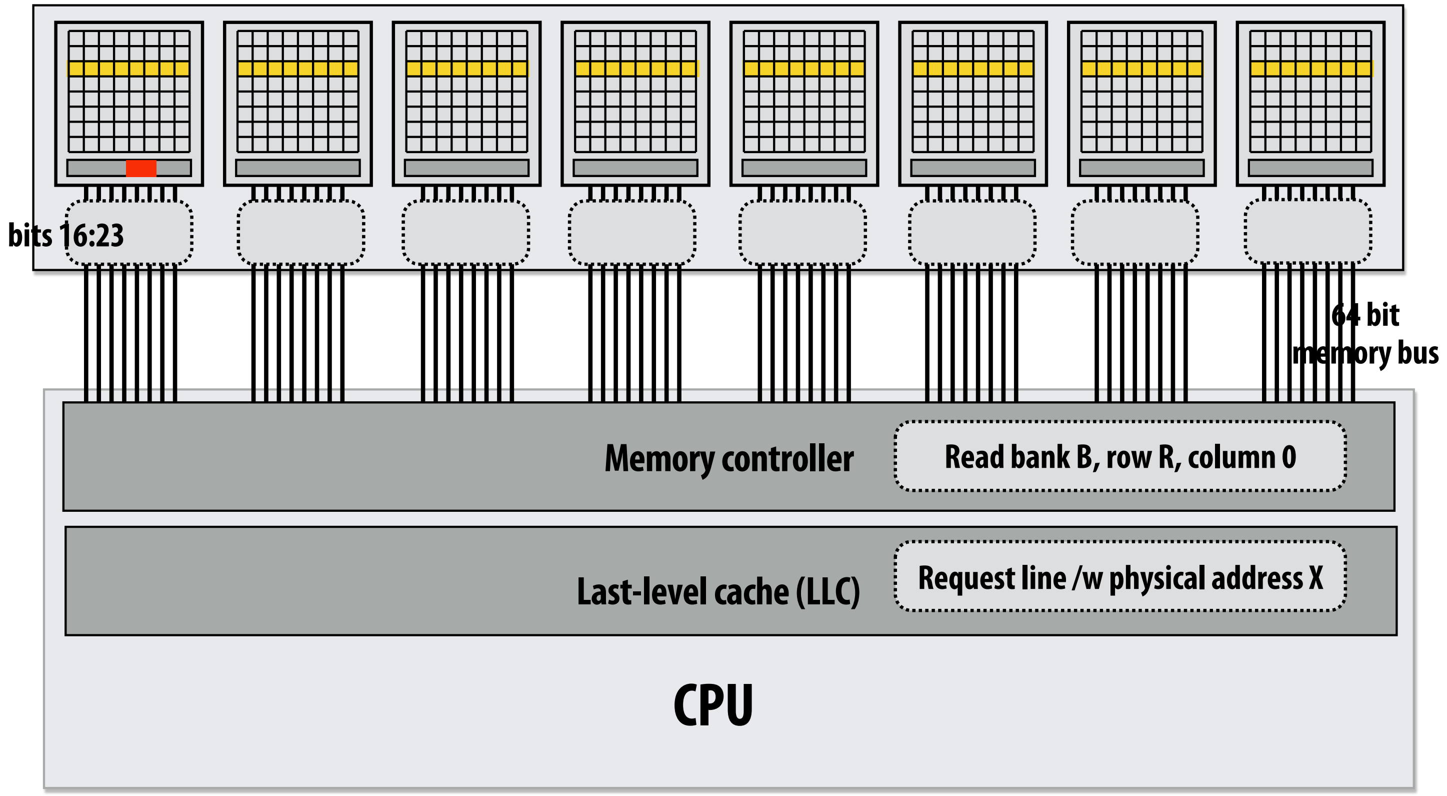
Reading one 64-byte (512 bit) cache line (the wrong way)

All data for cache line serviced by the same chip
Bytes sent consecutively over same pins



Reading one 64-byte (512 bit) cache line (the wrong way)

All data for cache line serviced by the same chip
Bytes sent consecutively over same pins

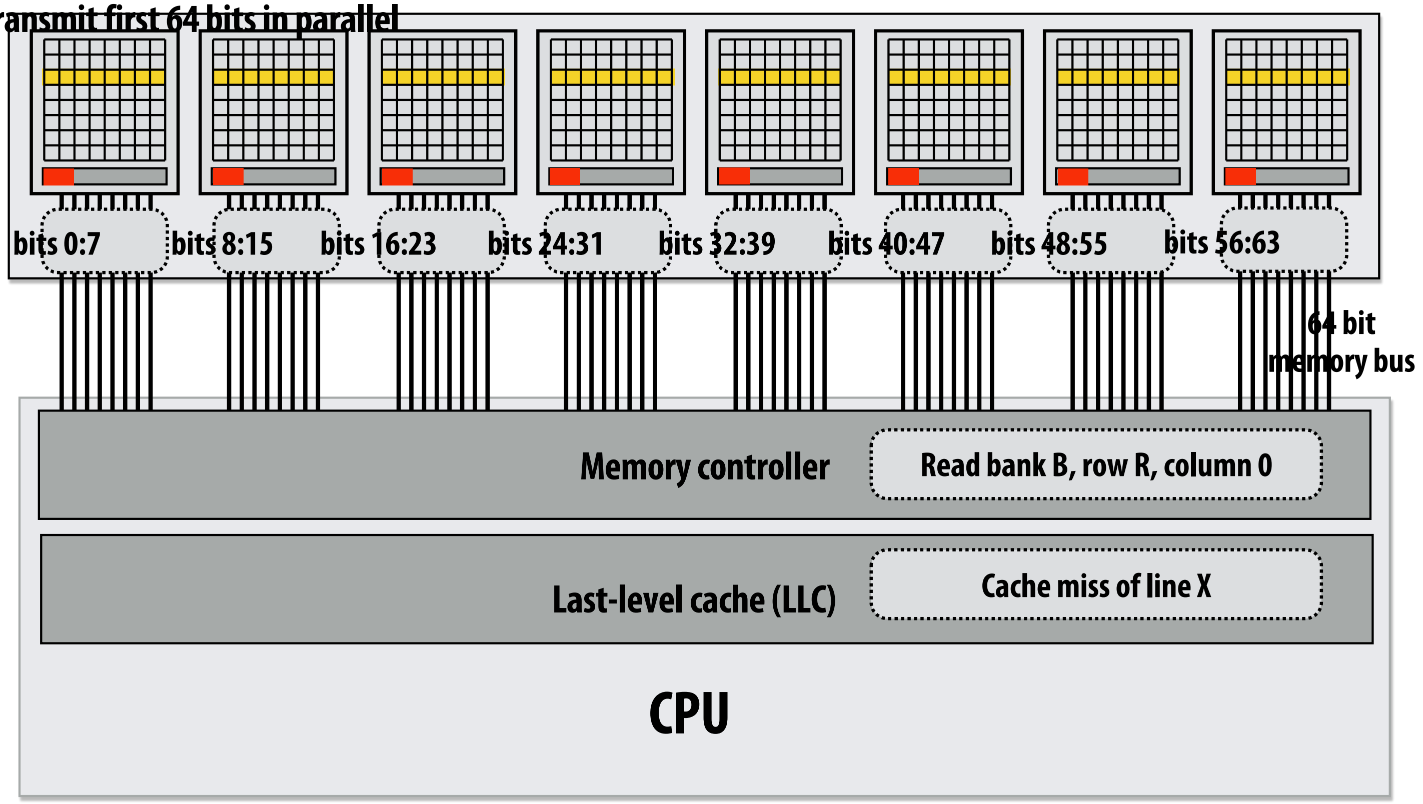


Reading one 64-byte (512 bit) cache line

Memory controller converts physical address to DRAM bank, row, column

Here: physical addresses are interleaved across DRAM chips at byte granularity

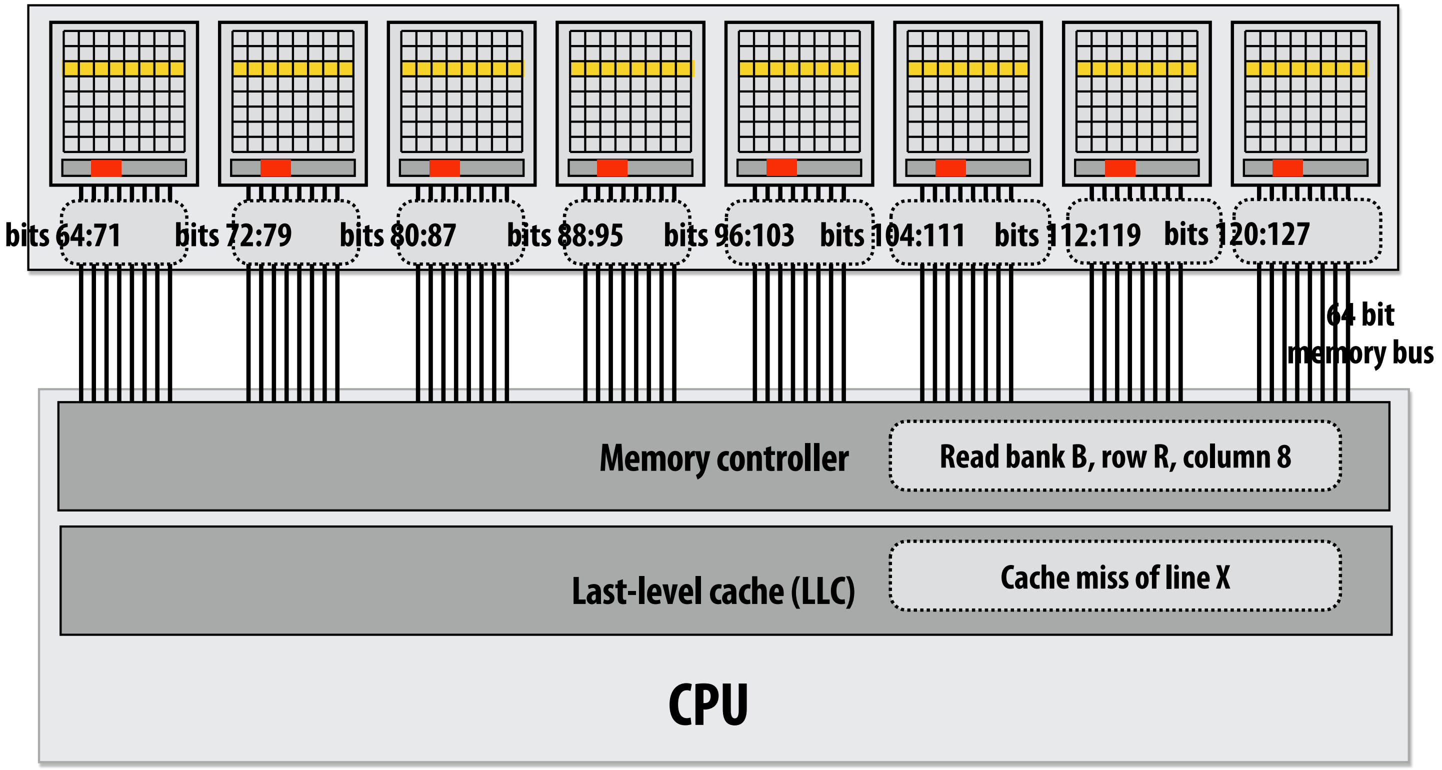
DRAM chips transmit first 64 bits in parallel



Reading one 64-byte (512 bit) cache line

DRAM controller requests data from new column *

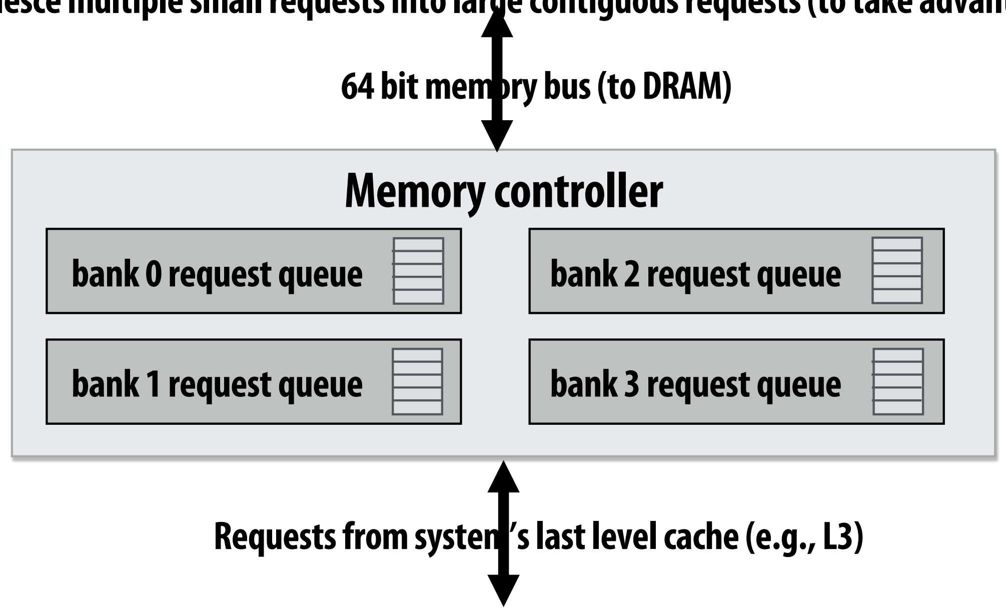
DRAM chips transmit next 64 bits in parallel



* Recall modern DRAM's support burst mode transfer of multiple consecutive columns, which would be used here

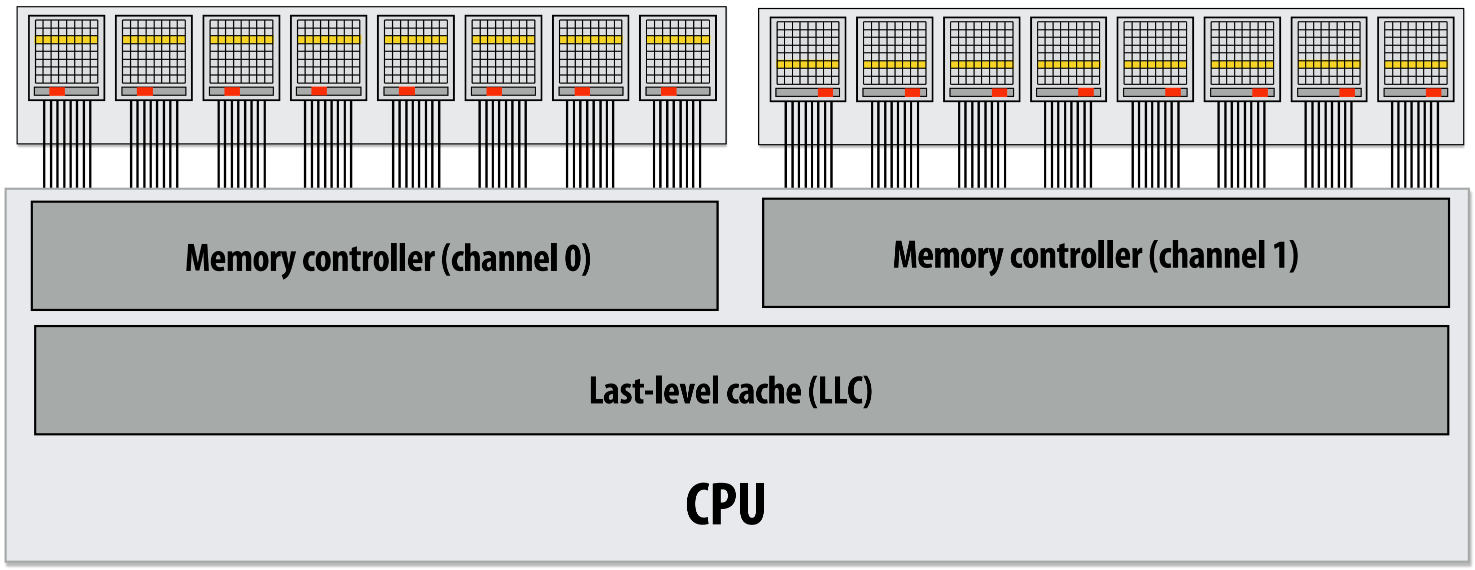
Memory controller is a memory request scheduler

- Receives load/store requests from LLC
- Conflicting scheduling goals
  - Maximize throughput, minimize latency, minimize energy consumption
  - Common scheduling policy: FR-FCFS (first-ready, first-come-first-serve)
    - Service requests to currently open row first (maximize row locality)
    - Service requests to other rows in FIFO order
  - Controller may coalesce multiple small requests into large contiguous requests (to take advantage of DRAM "burst modes")



Dual-channel memory system

- Increase throughput by adding memory channels (effectively widen bus)
- Below: each channel can issue independent commands
  - Different row/column is read in each channel
  - Simpler setup: use single controller to drive same command to multiple channels



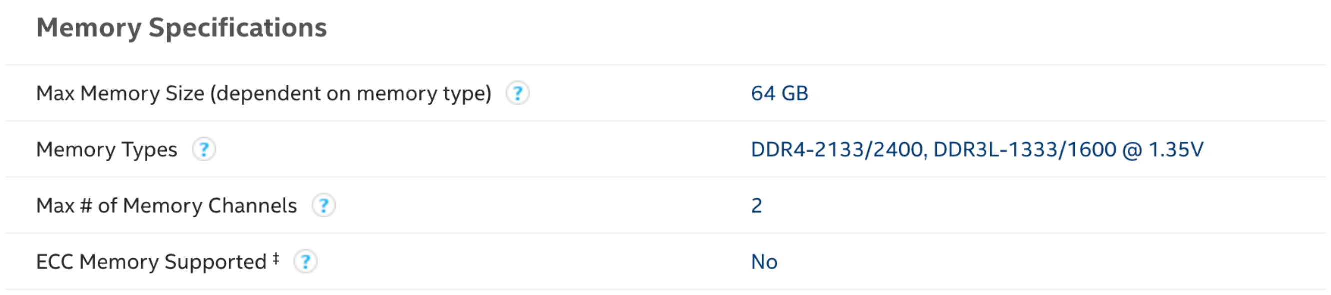
Example: DDR4 memory

Processor: Intel® Core™ i7-7700K Processor (in Myth cluster)

DDR4 2400

- 64-bit memory bus x 1.2GHz x 2 transfers per clock* = 19.2GB/s per channel
- 2 channels = 38.4 GB/sec
- ~13 nanosecond CAS

Memory system details from Intel's site:



Memory Specifications

Max Memory Size (dependent on memory type)	64 GB
Memory Types	DDR4-2133/2400, DDR3L-1333/1600 @ 1.35V
Max # of Memory Channels	2
ECC Memory Supported ‡	No

*** DDR stands for "double data rate"**

https://ark.intel.com/content/www/us/en/ark/products/97129/intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html

DRAM summary

- DRAM access latency can depend on many low-level factors
  - Discussed today:
    - State of DRAM chip: row hit/miss? is recharge necessary?
    - Buffering/reordering of requests in memory controller
- Significant amount of complexity in a modern multi-core processor has moved into the design of memory controller
  - Responsible for scheduling ten's to hundreds of outstanding memory requests
  - Responsible for mapping physical addresses to the geometry of DRAMs
  - Area of active computer architecture research

Modern architecture challenge: improving memory performance:

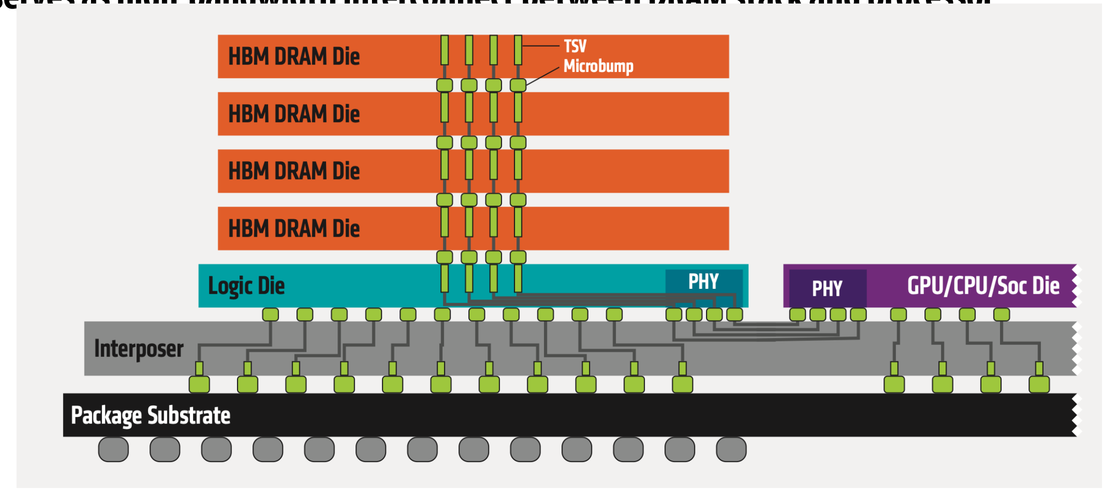
Decrease distance data must move by locating memory closer to processors

(enables shorter, but wider interfaces)

Increase bandwidth, reduce power by chip stacking

Enabling technology: 3D stacking of DRAM chips

- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
- TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack "logic layer" is memory controller, manages requests from processor
- Silicon "interposer" serves as high-bandwidth interconnect between DRAM stack and processor

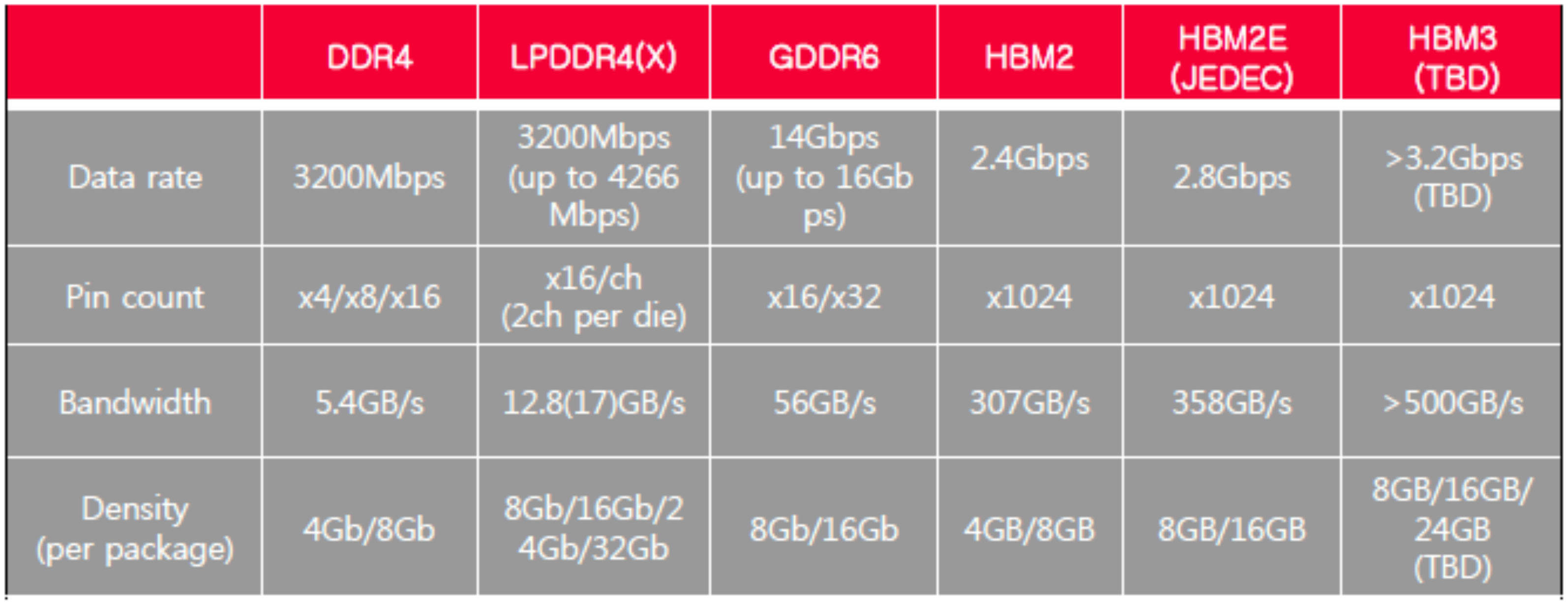


Technologies:
Micron/Intel Hybrid Memory Cube (HBC)
High-bandwidth memory (HBM) - 1024 bit interface to stack

Image credit: AMD

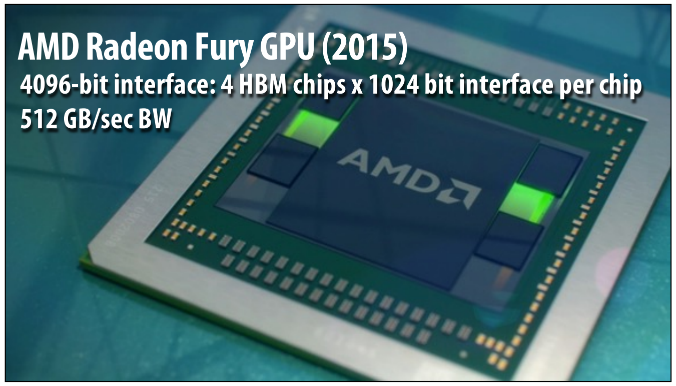
HBM Advantages

More Bandwidth
High Power Efficiency
Small Form Factor



	DDR4	LPDDR4(X)	GDDR6	HBM2	HBM2E (JEDEC)	HBM3 (TBD)
Data rate	3200Mbps	3200Mbps (up to 4266 Mbps)	14Gbps (up to 16Gb ps)	2.4Gbps	2.8Gbps	>3.2Gbps (TBD)
Pin count	x4/x8/x16	x16/ch (2ch per die)	x16/x32	x1024	x1024	x1024
Bandwidth	5.4GB/s	12.8(17)GB/s	56GB/s	307GB/s	358GB/s	>500GB/s
Density (per package)	4Gb/8Gb	8Gb/16Gb/24Gb/32Gb	8Gb/16Gb	4GB/8GB	8GB/16GB	8GB/16GB/24GB (TBD)

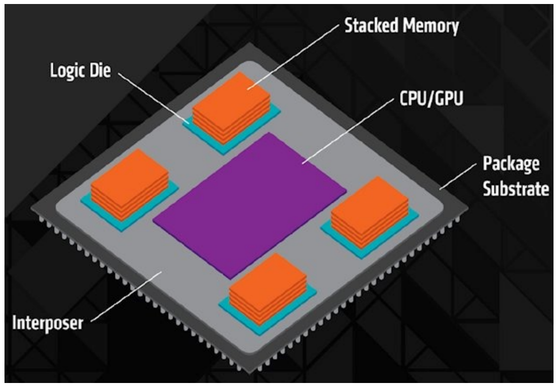
GPUs are adopting HBM technologies

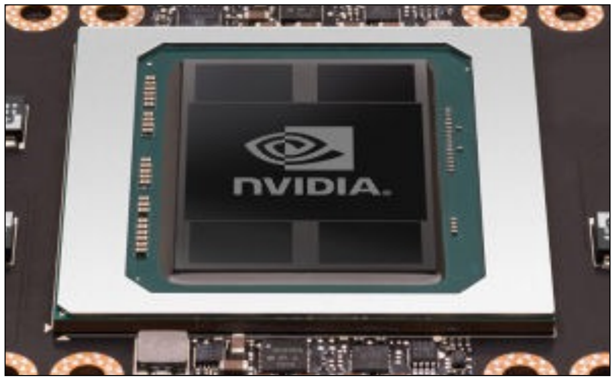


AMD Radeon Fury GPU (2015)

4096-bit interface: 4 HBM chips x 1024 bit interface per chip

512 GB/sec BW



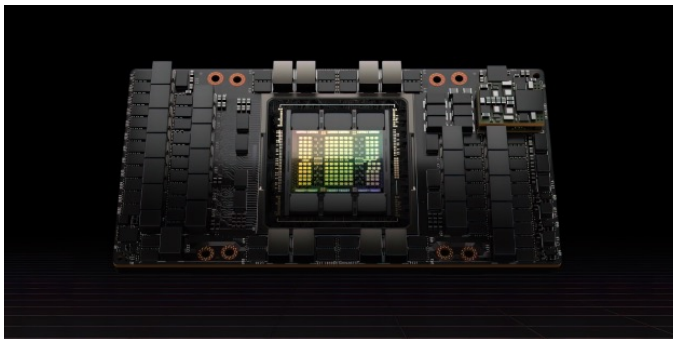


NVIDIA P100 GPU (2016)

4096-bit interface: 4 HBM2 chips x 1024 bit interface per chip

720 GB/sec peak BW

4 x 4 GB = 16 GB capacity



NVIDIA H100 GPU (2022)

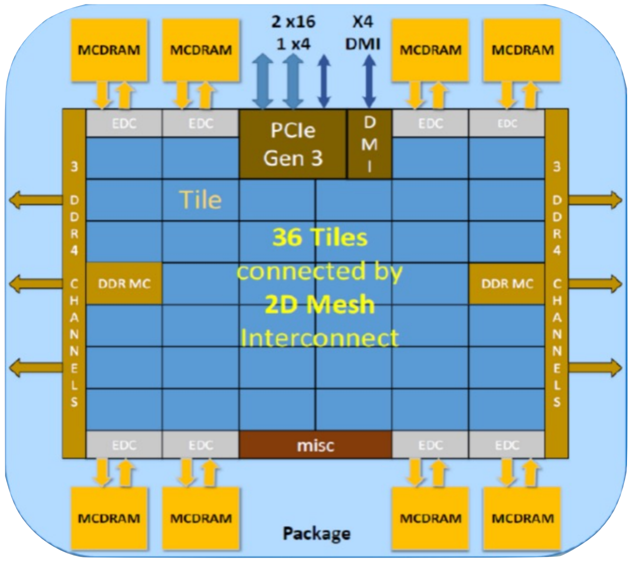
6144-bit interface: 6 HBM3 stacks x 1024 bit interface per stack

3.2 TB/sec peak BW

80 GB capacity

Xeon Phi (Knights Landing) MCDRAM

- 16 GB in package stacked DRAM
- Can be treated as a 16 GB last level cache
- Or as a 16 GB separate address space ("flat mode")
- Intel's claims:
  - ~ same latency at DDR4
  - ~5x bandwidth of DDR4
  - ~5x less energy cost per bit transferred



```
ate buffer in MCDRAM ("high bandwidth" memory malloc)
loat* foo = hbw_malloc(sizeof(float) * 1024);
```

Summary: the memory bottleneck is being addressed in many ways

- **By the application programmer**
  - Schedule computation to maximize locality (minimize required data movement)
- **By new hardware architectures**
  - Intelligent DRAM request scheduling
  - Bringing data closer to processor (deep cache hierarchies, 3D stacking)
  - Increase bandwidth (wider memory systems)
  - Ongoing research in locating limited forms of computation “in” or near memory
  - Ongoing research in hardware accelerated compression (not discussed today)
- **General principles**
  - Locate data storage near processor
  - Move computation to data storage
  - Data compression (trade-off extra computation for less data transfer)