New Hardware Architectures for Data Management

Cagri Balkesen, Louis Woods, and Jens Teubner
ETH Zurich, Systems Group
{firstname.lastname}@inf.ethz.ch

March 12, 2013

CMOS Technology

1965: "Moore's Law": [Moore 1965]

- Number of transistors/chip doubles every two years.
  - → Microarchitecture → 40 % faster (Pollack's rule)

1974: "Dennard Scaling": [Dennard *et al.* 1974]

- Reduced CMOS gate length:
  - → faster switching (higher frequency)
  - → reduced supply voltage and capacity
  - → **power/area remains constant!**

→ **Performance doubles every two years "at not cost."**

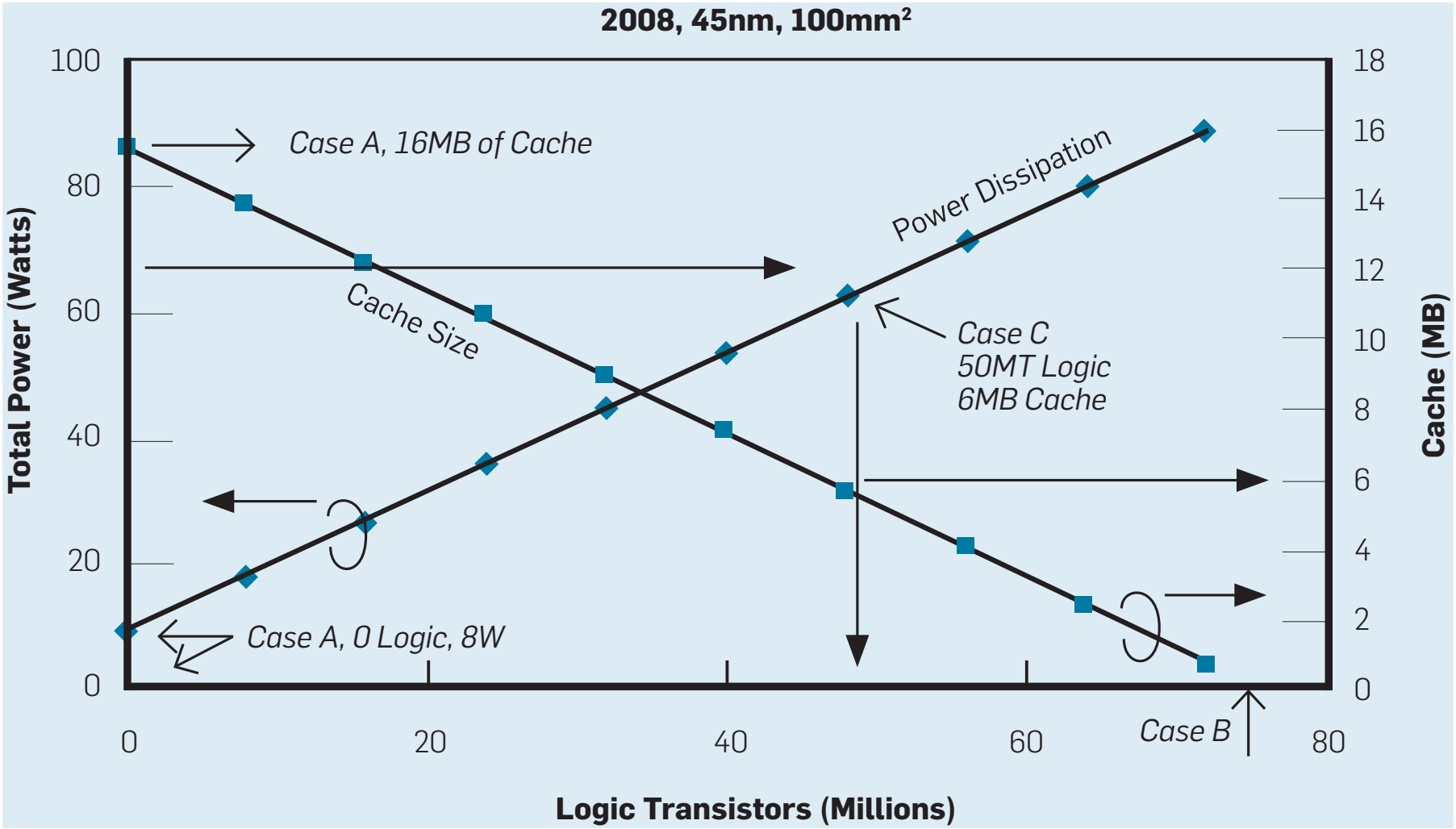
Dennard Scaling

Dennard scaling is reaching its limits.

- Supply Voltage ↘ → Threshold Voltage ↘
- Threshold Voltage ↘ → **Leakage Current** ↗
- Leakage Current ↗ → power consumption ↗

All modern chip designs are **power-limited**!

Constrained by Power



Source: Borkar and Chien. The Future of Microprocessors. *CACM 2011.*

Moore's Law

Moore's Law **still prevails**.

→ More and more transistors to spend.

→ But **how** (without exceeding the energy budget)?

Design Space

1. **Parallelism**
   → Lower clock, better energy efficiency
2. **Locality**
   → Moving data costs a lot of energy
3. **Heterogeneous Hardware**
   → Specialized hardware orders of magnitude more energy efficient
   ⇝ Dark silicon [Esmaeilzadeh *et al.* 2013]

Today:

1. Join Processing on **Multi-Cores**
2. **Graphics Processors (GPUs)**
3. **Field-Programmable Gate Arrays (FPGAs)**

Part II

Multi-Core Architectures

Key Challenges

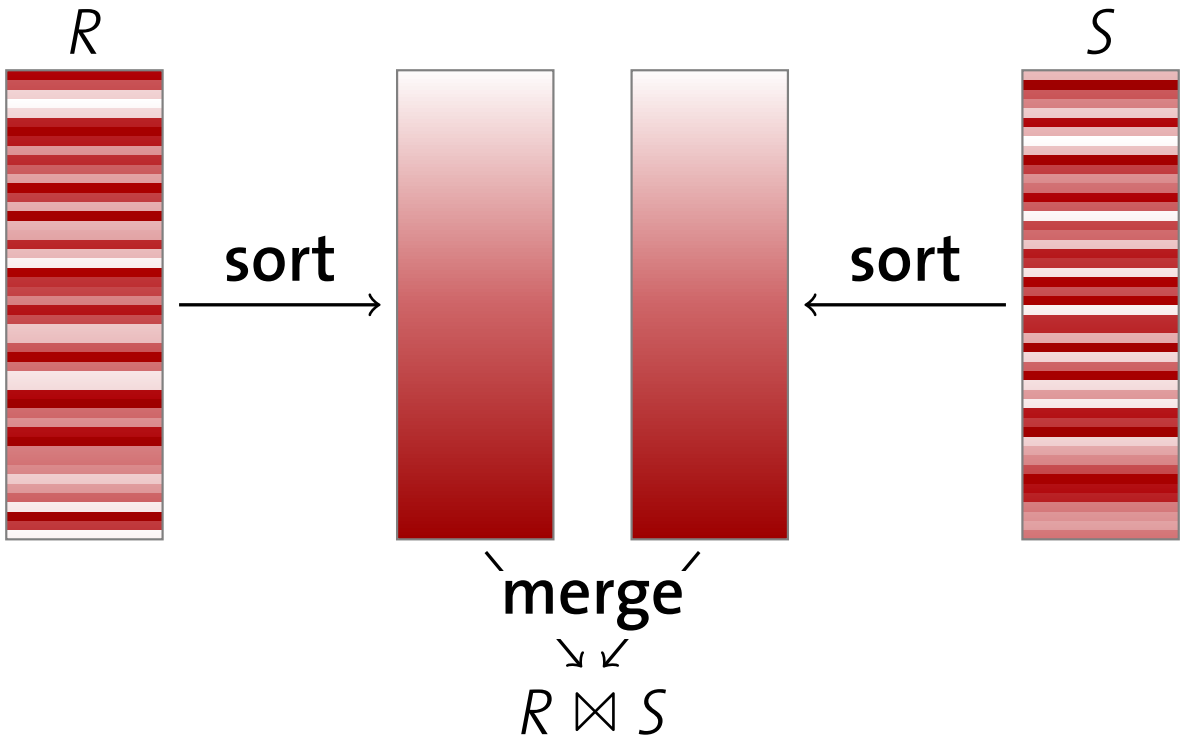
Key challenges:

- memory wall
- parallelism
  - → task-level parallelism (SMT, multi-core)
  - → data-level parallelism (SIMD)

Today:

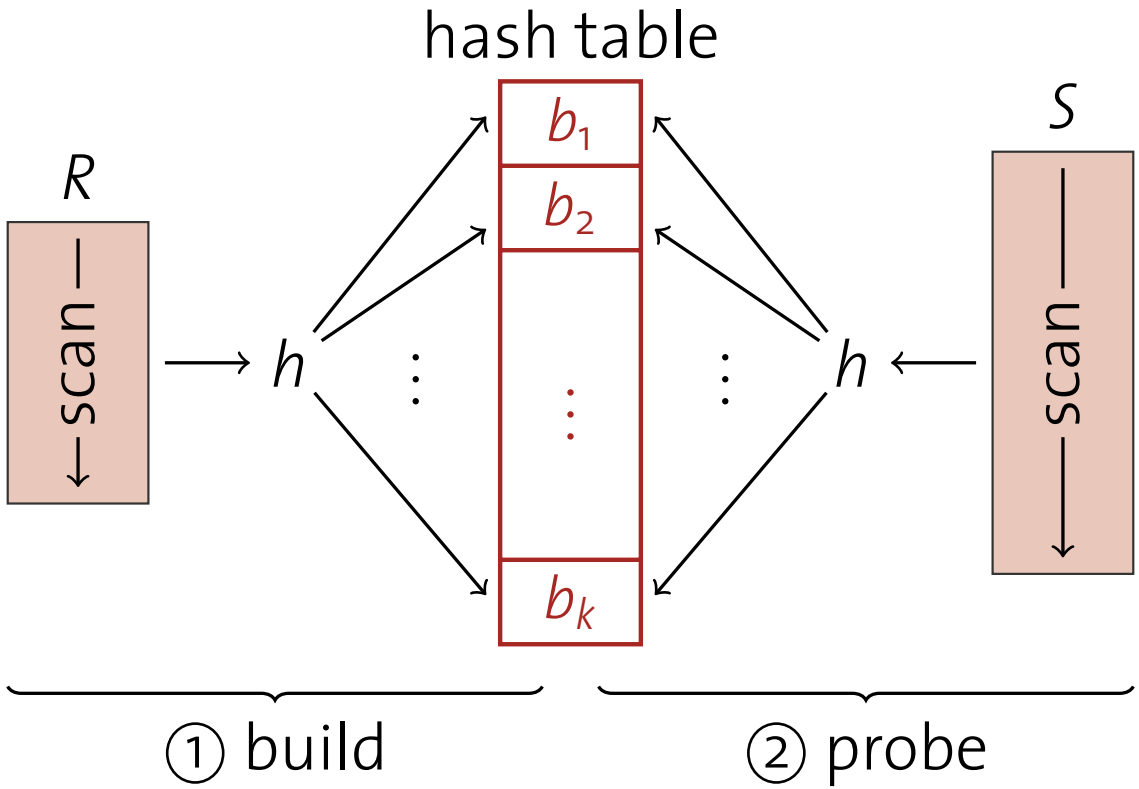
- in-memory joins on modern multi-core machines

Approach 1: Sort (and Merge)



- ✓ Can be done as **external sort**
- ✓ $\mathcal{O}(N \log N)$

Approach 2: Hash

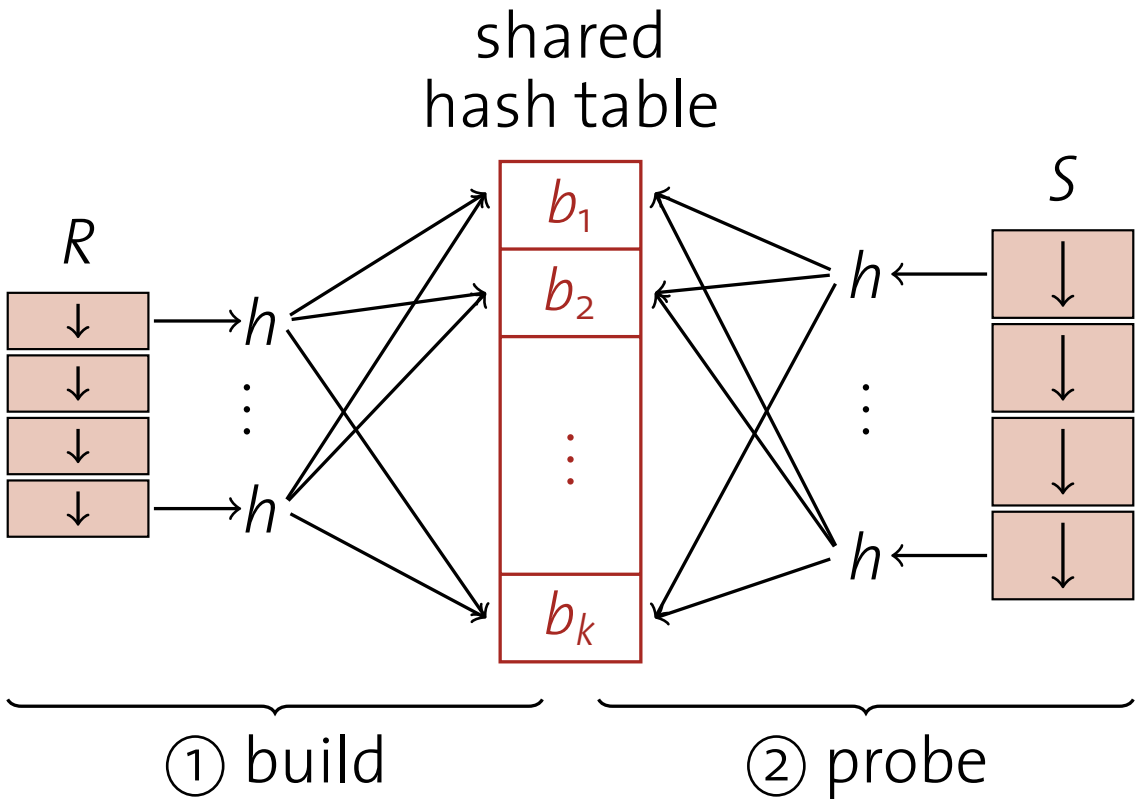


- ✓ $\mathcal{O}(N)$ (approx.)
- ✓ Easy to **parallelize**

Modern Hardware?

Parallel Hash Join

Parallel Hash Join ("no partitioning" join of [Blanas *et al.* 2011])



✓ Protect using locks; **very low contention**

Modern Hardware

☹ **Random access pattern**

→ Every hash table access a **cache miss**

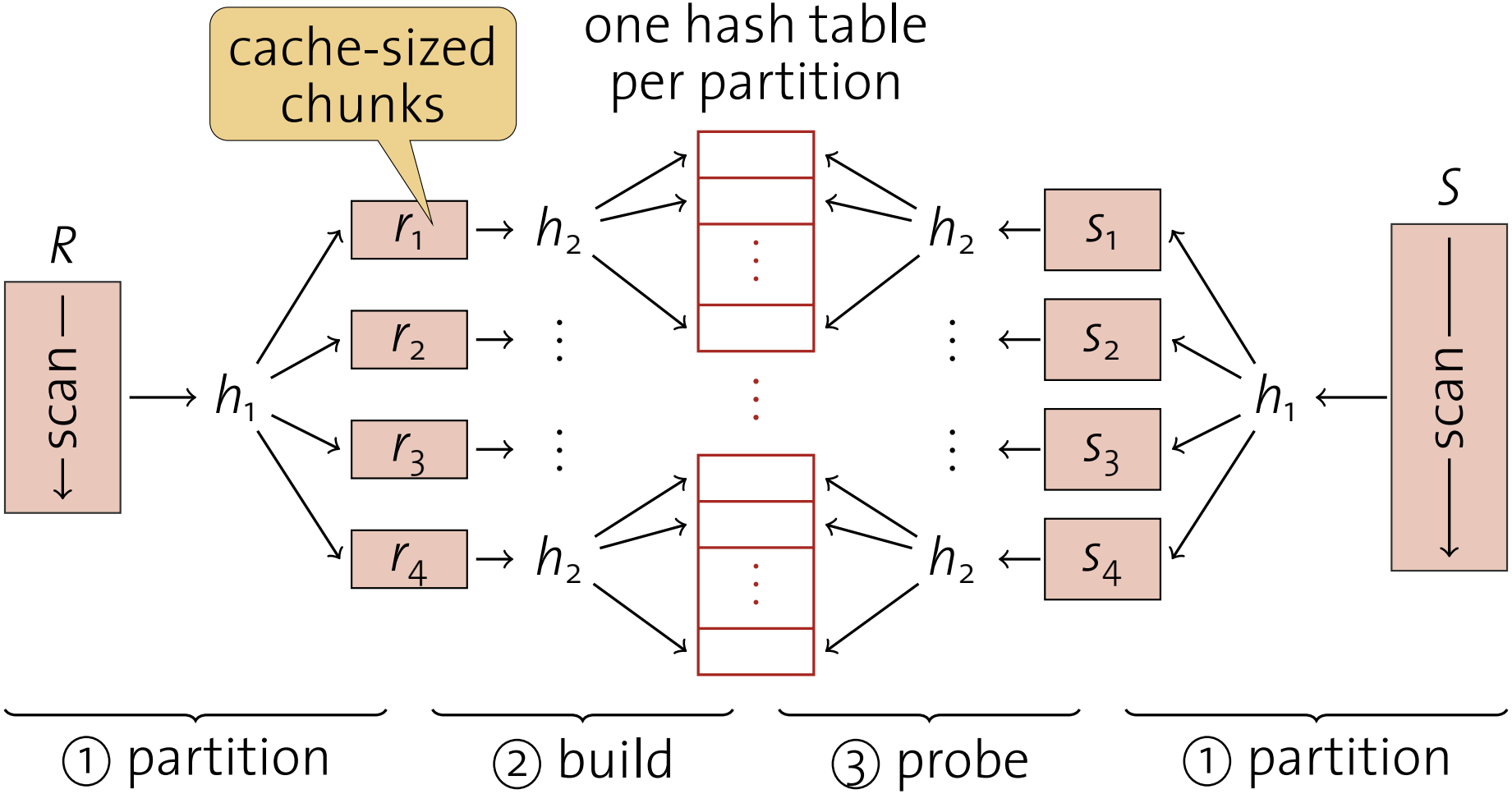
Cost per tuple (build phase):

- 34 assembly instructions
- 1.5 cache misses
- 3.3 TLB misses

} hash join is severely **latency-bound**

Partitioned Hash Join

Thus: **partitioned hash join** [Shatdal *et al.* 1994]



(parallelism: assign partitions to threads → no locking needed)

Cache Effects

Build/probe now contained within caches:

- 15/21 instructions per tuple (build/probe)
- ≈ 0.01 cache misses per tuple
- almost no TLB misses ✓

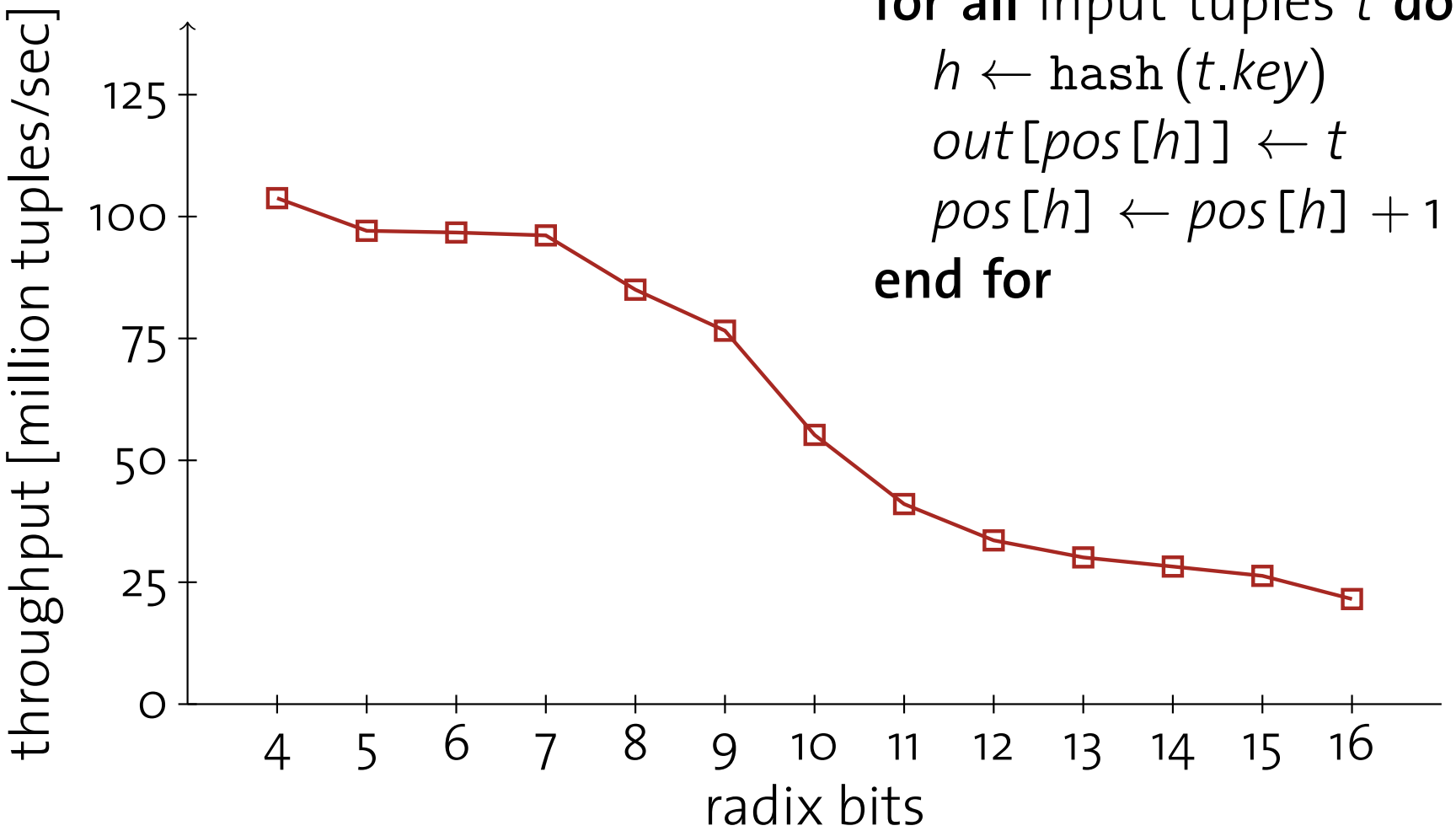
Partitioning is now critical

→ Many partitions, far apart

→ Each one will reside on its own page

→ Run out of **TLB entries** (100–500)

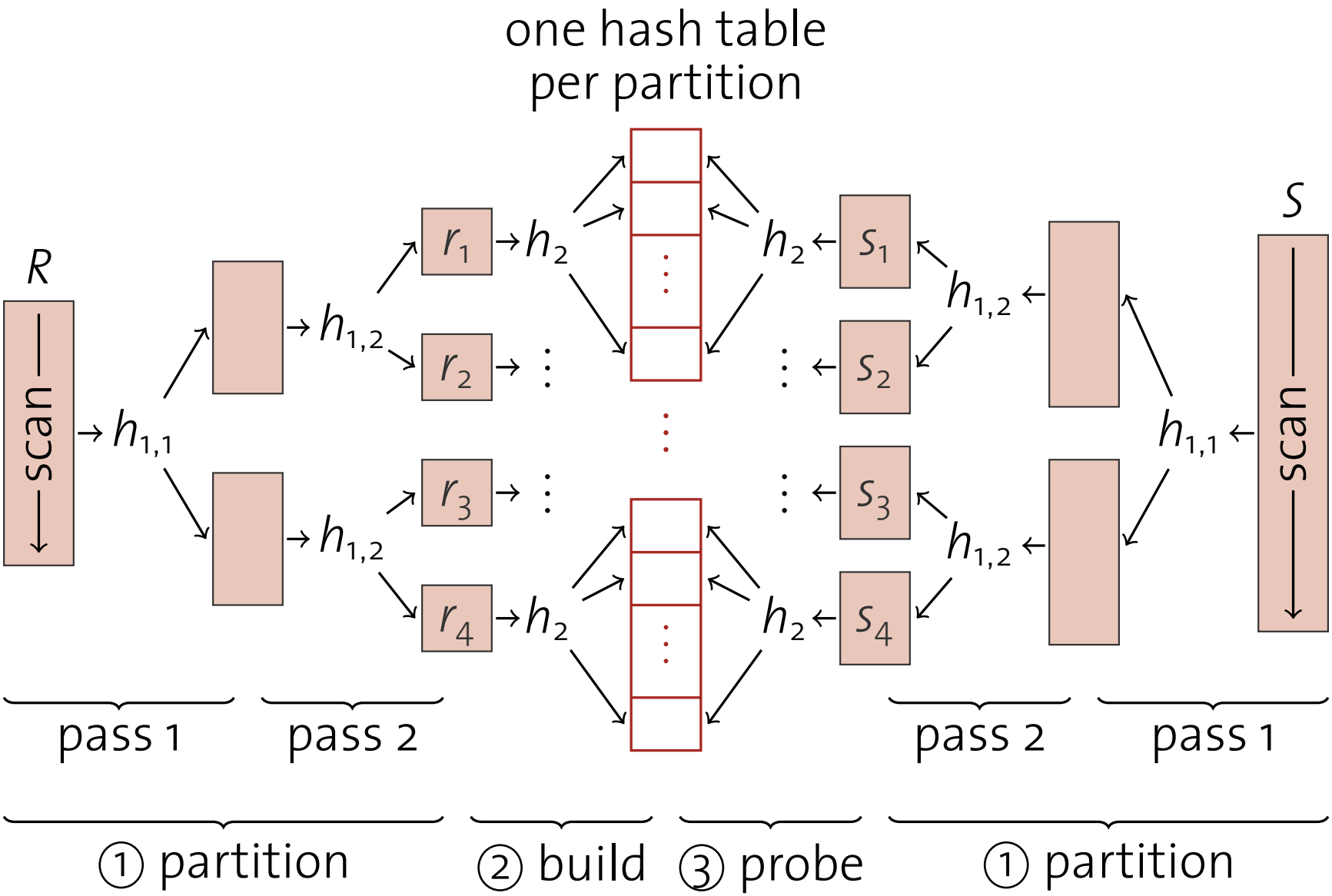
Cost of Partitioning



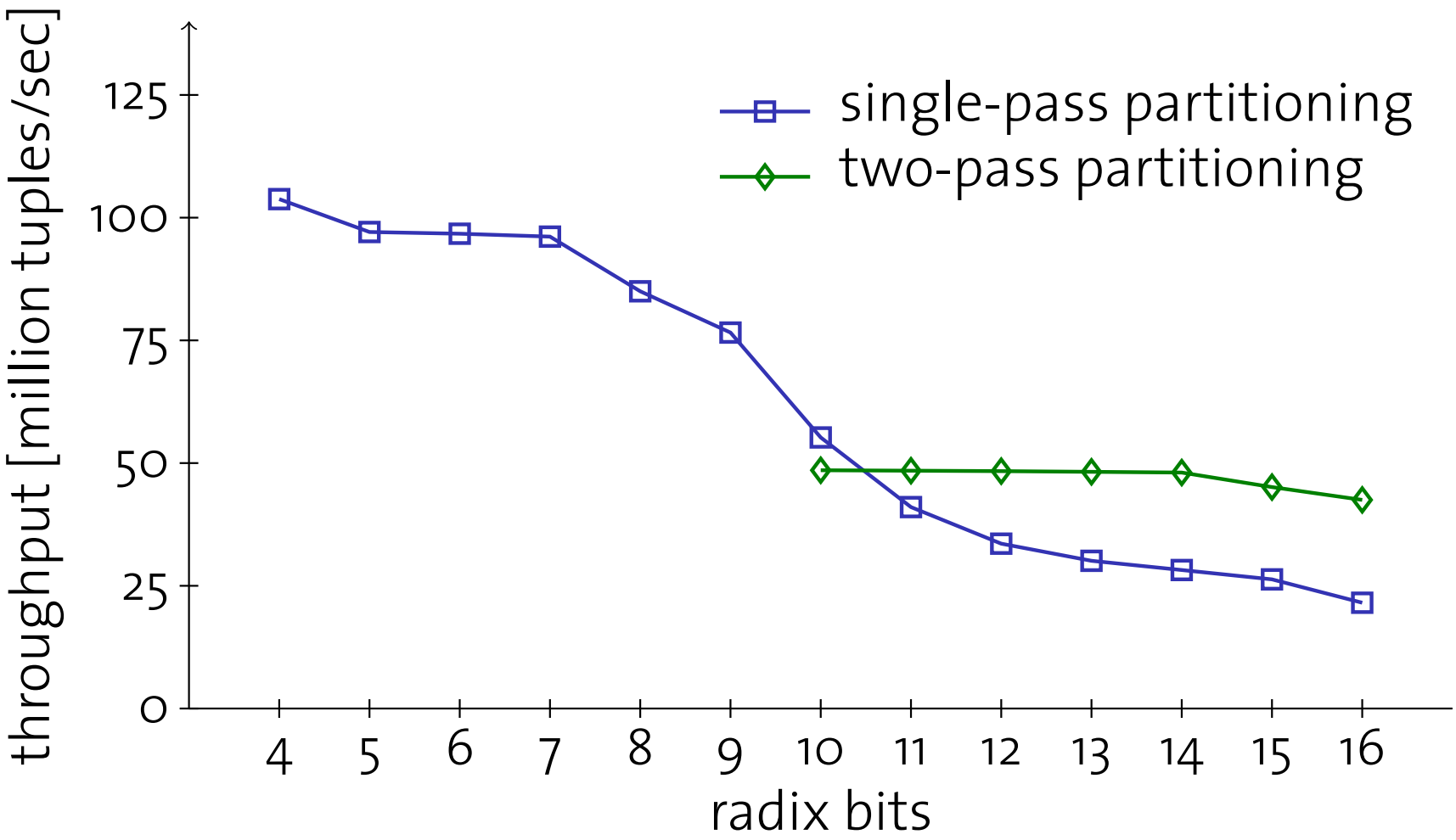
for all input tuples t **do**
$h \leftarrow \texttt{hash}(t.key)$
$out[pos[h]] \leftarrow t$
$pos[h] \leftarrow pos[h] + 1$
end for

→ Expensive beyond $\approx 2^8$–2^9 partitions.

Multi-pass partitioning ("radix partitioning")



Two-pass partitioning



Hash join is $\mathcal{O}(N \log N)$!

```
for all input tuples t do
    h ← hash(t.key)
    copy t to out[pos[h]]
    pos[h] ← pos[h] + 1
end for
```

memory access

Naïve partitioning (cf. slide 16)

```
for all input tuples t do
    h ← hash(t.key)
    buf[h][pos[h] mod bufsiz] ← t
    if pos[h] mod bufsiz = 0 then
        copy buf[h] to out[pos[h] − bufsiz]
    end if
    pos[h] ← pos[h] + 1
end for
```

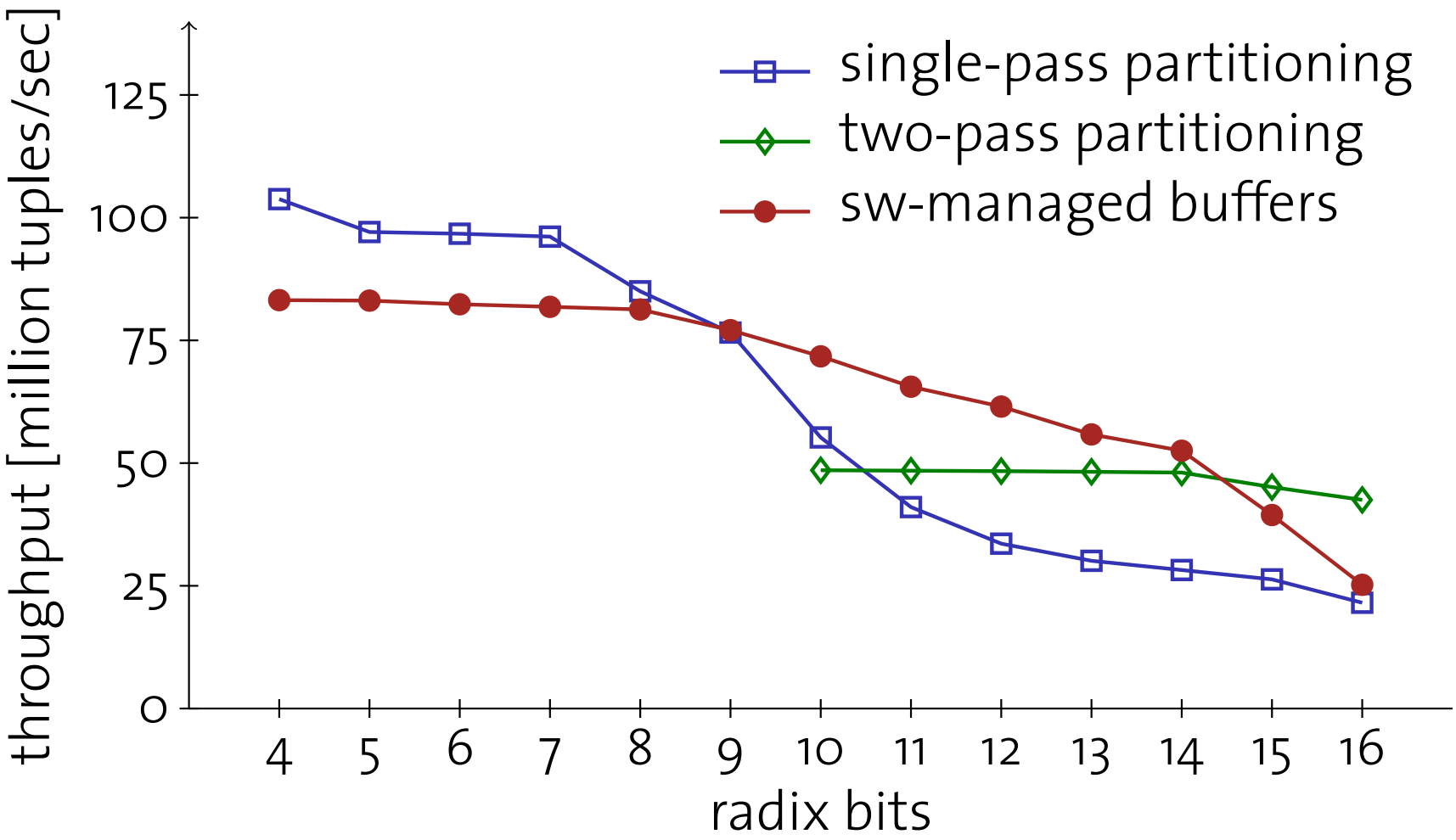
memory access

Software-Managed Buffers

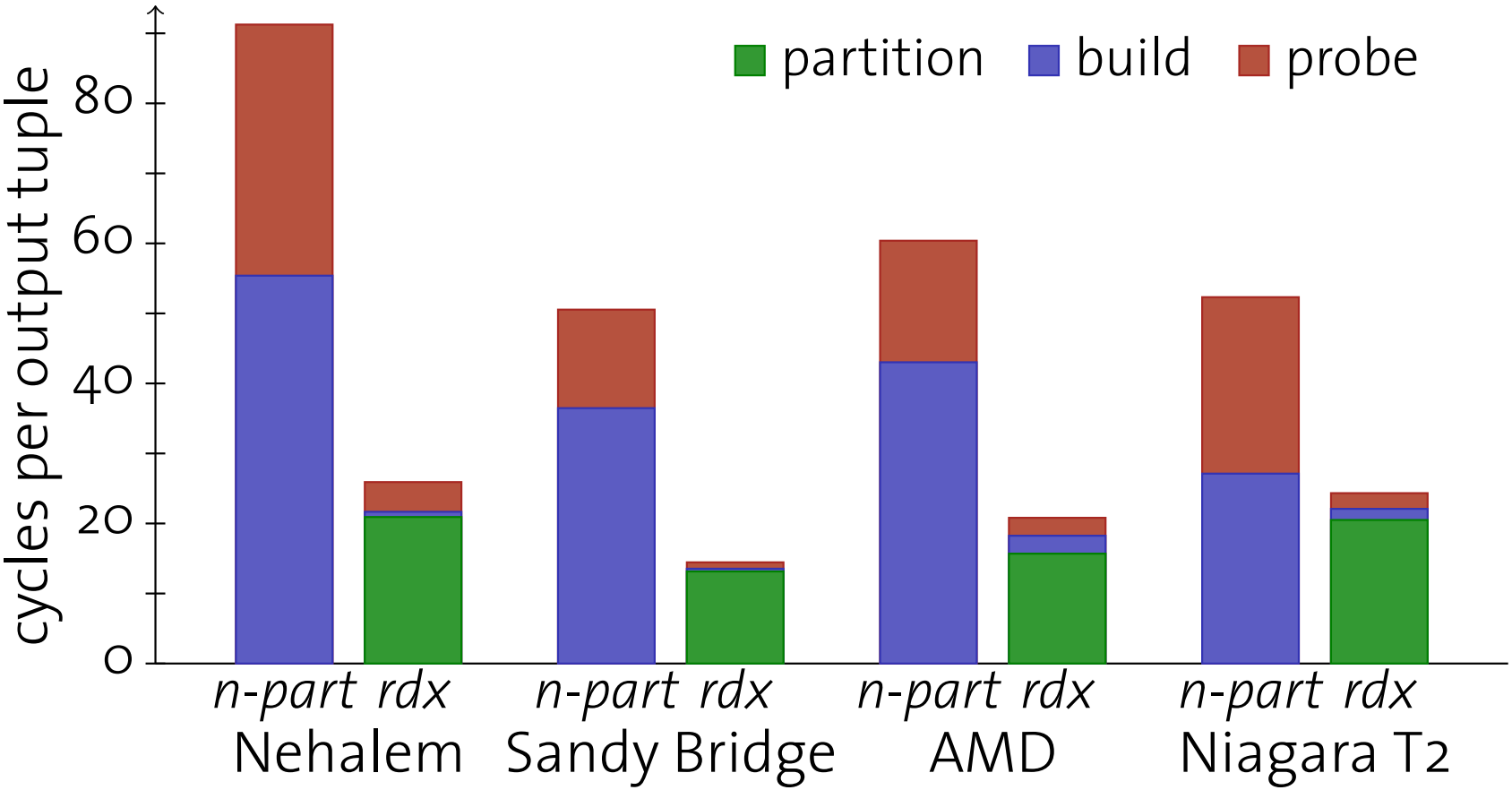
→ TLB miss only every *bufsiz* tuples

→ Choose *bufsiz* to match **cache line size**

Software-Managed Buffers



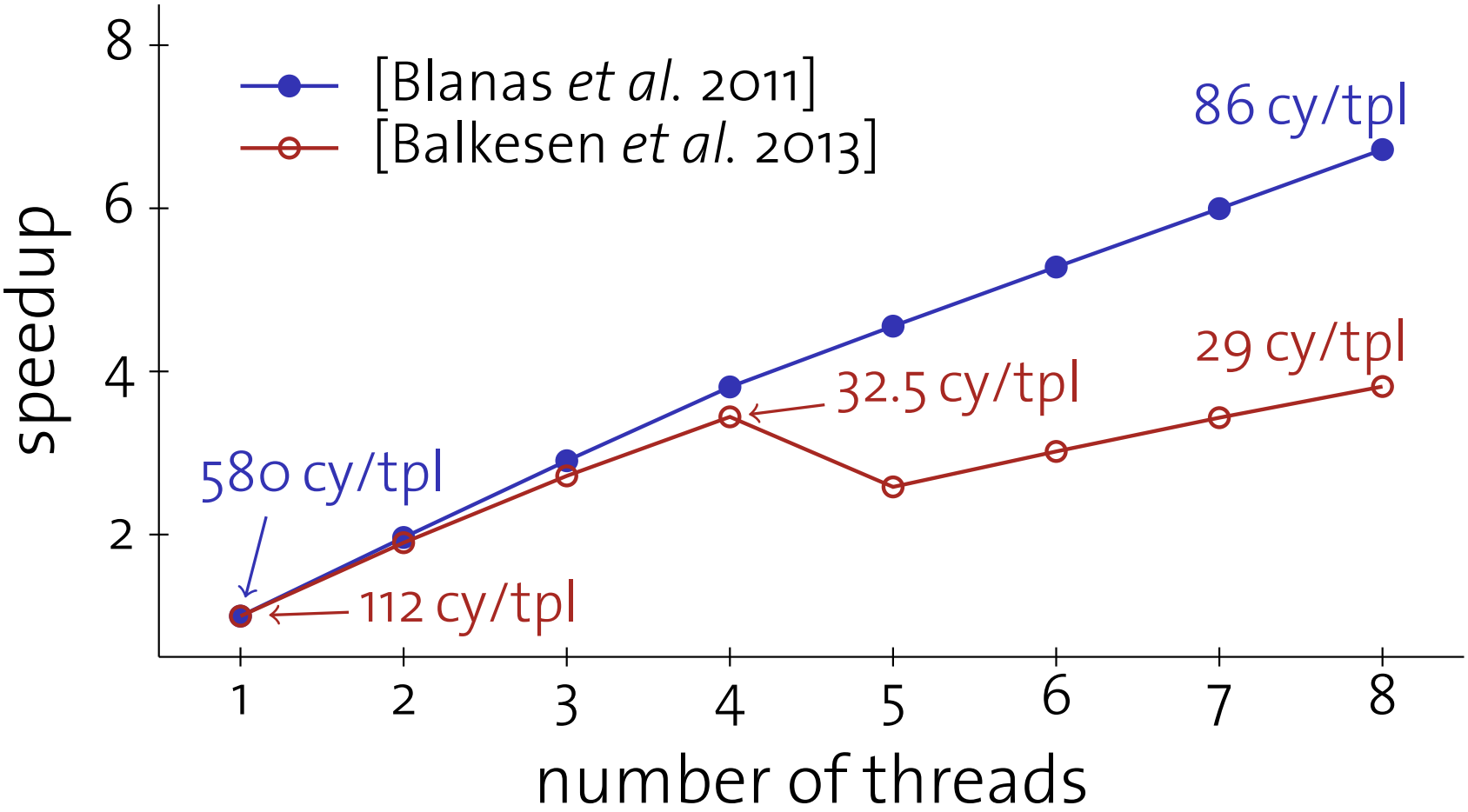
Plugging it Together



- 977 MiB ⋈ 977 MiB
- *e.g.*, Nehalem: 25 cy/tpl ≈ 90 million tuples per second

Nehalem: 4 cores/8 threads; 2.26 GHz · Sandy Bridge: 8 cores/16 threads; 2.7 GHz
AMD Bulldozer: 16 cores; 2.3 GHz · Niagara 2: 8 cores/64 threads; 1.2 GHz

A Word on "Scalability"

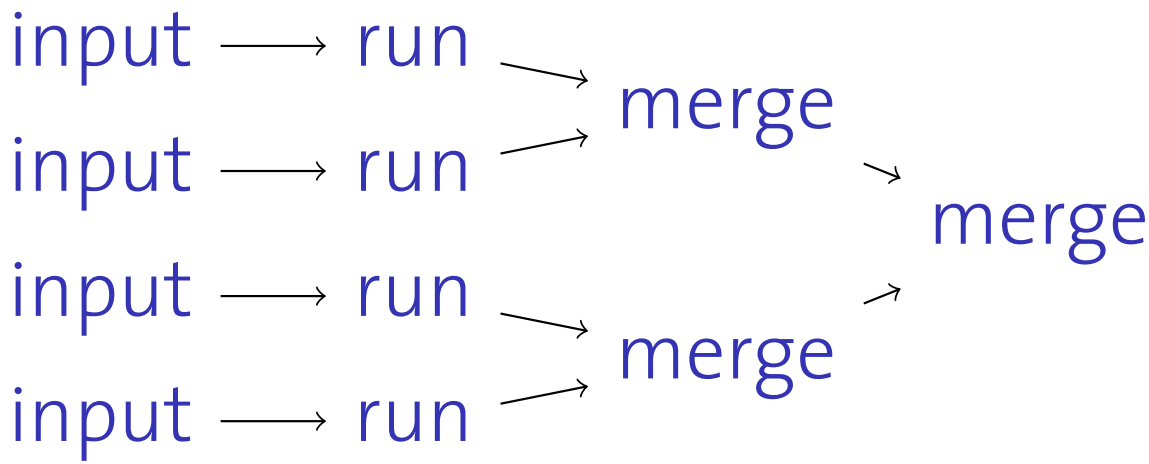


Sort-Merge Join

Critical part of sort-merge join is **sorting**.

- Method of choice: **merge sort**

  → two parts: **run generation** and **merging**

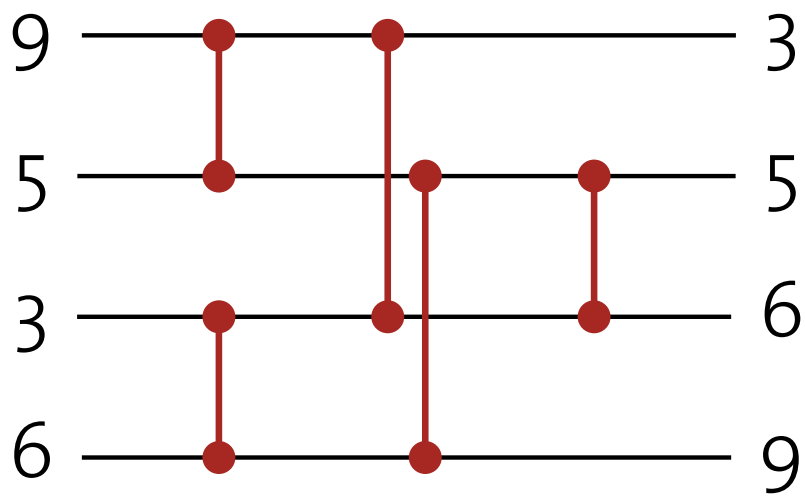


→ Both are good candidates for **SIMD acceleration**

Sorting networks

→ **branch free**, support **data parallelism**

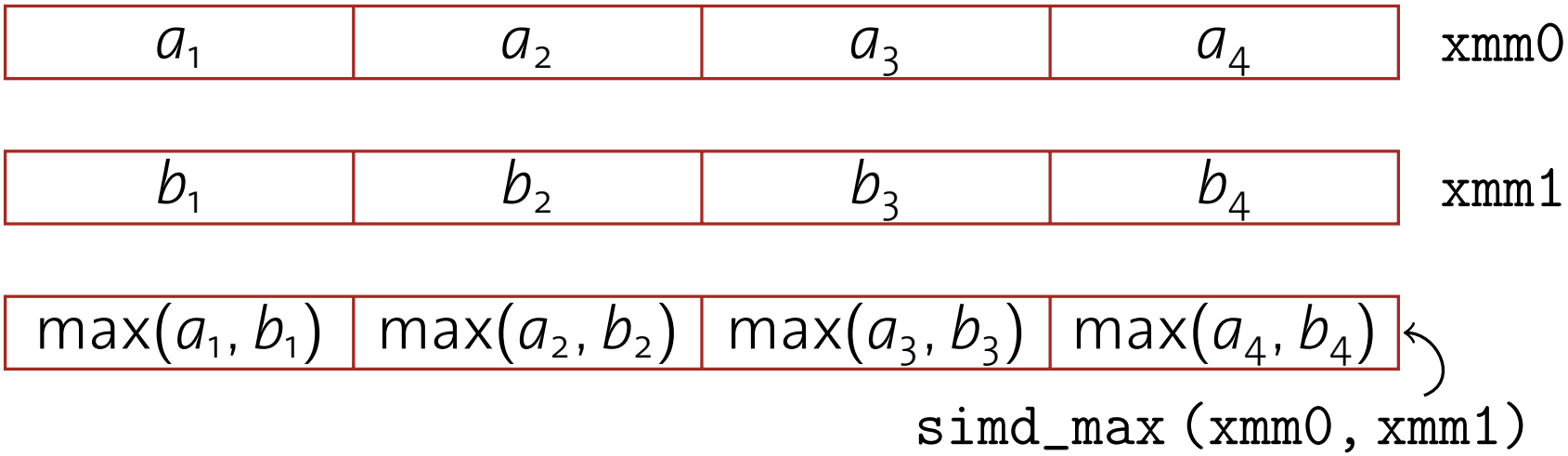
E.g., network for four elements (“even-odd network”):



→ Build larger networks by **merging** sorted runs.

SIMD instructions

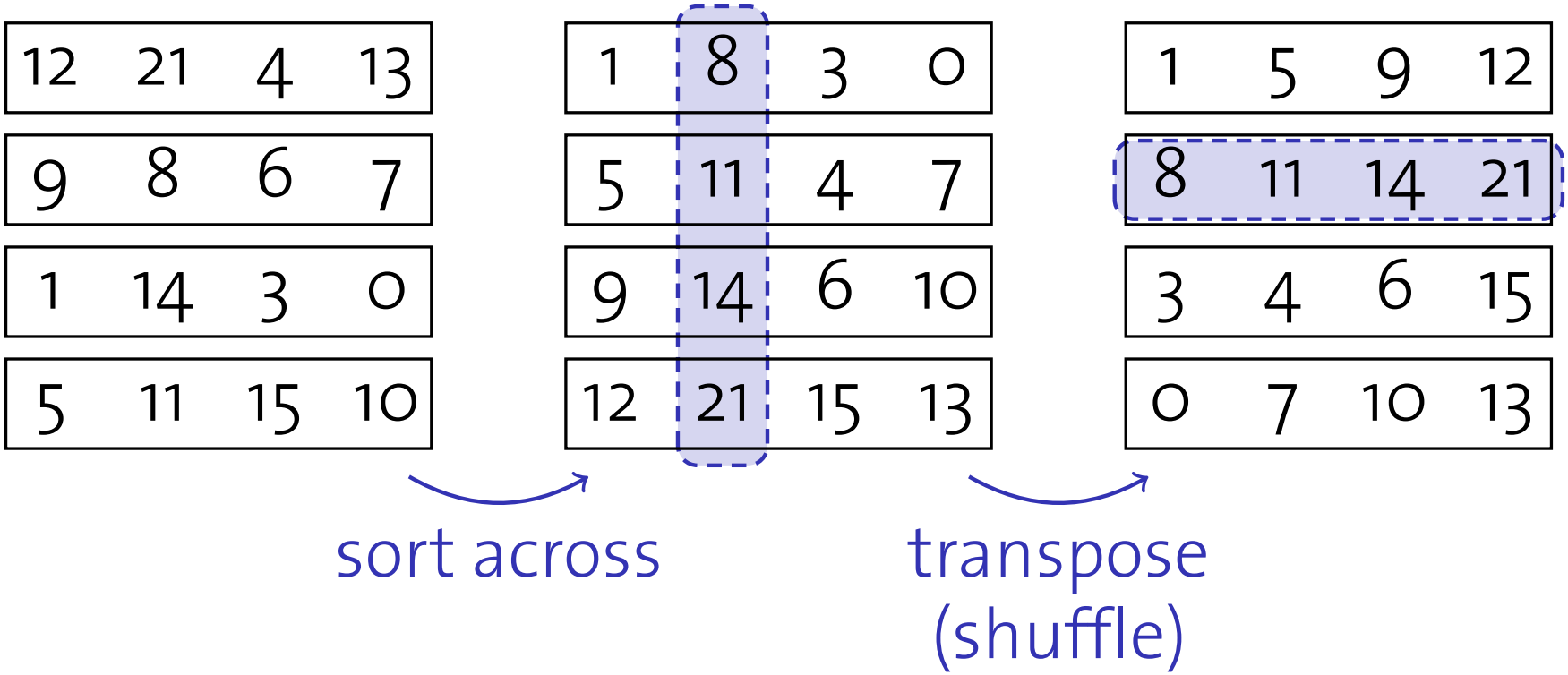
E.g., four words per SIMD register:



Operations **across** registers, **not within**

→ **But:** Can **shuffle** across and within

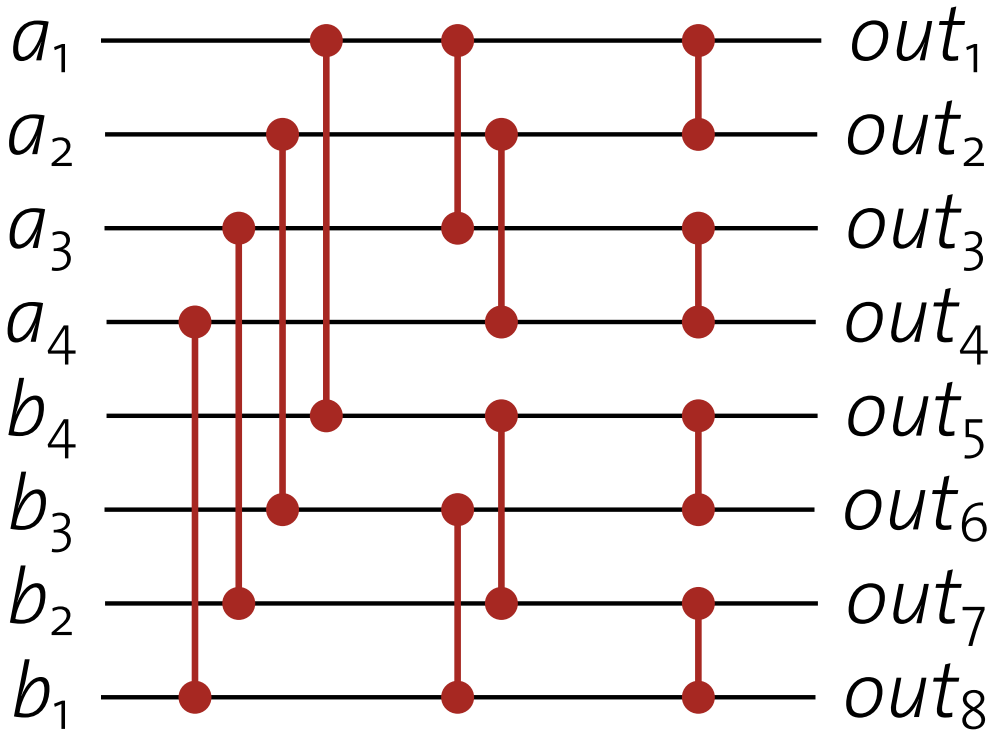
Run generation



- 10 min/max, 8 shuffle, 8 load/store
- 64 bytes in, 64 bytes out (128-bit SIMD)

Merging

Two sorted runs, four items each:



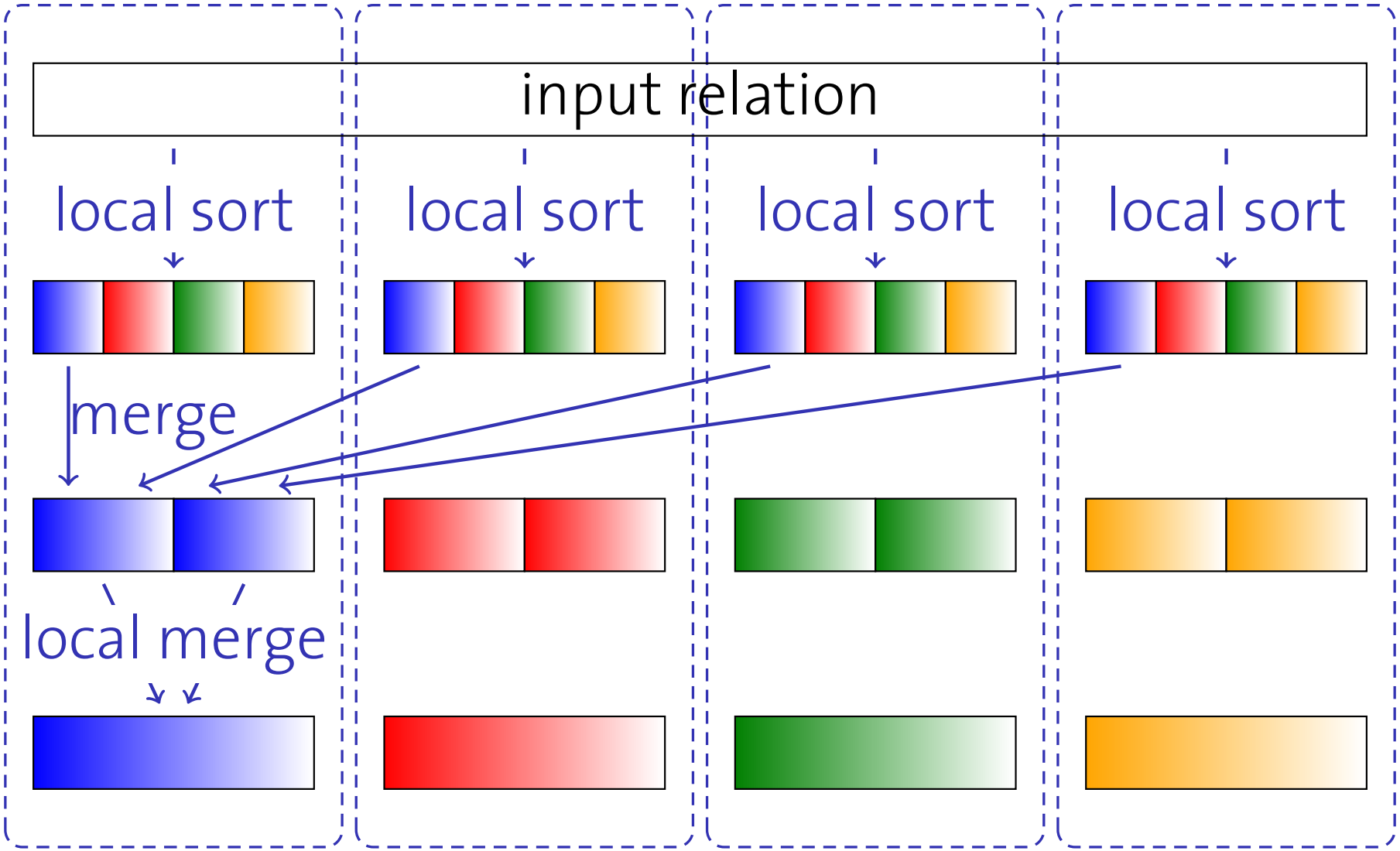
- Input: two SIMD registers **a** and **b**, sorted
- 6 min/max, 10 shuffle, 4 load/store

1. Load SIMD set from both runs in registers **a** and **b**.
2. Perform **SIMD merge** of **a** and **b** ($\rightarrow$ result in [**a**, **b**]).
3. **Write a** to output.
4. **Fetch next** SIMD set from run **where head is smaller**; replace **a**.
5. Goto 2 while there is still input to process.

E.g., 

- run 1: 3, 7, 14, 29, 37, 48, 52, 67, 69, 74, 89, 91
- run 2: 9, 11, 16, 21, 25, 39, 46, 71, 79, 86, 88, 95
- output: 3, 7, 9, 11, 14, 16, 21, 25, 29, 37, 39, 46, 48, 52, 67, 69, 71, 74, 79, 86, 88, 89, 91, 95

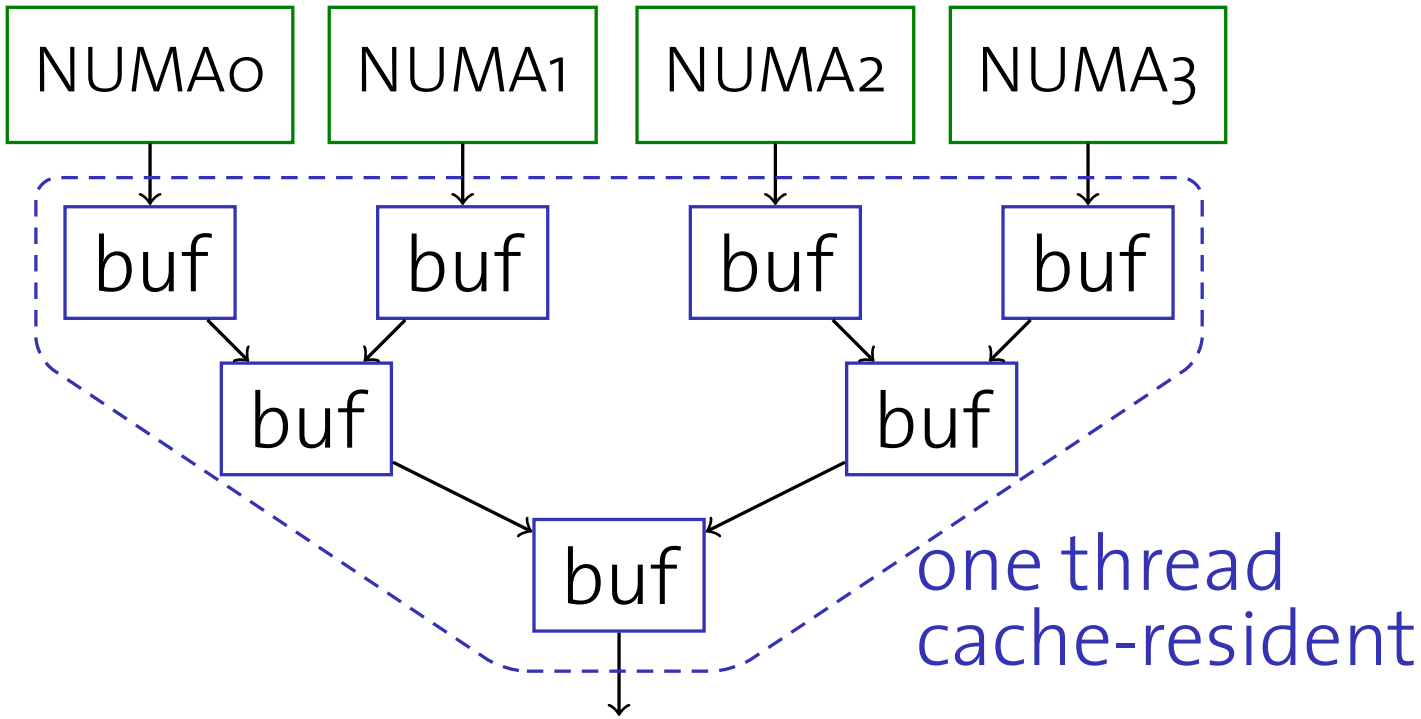
Sorting and NUMA



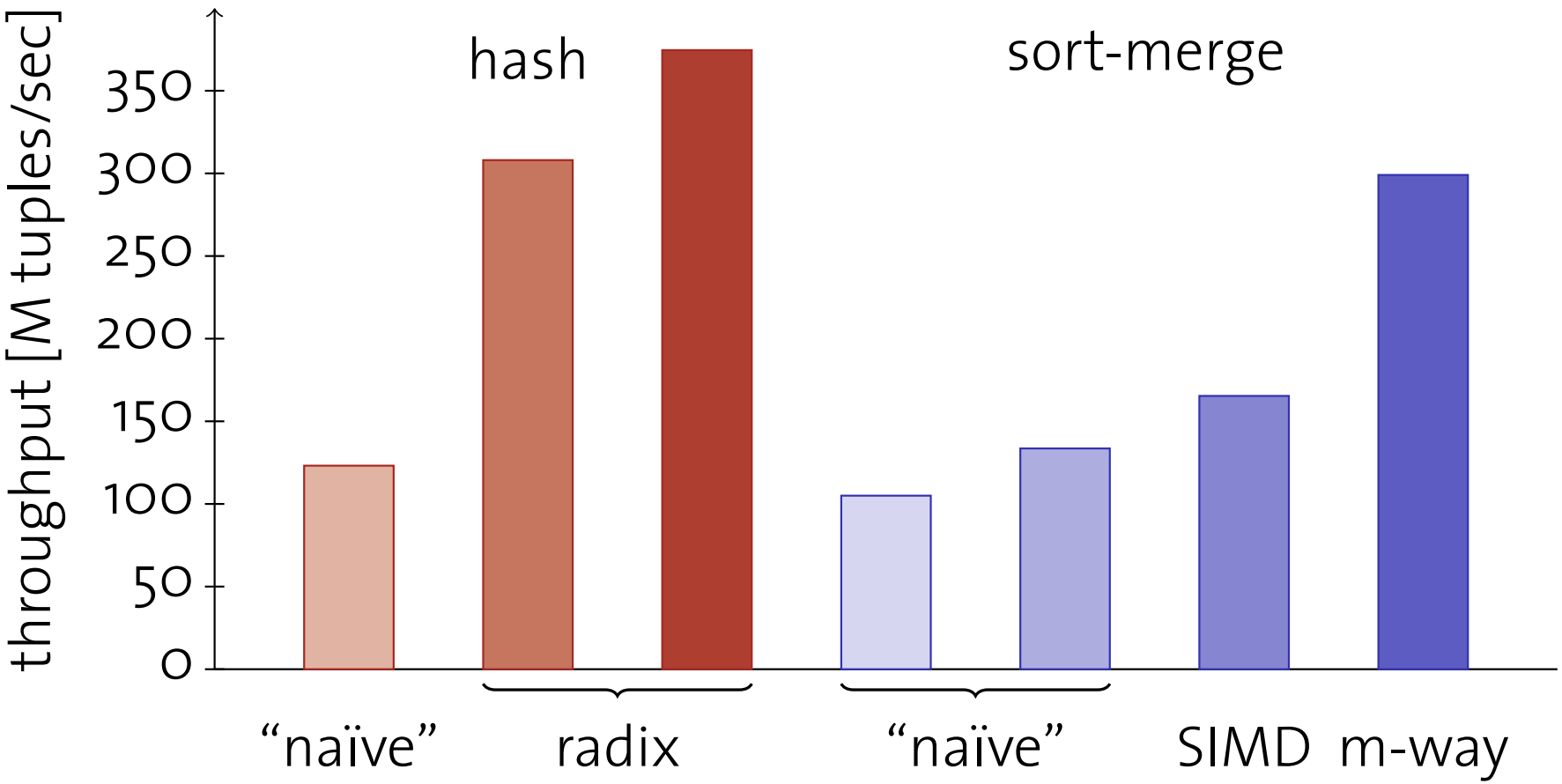
Problem: Merging is **bandwidth-bound**.

→ Merge multiple runs (from NUMA regions) at once

→ Might need **more instructions**, but brings bandwidth and compute **into balance**.



Sorting vs. Hashing

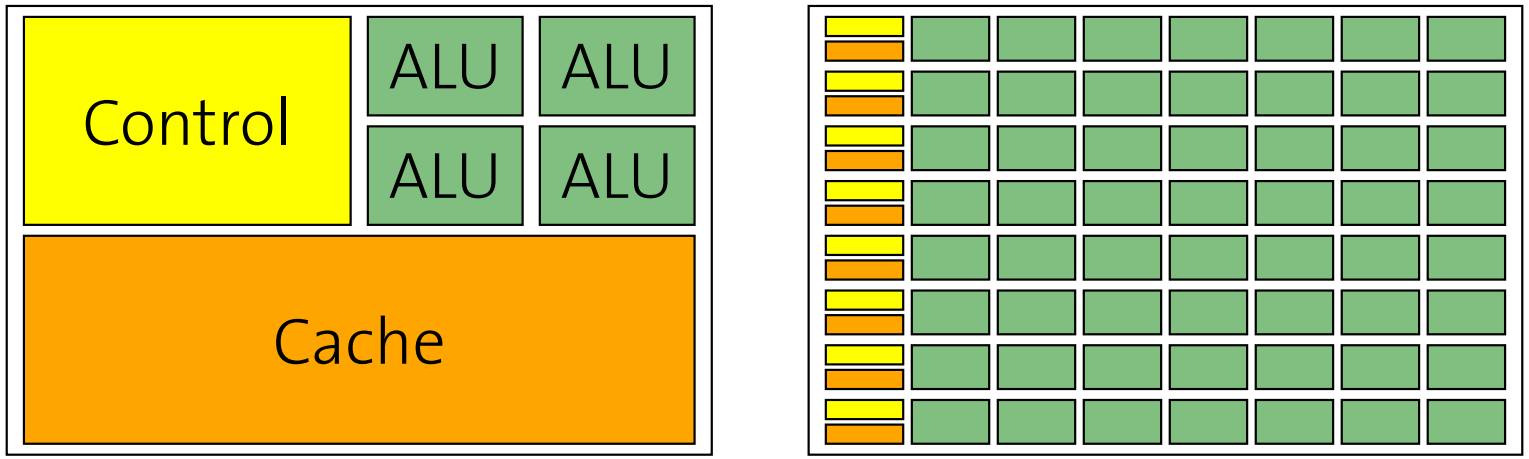


- 12.8 GB ⋈ 12.8 GB
- Intel E5-4640 ("Sandy Bridge"), 2.4 GHz, 32 cores

Part III

Graphics Processors (GPUs)

Graphics Processors (GPUs) ↔ CPUs



CPU: Optimize for **instruction latency** (→ control logic and caches)

- Decreasing die share performs actual work (ALUs).

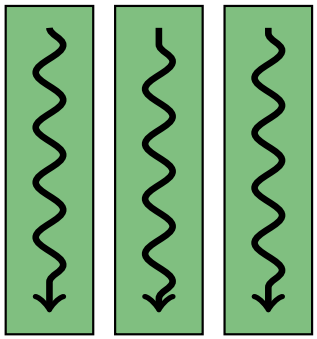
GPU: Use chip space to perform work, not for infrastructure

- Simple logic, massive parallelism; optimize for **throughput**.

Parallelism

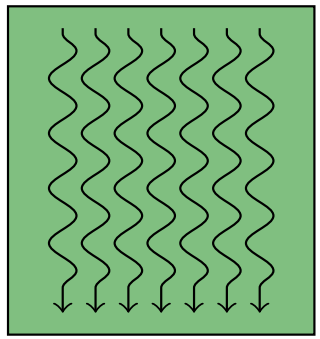
CPU: task parallelism

- heavyweight threads
- 10s of threads, 10s of cores
- threads managed explicitly
- threads run different code



GPU: data parallelism

- lightweight threads
- 10,000s of threads, 100s of cores
- scheduled in batches
- all threads run same code
  - → SPMD, single program, multiple data



High-Degree Parallelism

Rationale for high-degree parallelism:

Don't try to **reduce** latency, but **hide** it.

→ While a thread is waiting for memory, execute other threads to hide that latency.

→ Hardware **thread scheduling** (simple, in-order).

→ Schedule in **batches** ("warps") to reduce hardware cost.

Scheduling in Batches

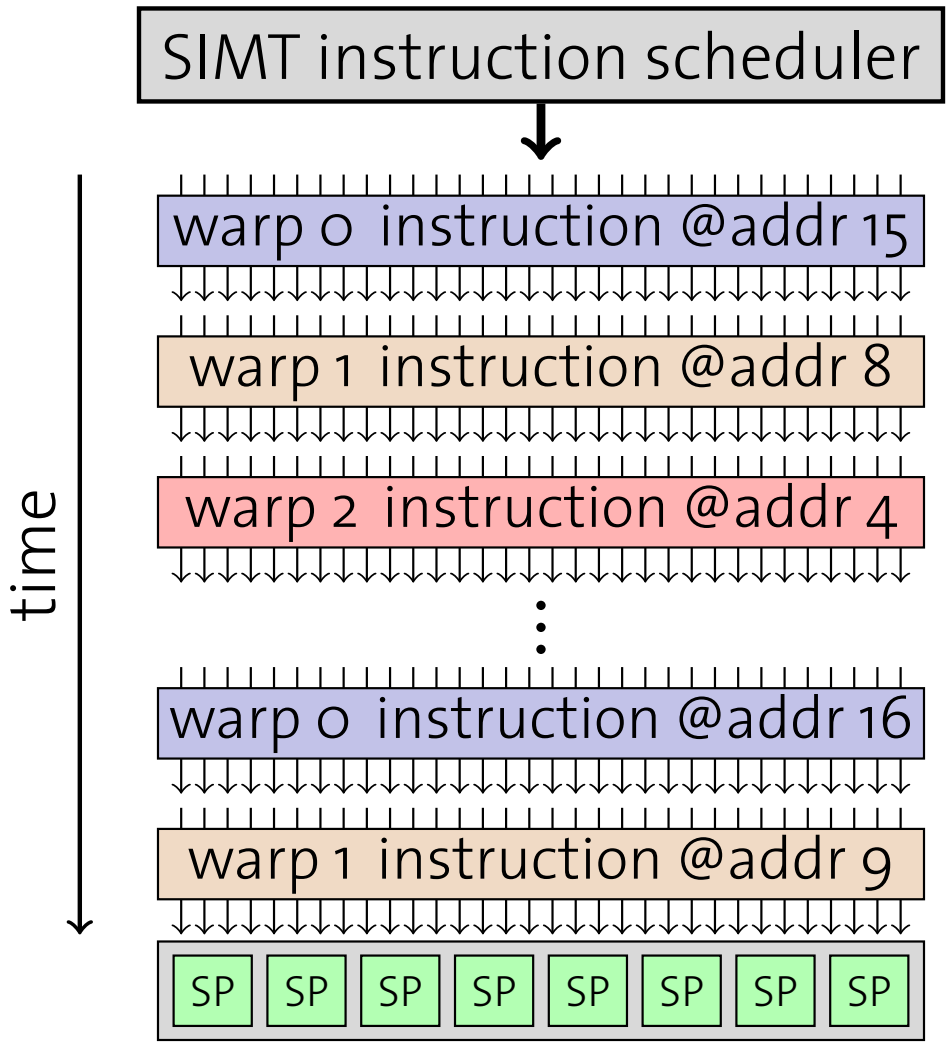
Threads are scheduled in units of 32, called **warps**.

- → **Warp:** Set of 32 threads that run identical code and start at same program address.
- → **SIMT:** Single Instruction Multiple Threads.
- → *e.g.*, NVIDIA Kepler: up to 15×64 warps $= 30$ k threads

warp (dt. Kett- oder Längsfaden)

- → Scoreboard tracks which warps are ready to execute.

SPMD / SIMT Processing



- **All** threads in one warp execute the **same instruction**.
- At each time step scheduler selects warp ready to execute (*i.e.*, all its data are available).
- Scheduling decided at **instruction level**.
- NVIDIA Fermi: dual issue; Kepler: quad issue.

branch divergence

Warps and Latency Hiding

Some runtime characteristics (CUDA 1.3):

- Issuing a warp instruction takes **4 cycles**.
- Register write-read latency: **24 cycles**.
- Global (off-chip) memory access: $\approx$ **400 cycles**.

Threads are executed **in-order**.

$\rightarrow$ **Hide latencies** by executing other warps when one is paused.

$\rightarrow$ Need **enough warps** to fully hide latency.

E.g.,

- Need $24/4 = 6$ warps to hide register dependency latency.
- Need $400/4 = 100$ instructions to hide memory access latency. If every 8th instruction is a memory access, $100/8 \approx 13$ warps would be enough.

NVIDIA Kepler Architecture

source: NVIDIA Kepler GK110 White Paper

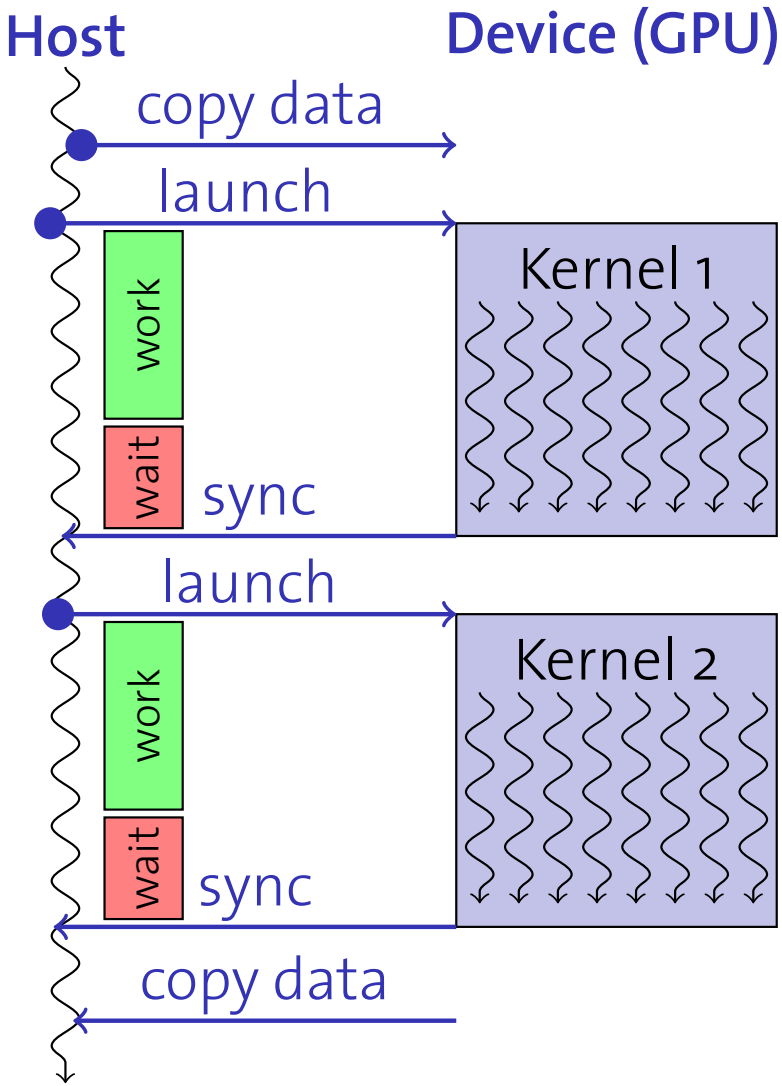
NVIDIA Streaming Multiprocessors (SMX)

source: NVIDIA Kepler GK110 White Paper

NVIDIA Kepler:

- 15 SMX per chip
- 192 "cores" per SMX (≙ ALU; integer and single-precision float)
- 64 double-precision units
- 32 "special function units" (sine, cosine, etc.)
- issue four warps, two instructions per warp

Computation Model (OpenCL)



- Host system and **co-processor** (GPU is only one possible co-processor.)
- Host triggers
  - data copying host ↔ co-processor,
  - invocations of **compute kernels**.
- Host interface: **command queue**.

Processing Model: (Massive) Data Parallelism

A **traditional loop**

```
for (i = 0; i < nitems; i++)
    do_something(i);
```

becomes a **data parallel kernel invocation** in OpenCL ($\rightsquigarrow$ `map`):

```
status = clEnqueueNDRangeKernel (
           commandQueue,
           do_something_kernel, ..., &nitems, ...);
```

```
__kernel void do_something_kernel(...)  {
    int i = get_global_id(0);
    ...;
}
```

Compute Kernels

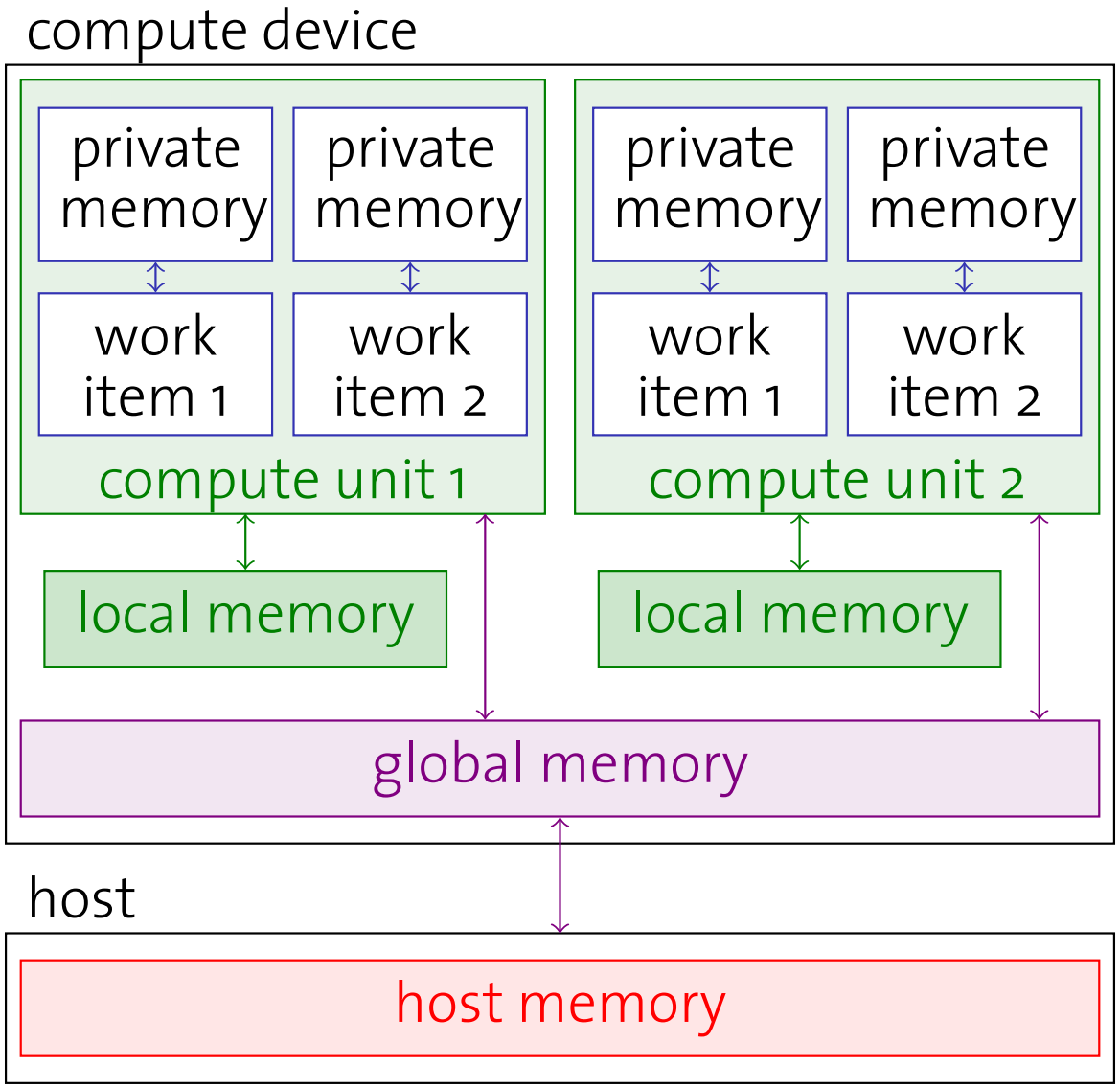
OpenCL defines a **C99-like** language for compute kernels.

- Compiled **at runtime** to particular core type.
- Additional set of built-in functions:
  - Context (*e.g.*, `get_global_id()`), math routines, …

```
__kernel void square(__global float *in,
                     __global float *out)
{
    int i = get_global_id(0);
    out[i] = in[i] * in[i];
}
```

- Very limited **thread interaction** (eases parallel execution)

OpenCL Memory Model



Part IV

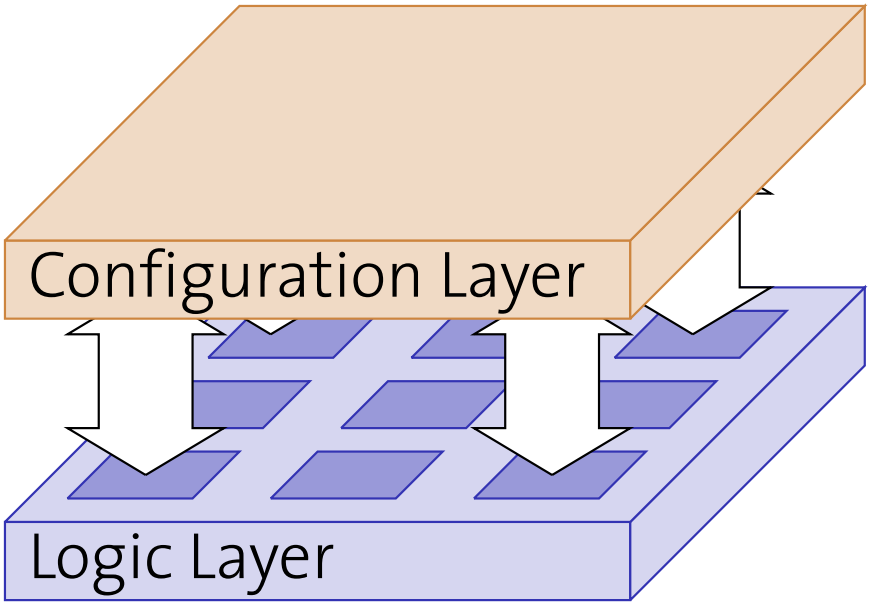
Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs)

- **Array** of logic **gates**
- Functionality fully **programmable**
- Re-programmable after deployment ("in the **field**")

- Technology already invented in the 80s
- Today's **chip sizes** allow designs of serious complexity
- Projected FPGA revenue in 2013: USD 3.5 billion

Reconfigurable Hardware



Configuration Layer:

- Configuration, stored in **SRAM**.

Logic Layer:

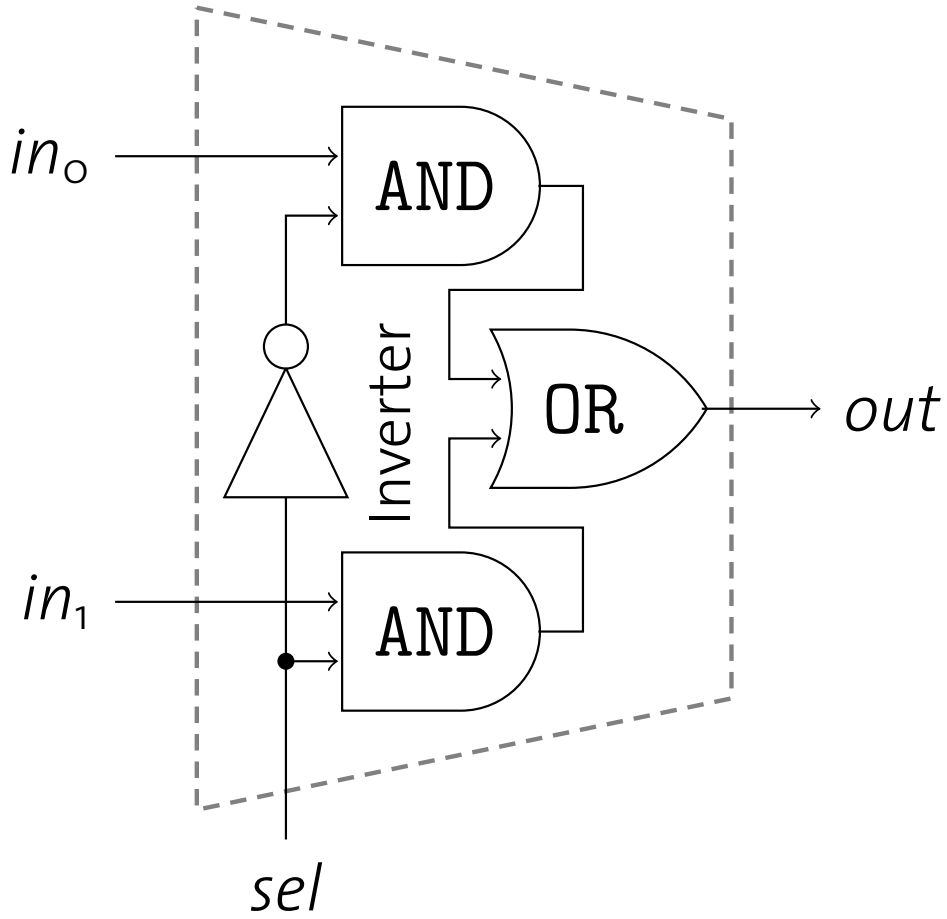
- Actual hardware logic (LUTs and flip-flops)

→ **Reconfiguration ≡ SRAM update**

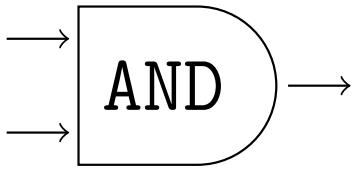
Hardware Circuits

Electronic circuits consist of three fundamental ingredients:

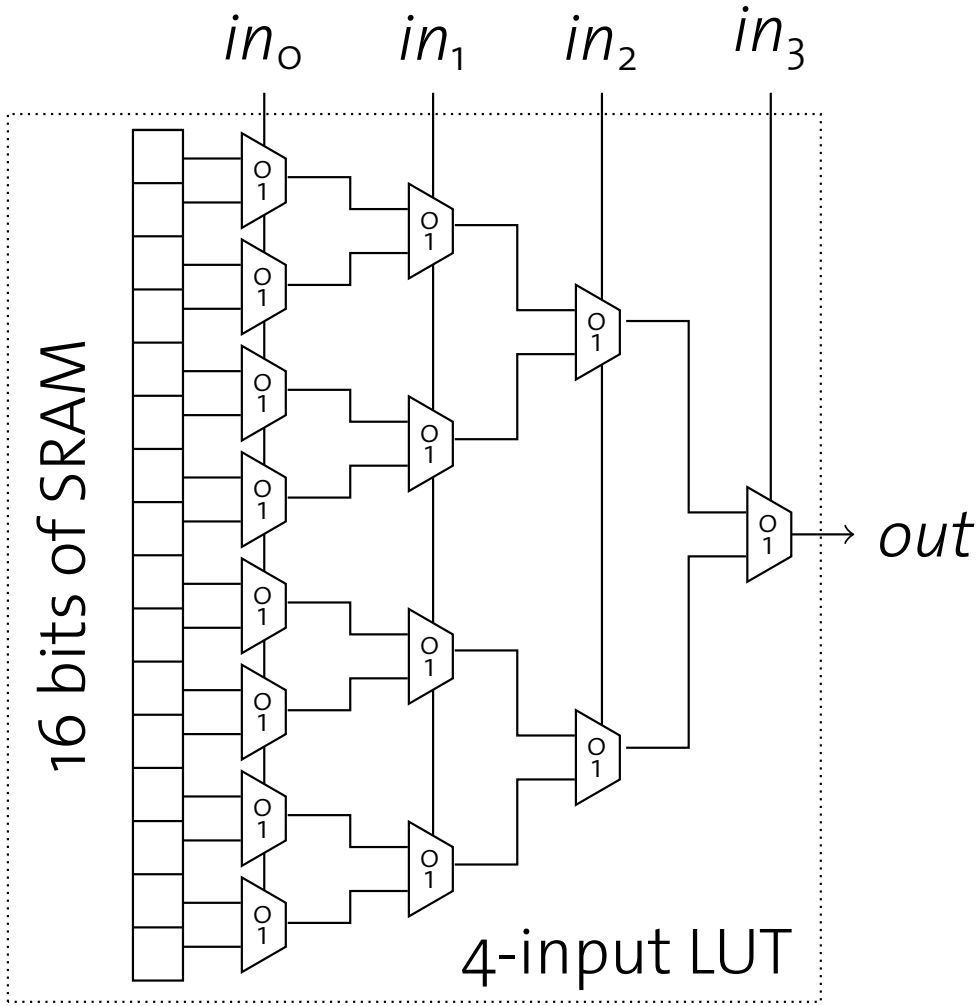
- combinational logic (gates)
- memory elements
- wiring (interconnect)



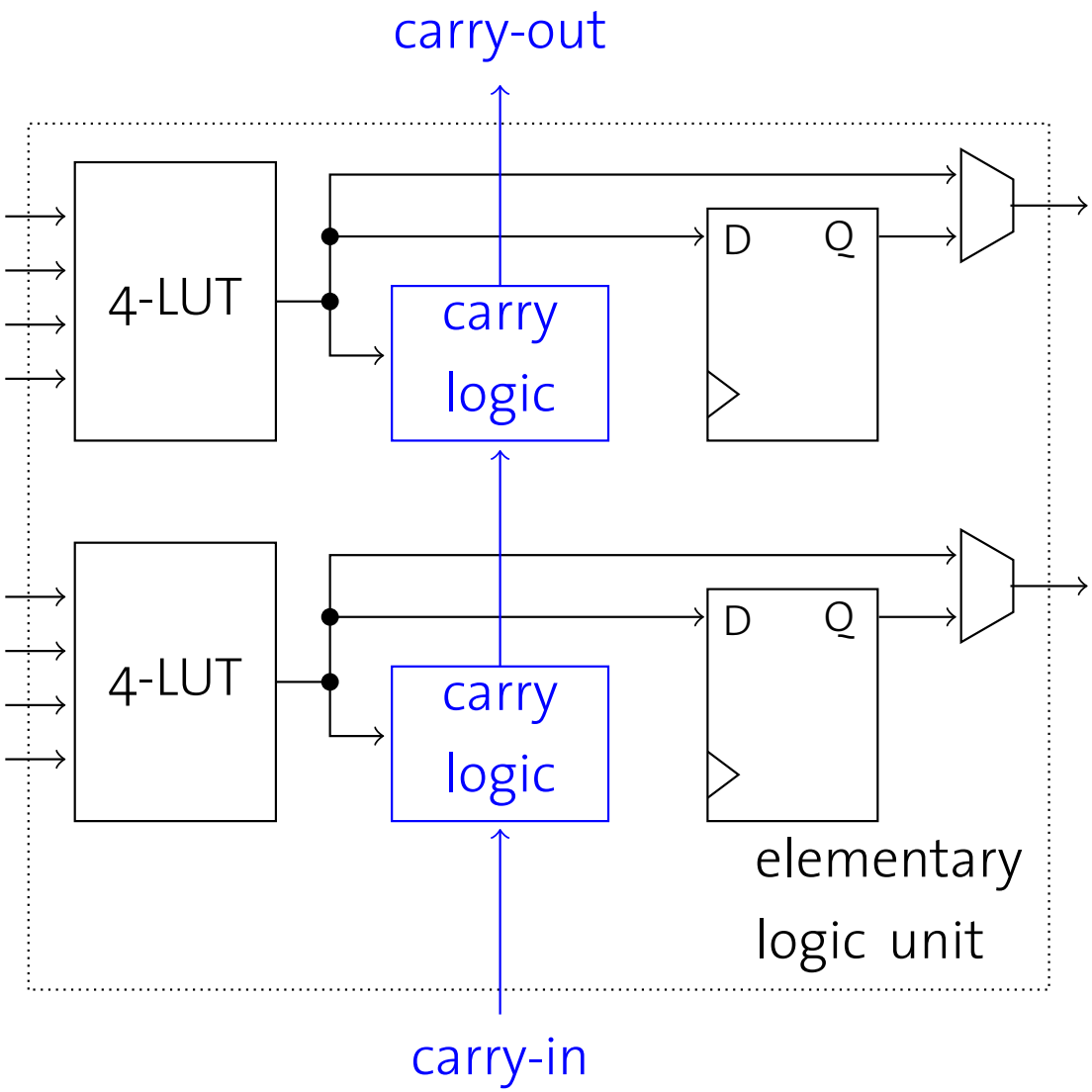
Reprogrammable Logic : Lookup Tables



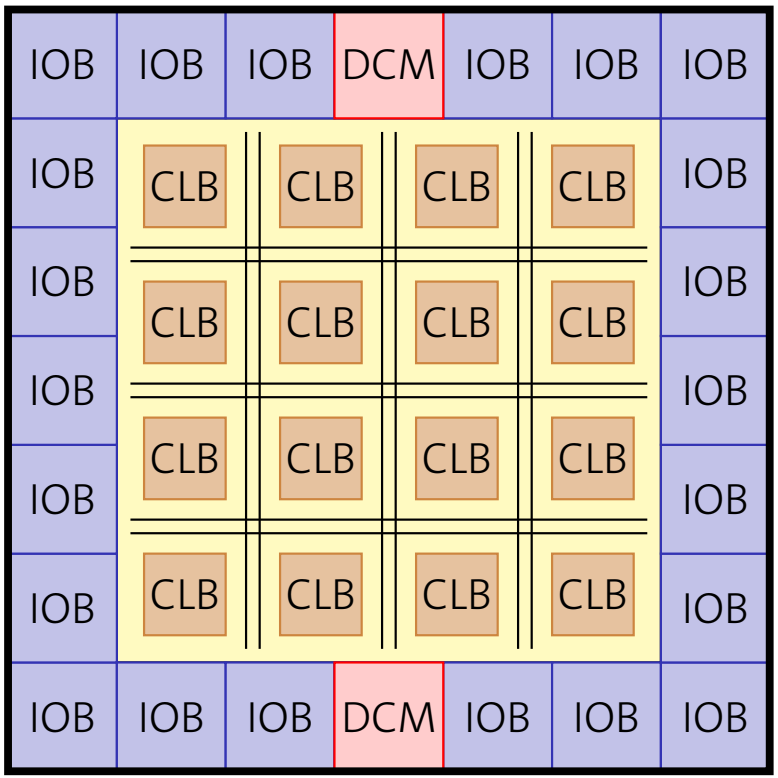
Input	Output
00	0
01	0
10	0
11	1



Elementary Logic Unit (Slice)



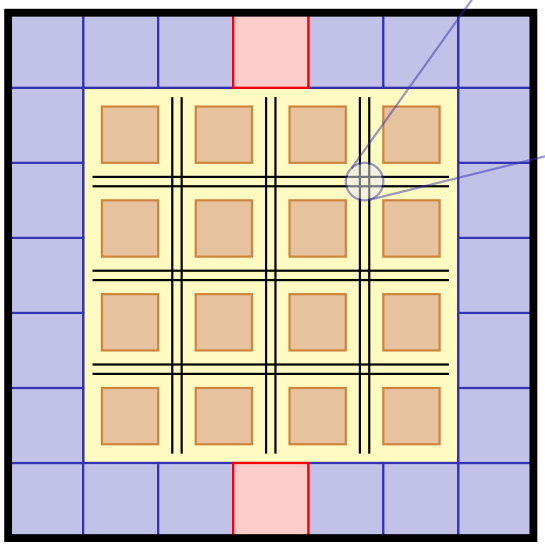
Basic FPGA Architecture

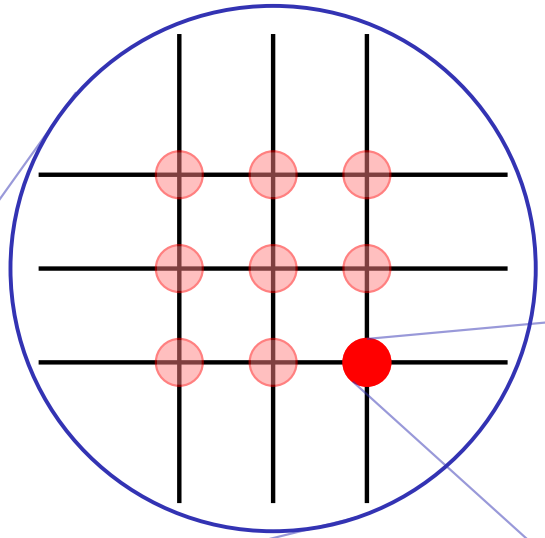


- chip layout: 2D array
- Components
  - CLB: Configurable Logic Block (collection of slices)
  - IOB: Input/Output Block
  - DCM: Digital Clock Manager
- Interconnect Network
  - signal lines
  - configurable switch boxes

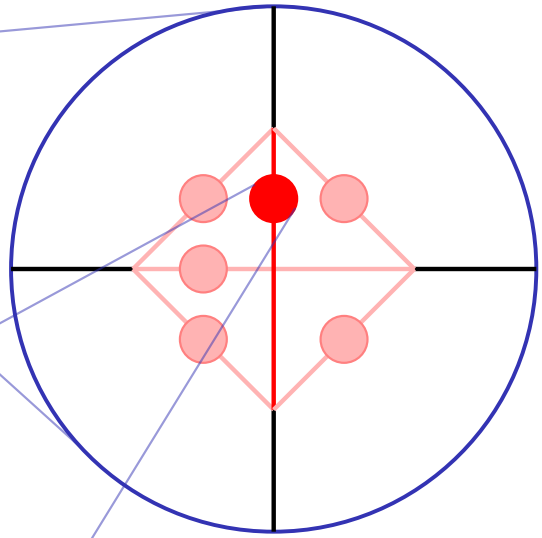
Configurable Wires (Interconnect)

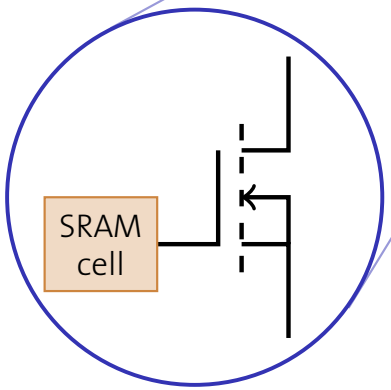
programmable Switch Box and bundle of lines





programmable intersection point

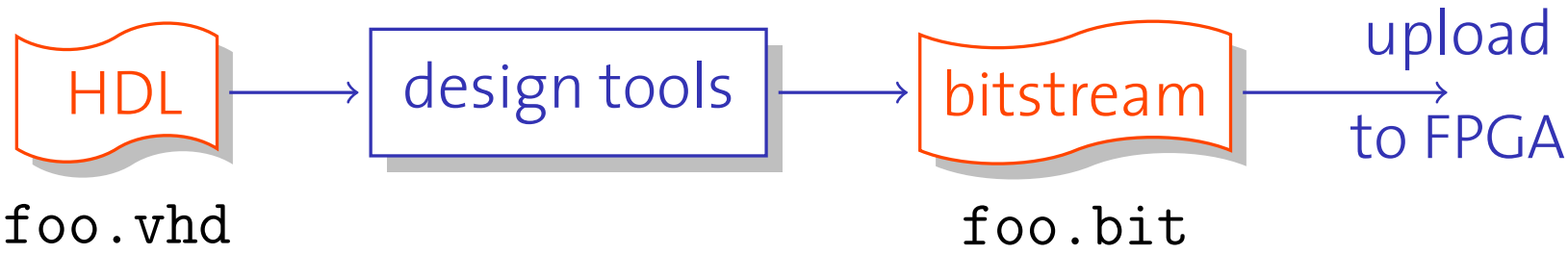




programmable switch with memory cell

Programming FPGAs

- FPGA reconfiguration ≡ SRAM update
- Generate new SRAM content (as a **"bitstream"**) using design tools.
- Input: high-level **circuit description**
- Typically: using a **hardware description language (HDL)**
  - Verilog
  - VHDL

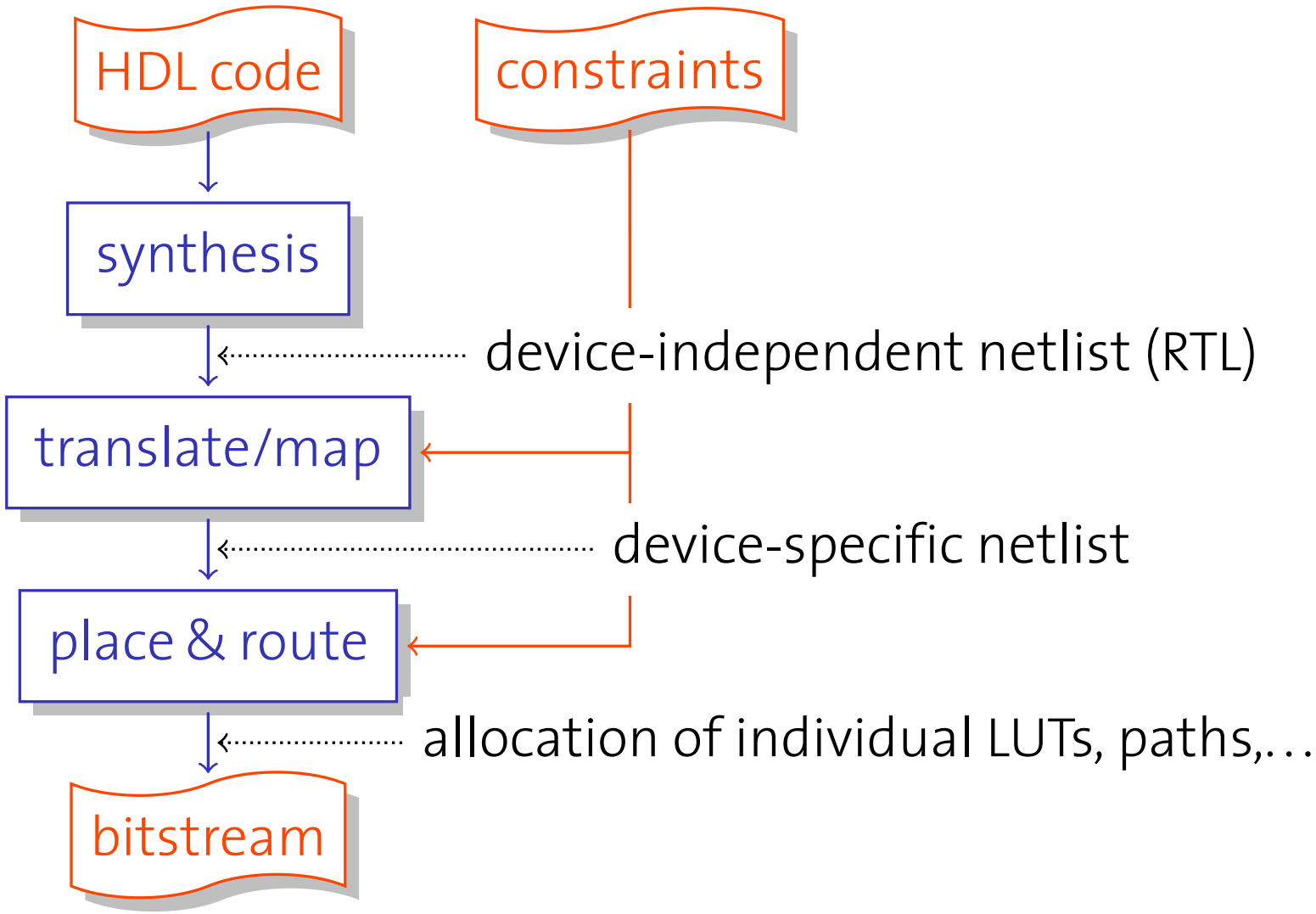


Example: VHDL

```
architecture Behavioral of compare is
begin
  process (A, B)
  begin
    if ( A = B ) then
      C <= '1';
    else
      C <= '0';
    end if;
  end process;
end Behavioral;
```

This is **not** a sequential program!

FPGA Design Flow



FPGA Design Cost

Notes:

- The FPGA design flow is **heavily compute-intensive**
  - Think of minutes, even hours
  - Cost increases dramatically with design size
  - Full circuit re-compilation is something you'll want to do **off-line** only
- **Device reconfiguration** is faster
  - After all, it's changing a few bits in SRAM only
  - Think of milli-seconds (however, current hardware is not optimized for fast re-configuration)

Circuit Simulation

Circuits can be **simulated in software**:

- **cycle-accurate** simulation
- at **any design stage** ("behavioral" vs. "post-routing" simulation)

In practice, you rarely need a physical device even

What To Use FPGAs For

FPGAs are good at:

- **massive throughput**
  - → leverage high pin count
- **data flow-style processing**
  - → data "flows through chip," flows and tasks map naturally to wires and components
- meeting **tight performance guarantees**
  - → Often, the performance of a circuit is fully predictable.
  - → important, *e.g.*, for **real-time tasks**
- regular expressions, **state machines**
  - → FPGA ≡ generic hardware state machine

What NOT To Use FPGAs For

FPGAs are not so good at:

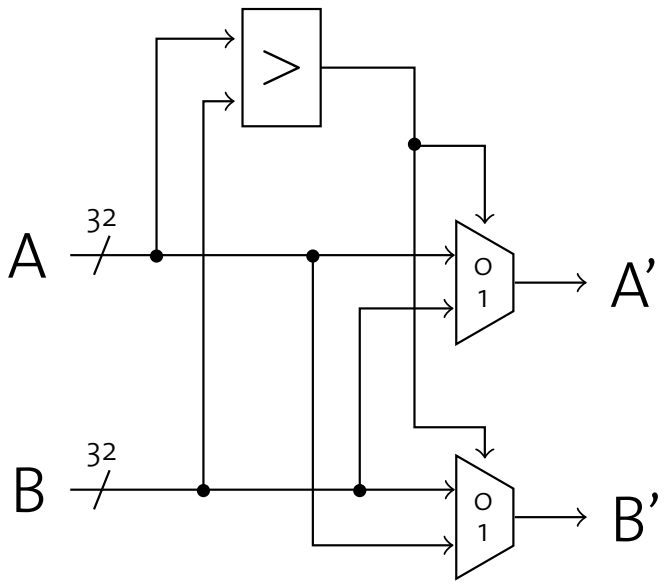
- **floating point operations**
  - → floating point requires lots of chip space
  - → Use a GPU if you really need floating point.
- **branching** and **runtime flexibility**
  - → low clock speed makes runtime decisions rather slow
- likewise: **complex and long algorithms**
  - → If you need a full-fledged instruction set processor, use an instruction set processor.

Use Case: Sorting with FPGAs

- Sorting Networks (Revisited) [Mueller *et al.* 2012]
  - High-througput sorting for small working sets
  - Data parallelism
  - Pipeline parallism
- FIFO-based Merge Sort [Koch *et al.* 2010]
  - Using embedded RAM blocks for larger problems
- External Large Problem Sorting [Koch *et al.* 2010]
  - Resorting to DRAM for even larger problems

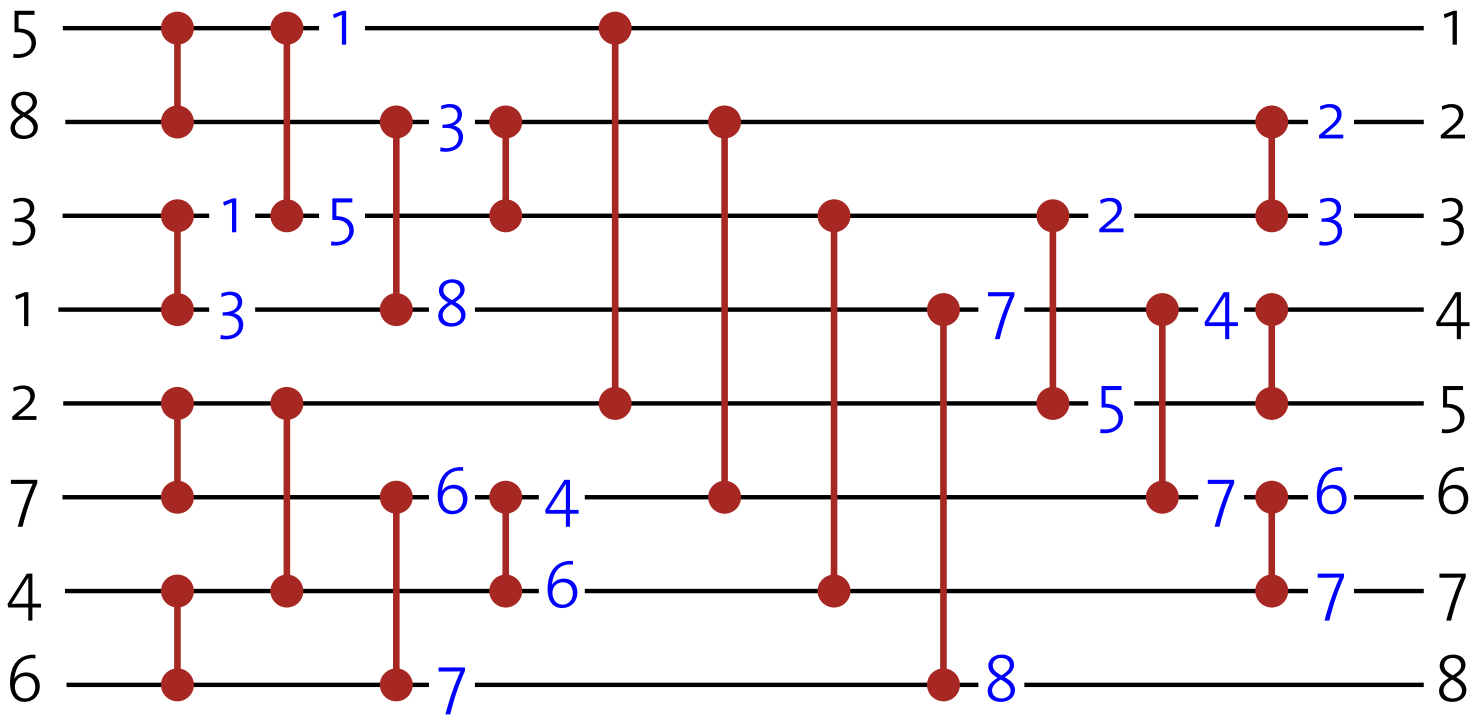
Compare-and-Swap Element

- The *compare-and-swap* element is the basic building block of hardware sorting networks
- It consists of a comparator circuit and two wide multiplexers



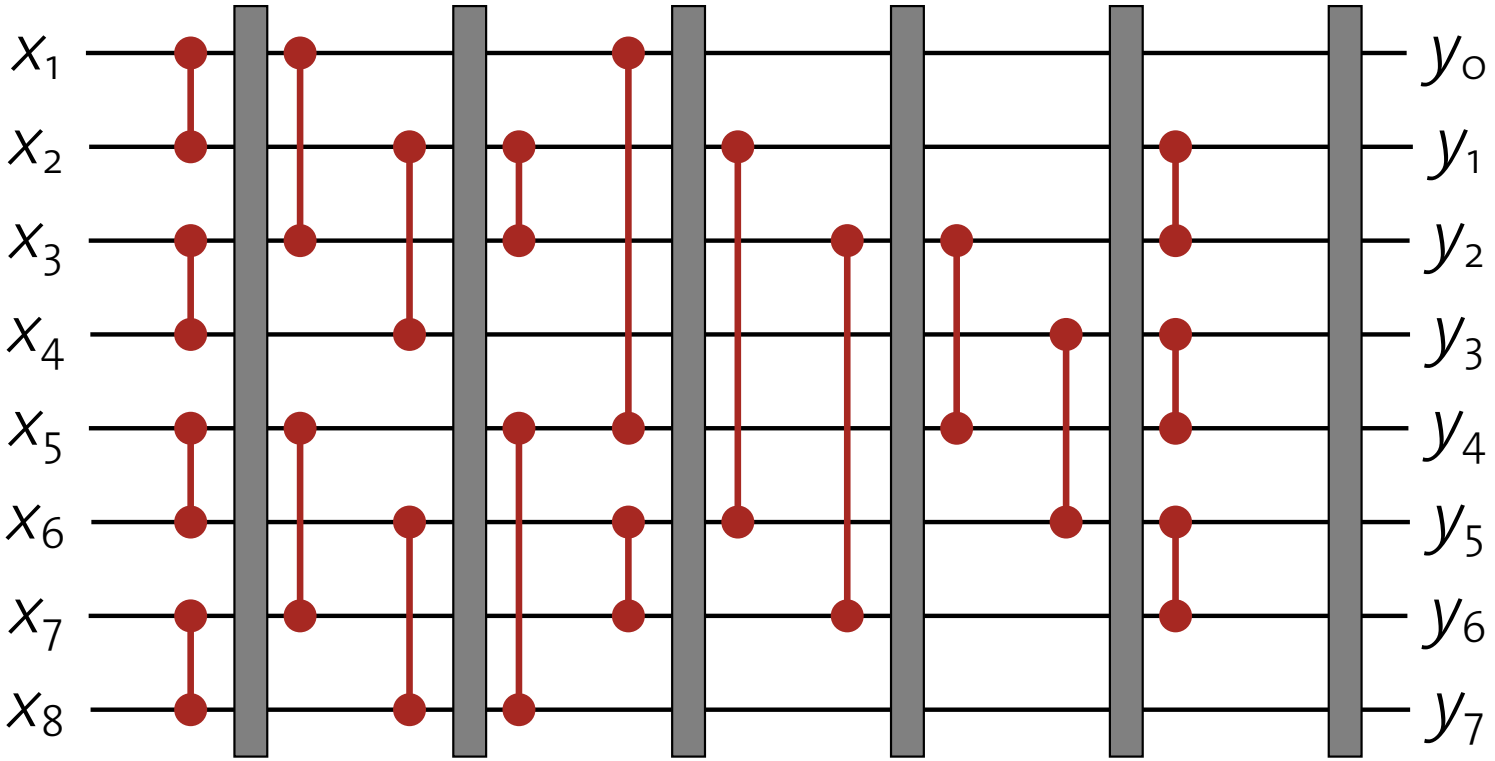
Even-odd Sorting Network

- Sorts eight values using 19 *compare-and-swap* elements



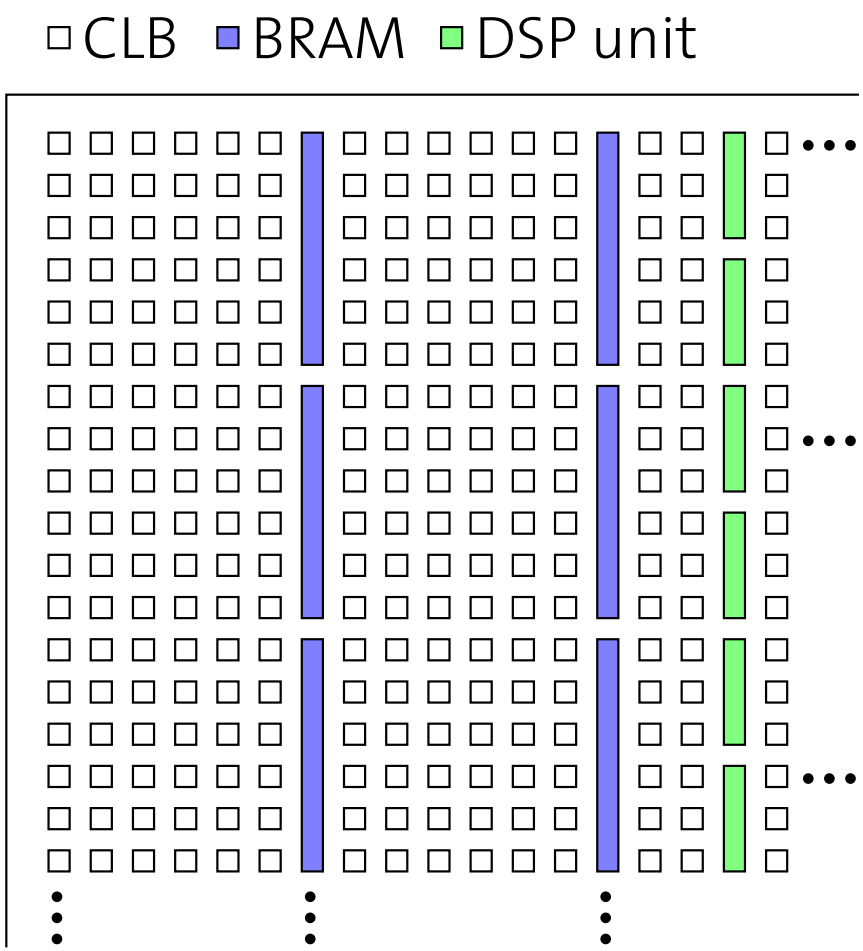
Pipeline Parallelism

- Longest signal path via six *compare-and-swap* elements



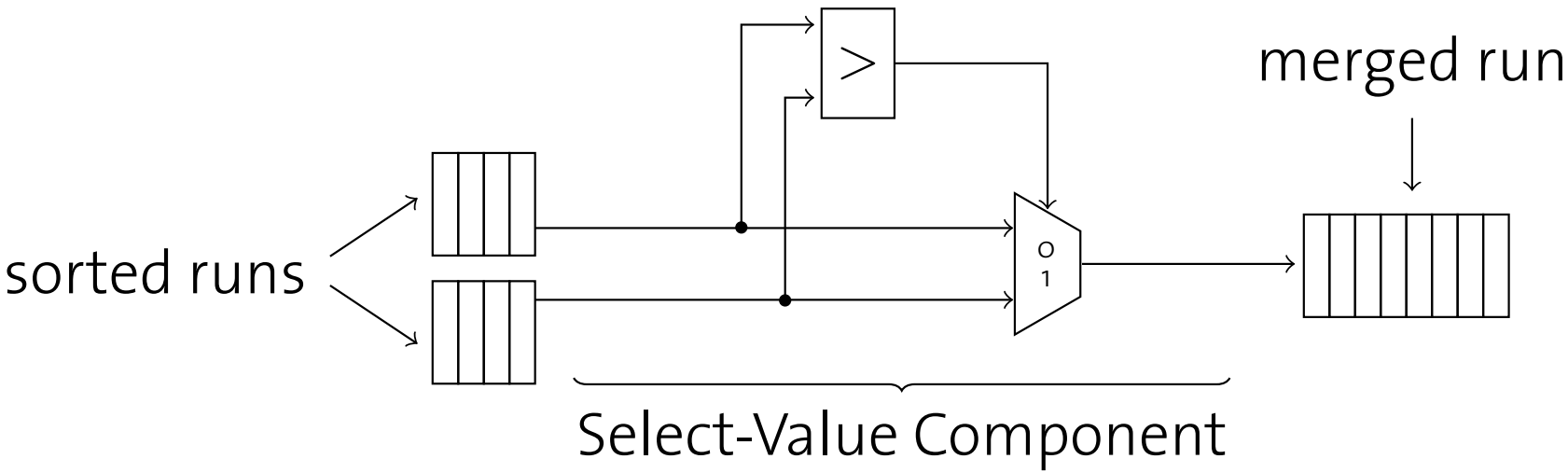
- Pipelined version $\rightarrow f_{clk} = 267\,MHz, 8 \times 32\,bit \rightarrow 8.5\,GB/s$

Sorting Larger Working Sets

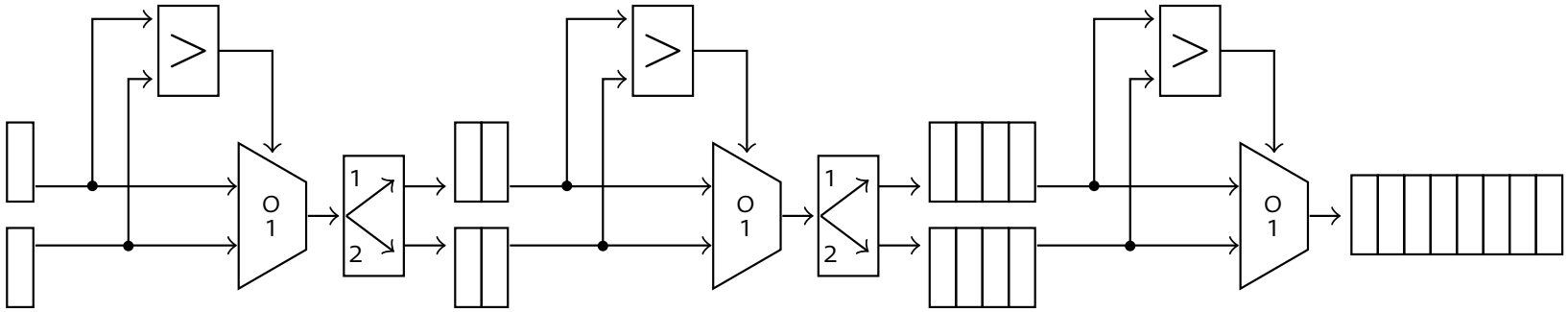


- BRAM = fast embedded RAM blocks (~ 4 KB)
- Programmable size and word width
- Dual-ported
- Can be configured as FIFO queues

FIFO-based Merge Sorter



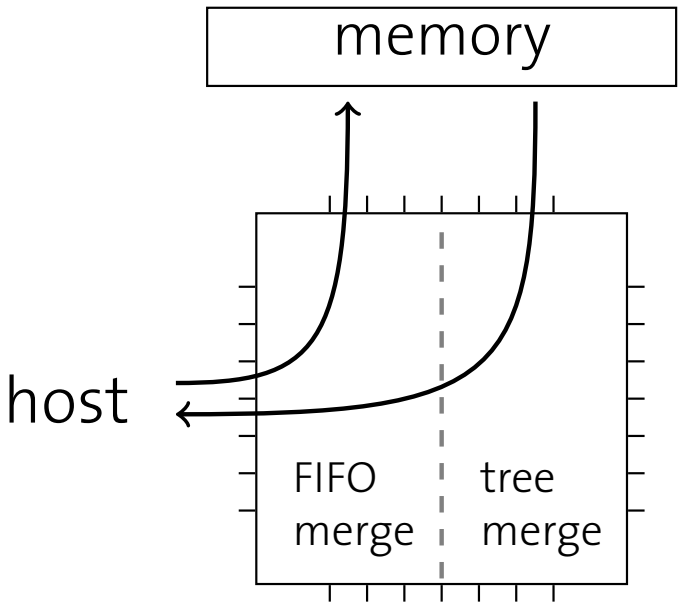
Cascade of FIFO-based Merge Sorters

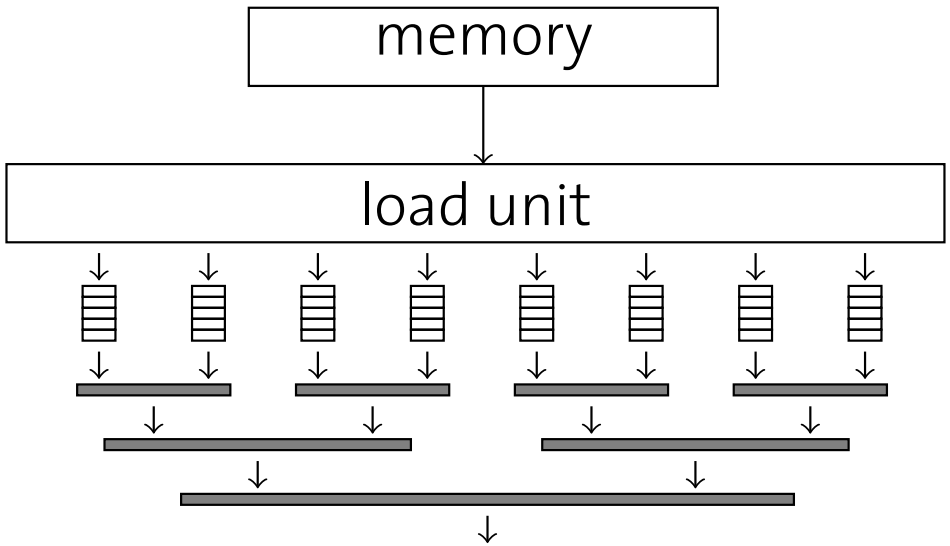


- Processing at each stage can start once first FIFO is filled
- Only one FIFO is read per cycle at each stage
- BRAM-based FIFOs allow simultaneous reading and writing
  - one FIFO should be enough
  - need to be able to read from different positions in FIFO
  - when done right $\rightarrow$ streaming is possible
  - problem size $\sim$ 40*K* 64-bit keys $\rightarrow$ 2 GB/s [D. Koch et. al.]

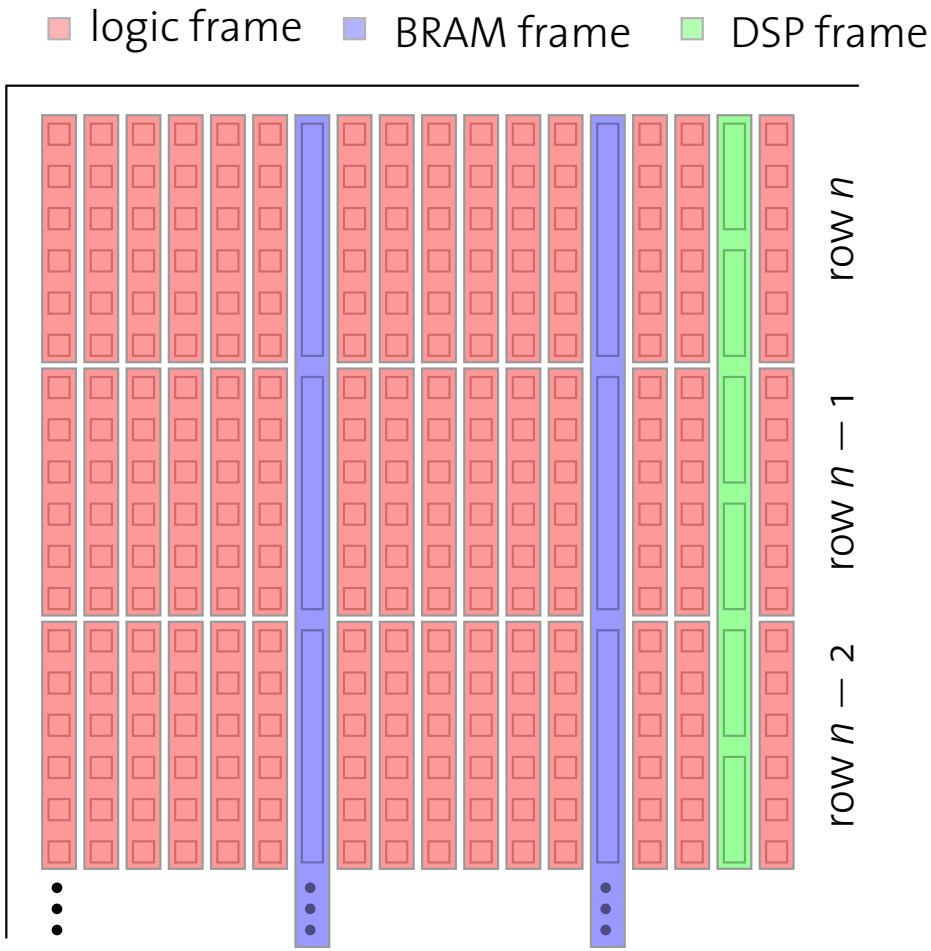
Merge Sorter Tree

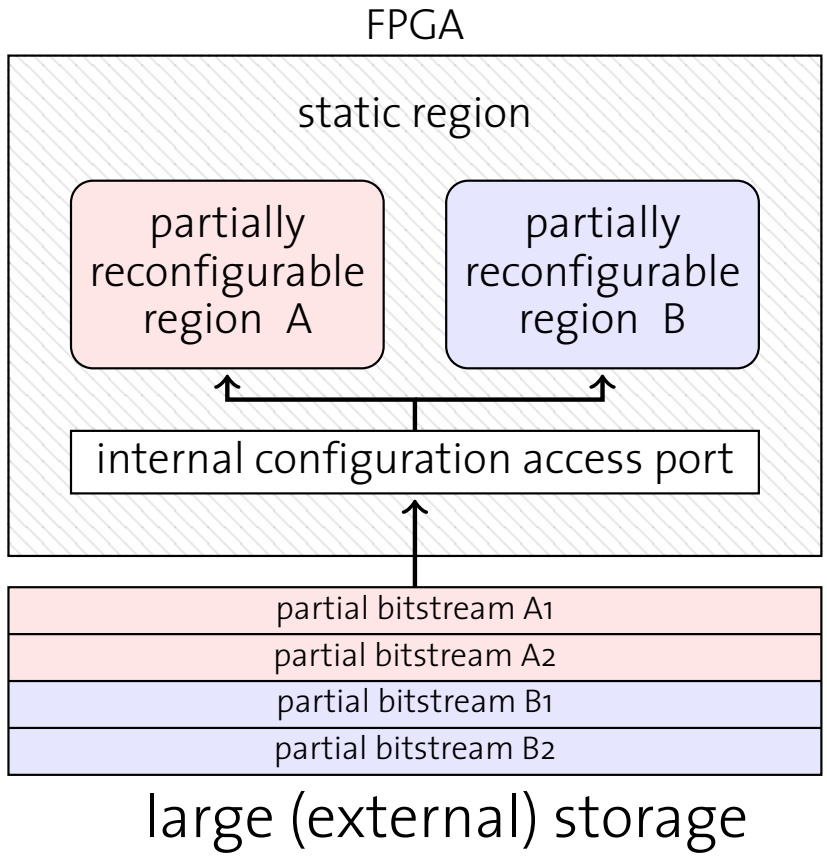
- What if problem size exeeds BRAM capacity?
- For larger problems we can resort to external memory
- Merge sorter tree using the same *select-value component*



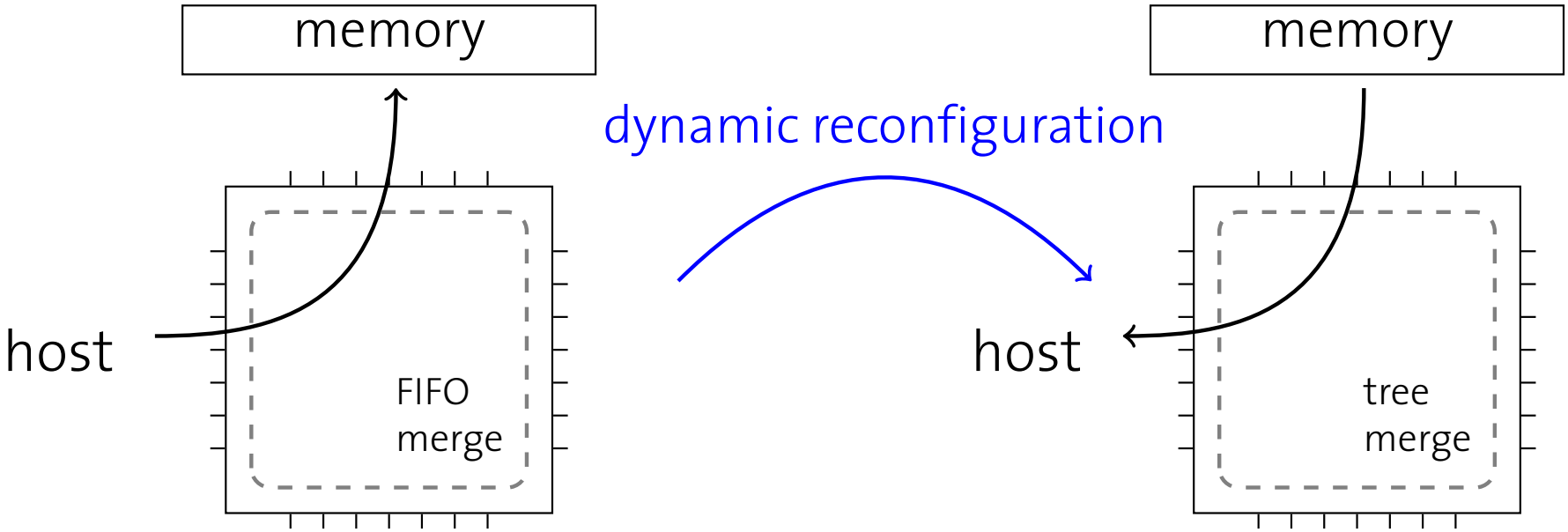


Partial Reconfiguration





Sorting with Partial Reconfiguration



- Max. reconfiguration speed = 400 MB/s
- Reconfiguration data (here) = 3 MB
- Reconfiguration cost ≡ sorting 15 MB (2 GB/s)
- Trade-off: larger problems favor dynamic reconfiguraiton!

Part V

Summary

Summary

Hardware technology is hitting limits.

- **Frequency scaling** halted years ago.
- **Multi-Core scaling** not sustainable either (power!)

Specialize to further benefit from **Moore's Law**:

→ Leverage **parallelism** and **locality**.

→ **Hardware/software co-design**

Moore's Law?

- **Might** slow down for **economic reasons** (but not yet).

Summary

Today:

1. Modern **Multi-Core Systems**
   - Leverage **parallelism** (SIMD, multi-core)
   - Preserve **locality** (cache awareness, NUMA)
2. **Graphics Processors (GPUs)**
   - **Throughput** instead of instruction latency
   - Restricted form of **parallelism** ($\rightsquigarrow$ locality)
3. **Field-Programmable Gate Arrays (FPGAs)**
   - **Tailor-made hardware**, re-configure at runtime
   - **Low frequency** ($\rightsquigarrow$ low power); **high bandwidth**

Advertisement

Interested in these topics?

- I'm hiring **PhD students**
- Contact me:

  Jens Teubner, `jens.teubner@cs.tu-dortmund.de`

technische universität
dortmund


Cagri Balkesen, Jens Teubner, and Gustavo Alonso.
Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware.
In *Proc. of the 29th IEEE Conf. on Data Engineering (ICDE)*, Brisbane, Australia, April 2013.


Spyros Blanas, Yinan Li, and Jignesh M. Patel.
Design and evaluation of main memory hash join algorithms for multi-core CPUs.
In *Proc. of the 2011 ACM SIGMOD Conf. on Management of Data*, pages 37–48, Athens, Greece, May 2011.


Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien. Yu, V. Leo Rideout, Ernest Bassous, , and Andre R. LeBlanc.
Design of ion-implanted MOSFETS with very small physical dimensions.
IEEE Journal of Solid State Circuits, SC-9(5):256–268, October 1974.


Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger.
Power challenges may end the multicore era.
Communications of the ACM, 56(2):93–102, 2013.


Dirk Koch and Jim Torresen.
FPGASort: A high-performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting.
In *Proc. of the 19th ACM SIGDA Int'l Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 45–54, Monterey, CA, USA, February 2011.


Gordon E. Moore.
Cramming more components into integrated circuits.
Electronics Magazine, 38(8), April 1965.


René Müller, Jens Teubner, and Gustavo Alonso.
Sorting networks on FPGAs.

The VLDB Journal, 21(1), February 2012.

Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton.
Cache conscious algorithms for relational query processing.
In *Proc. of the 20th Int'l Conf. on Management of Data (VLDB)*, pages 510–521, Santiago, Chile, September 1994.