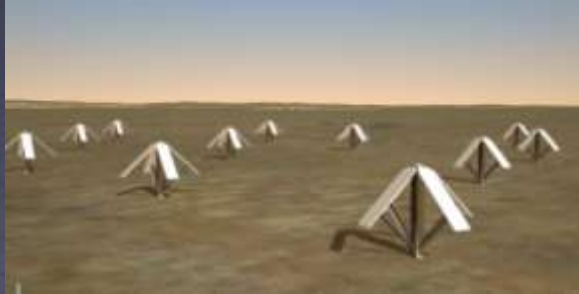


Memory-Driven Near-Data Acceleration

and its application to DOME/SKA

Jan van Lunteren
Heiner Giefers
Christoph Hagleitner
Rik Jongerius



~0.25M Antennae 0.07-0.45GHz



~0.25M Antennae 0.5-1.7GHz



~3000 Dishes 3-10GHz

Square Kilometer Array (SKA)

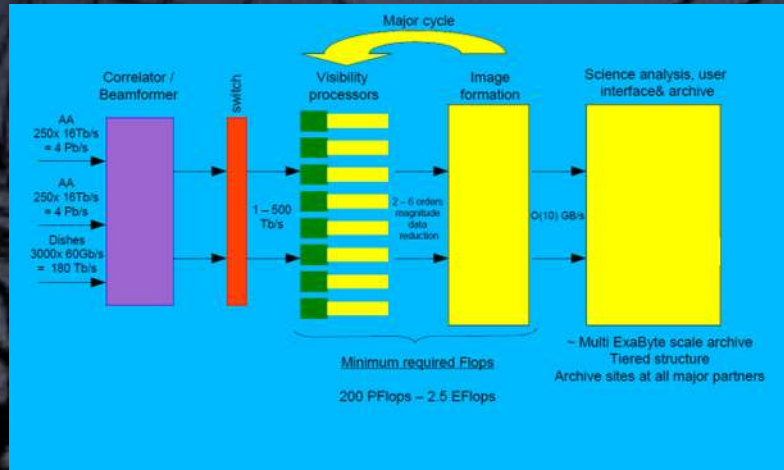
- The world's largest and most sensitive radio telescope
- Co-located in South Africa and Australia
 - deployment SKA-1: 2020
 - deployment SKA-2: 2022+

DOME

- Dutch-government-sponsored SKA-focused research project between IBM and the Netherlands Institute for Radio Astronomy (ASTRON)



Source: T. Engbersen et al., "SKA – A Bridge too far, or not?," Exascale Radio Astronomy, 2014
Image credit: SKA Organisation



Big Data (~Exabytes/day) and Exascale Computing Problem

- Example: SKA1-Mid band 1 SDP (incl. 2D FFTs, gridding, calibration, imaging)
 - meeting the design specs requires ~550 PFLOPs
 - extrapolating HPC trends: 37GFLOPs/W in 2022 (20% efficiency)
 - results in **15MW** power consumption – power budget is only **2MW**
 - peak performance ~2.75 EFLOPs (at 20% efficiency)

→ Innovations needed to realize SKA

Sources: R. Jongerius, "SKA Phase 1 Compute and Power Analysis," CALIM 2014
 Prelim. Spec. SKA, R.T. Schilizzi et al. 2007 / Chr. Broekema
 Image credit: SKA Organisation

Programmable general-purpose off-the-shelf technology

- CPU, GPU, FPGA, DSP
- + ride technology wave
- do not meet performance and power efficiency targets

Fixed-function application-specific custom accelerators

- ASIC
- + meet performance and power efficiency targets
- do not provide required flexibility
- high development cost

high energy cost of programmability

Example Energy Estimates (45nm)

- 70pJ for instruction (l-cache and register file access, control)
- 3.7pJ for 32b floating-point multiply operation
- 10-100pJ for cache access
- 1-2nJ for DRAM access

Source: M. Horowitz, "Computing's Energy Problem (and what can we do about it)," ISSCC 2014.

large impact of memory on energy consumption

Programmable general-purpose off-the-shelf technology

- CPU, GPU, FPGA, DSP
- + ride technology wave
- do not meet performance and power efficiency targets

Fixed-function application-specific custom accelerators

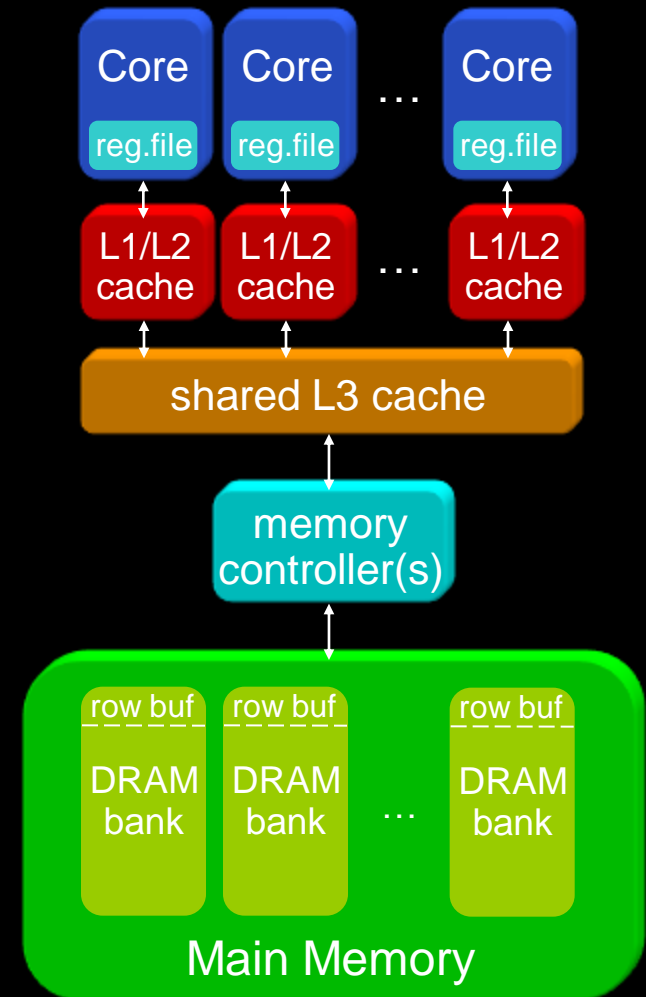
- ASIC
- + meet performance and power efficiency targets
- do not provide required flexibility
- high development cost

Research Focus

- Can we design a *programmable custom accelerator* that outperforms off-the-shelf technology for a sufficiently large number of applications to justify the costs?
- Focus on critical applications that involve *regular processing* steps and for which the key challenges relate to storage of and access to the data structure:
 - “how to bring the *right operand values* efficiently to the execution pipelines”
 - examples: 1D/2D FFTs, gridding, linear algebra workloads, sparse, stencils

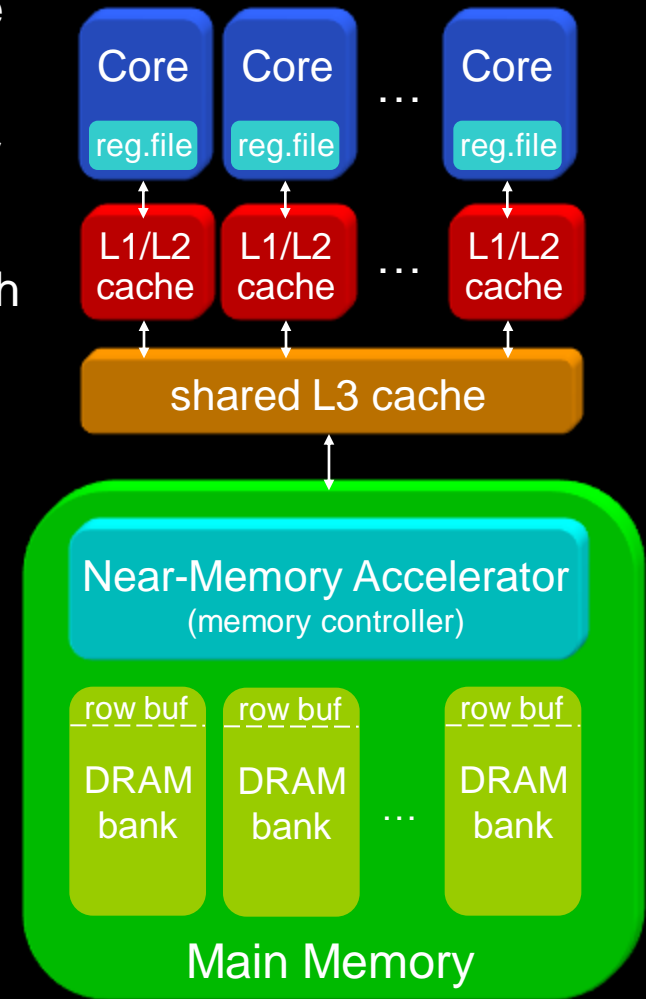
Observations (“common knowledge”)

- Access bandwidth/latency/power depend on complex interaction between **access characteristics** and **memory system operation**
 - access patterns/strides, locality of reference, etc.
 - cache size, associativity, replacement policy, etc.
 - bank interleaving, row buffer hits, refresh, etc.
- Memory system **operation** is typically **fixed** and cannot be adapted to the workload characteristics
 - extremely **challenging** to make it **programmable** due to performance constraints (in “critical path”)
 - opposite happens: “**bare metal**” programming to adapt workload to memory operation to achieve substantial performance gains
- Data organization often has to be changed between consecutive processing steps



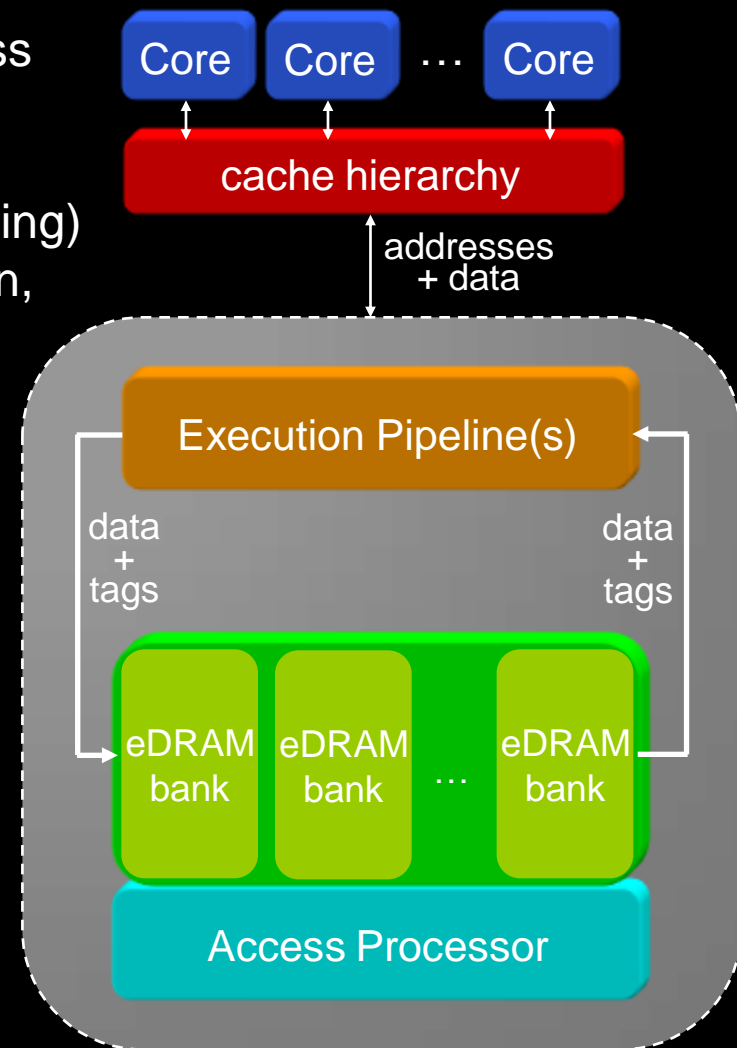
Programmable Near-Memory Processing

- Offload performance-critical applications/functions to programmable accelerators closely integrated into the memory system
- 1) Exploit benefits of *near-memory* processing (“*closer to the sense amplifiers*” / reduce data movement)
- 2) Make memory system operation *programmable* such that it can be adapted to workload characteristics
- 3) Apply an architecture/programming model that enables a more *tightly coupled scheduling* of accesses and data operations to match access bandwidth with processing rate to reduce overhead (including *instruction fetch* and *decode*)
- Integration examples
 - on die with eDRAM technology
 - 2.5D / 3D stacked architectures
 - memory module



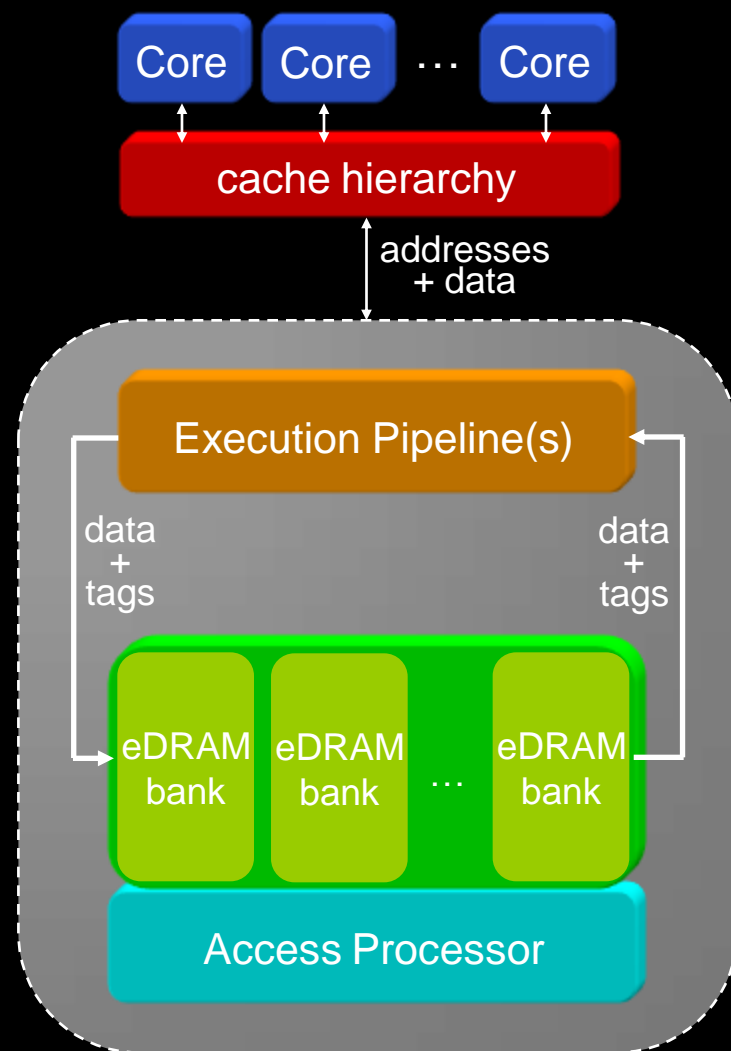
Decoupled Access/Execute Architecture

- Access processor (AP) handles all memory access related functions
 - basic operations are programmable (address generation, address mapping, access scheduling)
 - memory details (cycle times, bank organization, retention times, etc.) are *exposed* to AP
 - AP is the “*master*”
- Execution pipelines (EPs) are configured by AP
 - no need for instruction fetch/decoding
- Tag-based data transfer
 - tags identify configuration data, operand data
 - enables out-of-order access and processing
 - used as rate-control mechanism to prevent the AP from overrunning the EPs
- Availability of all operand values in the EP input buffer/register triggers execution of the operation



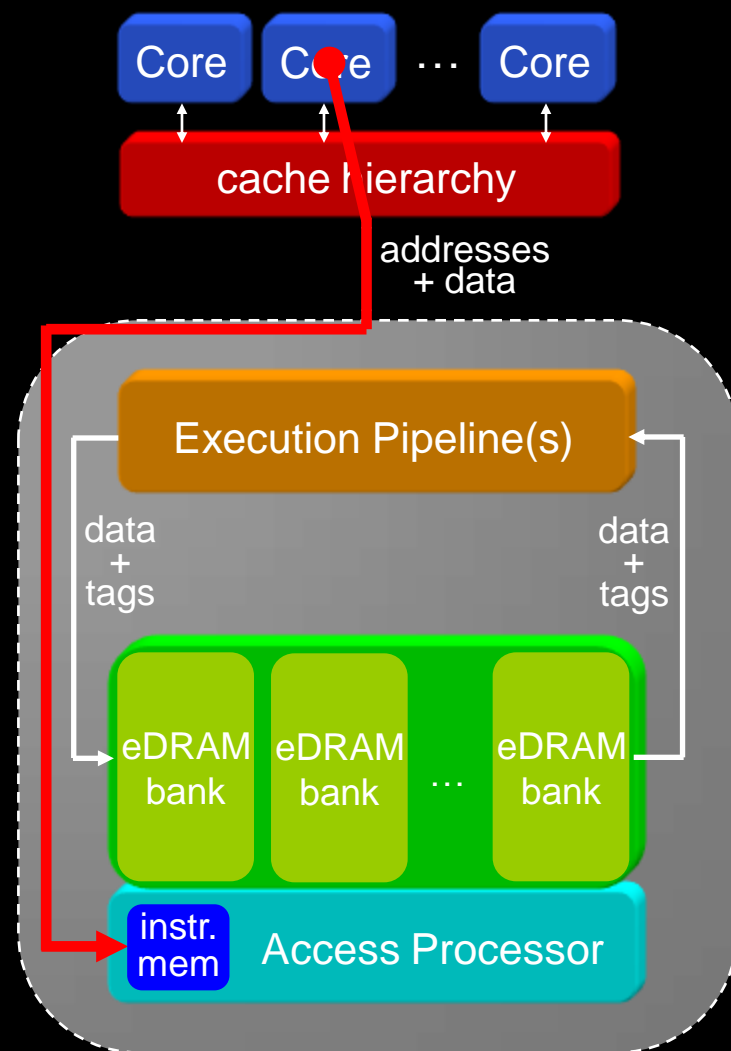
Example: FFT

(sample data in eDRAM)



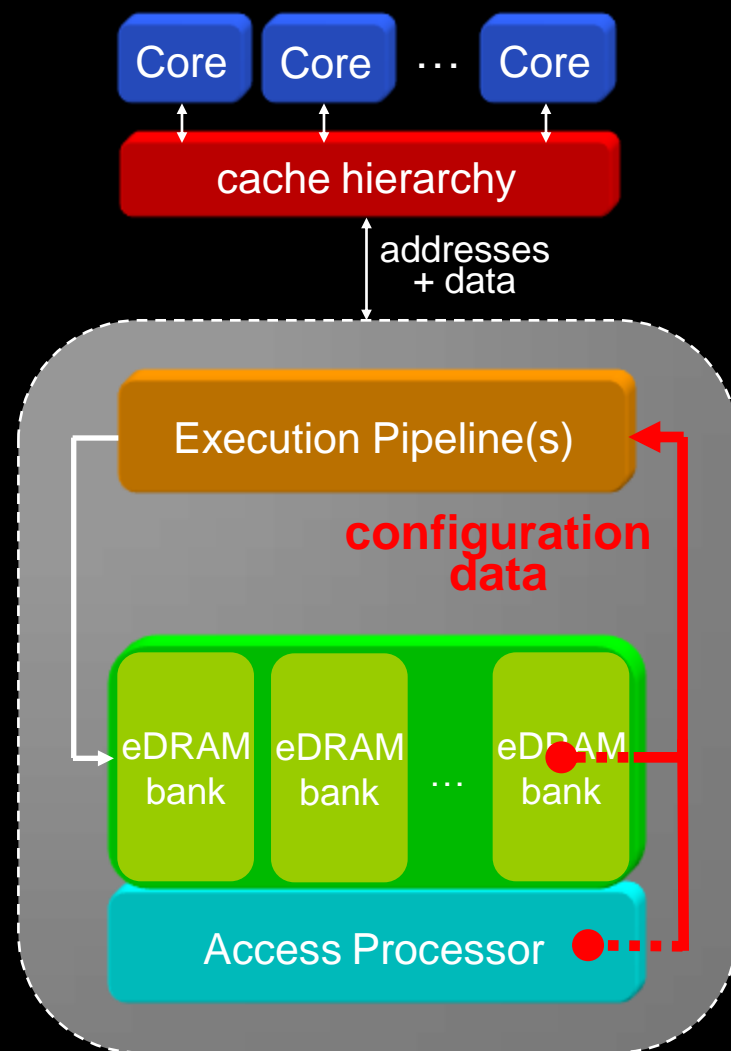
Example: FFT

- 1) Application selects function and initializes near-memory accelerator



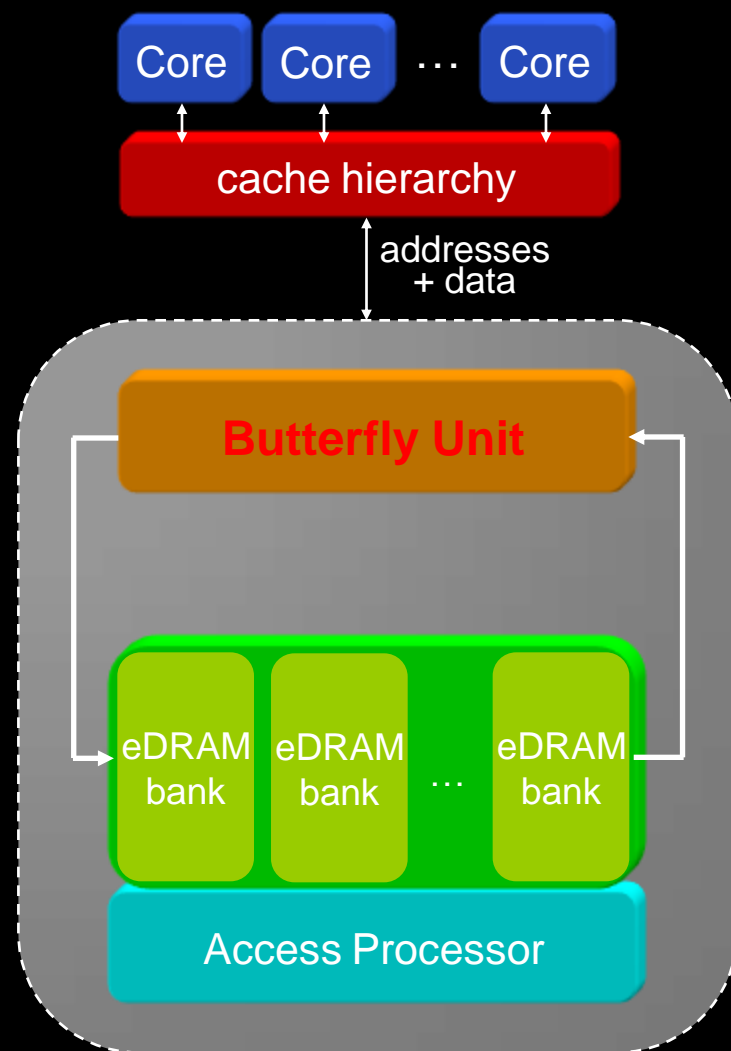
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline



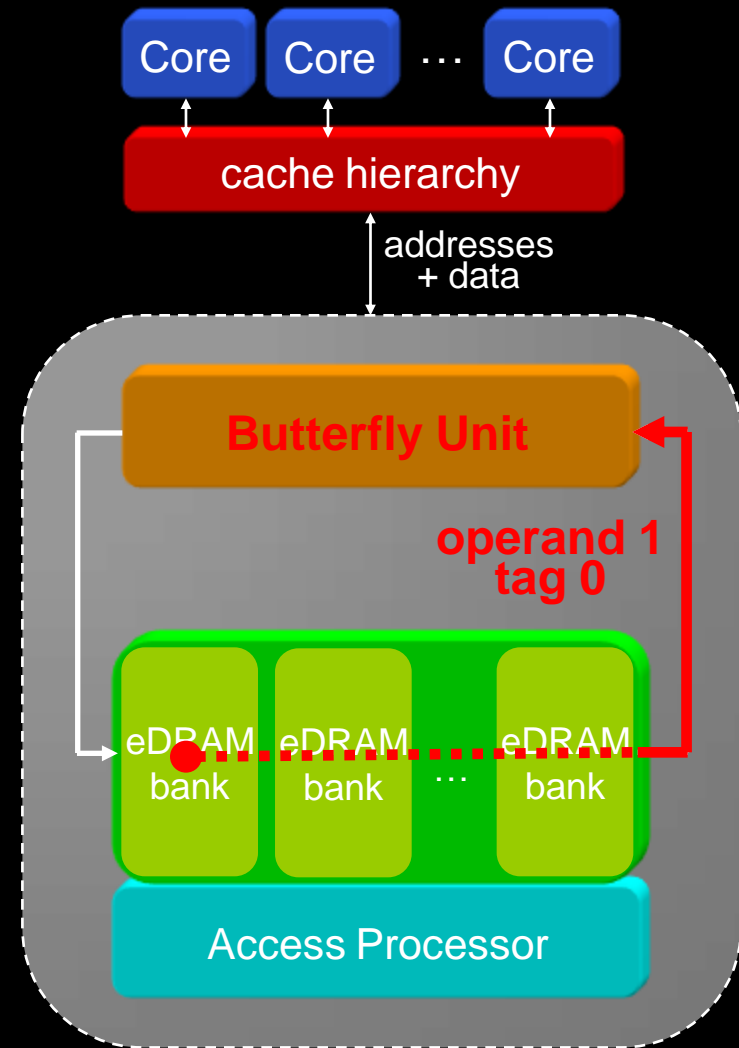
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline



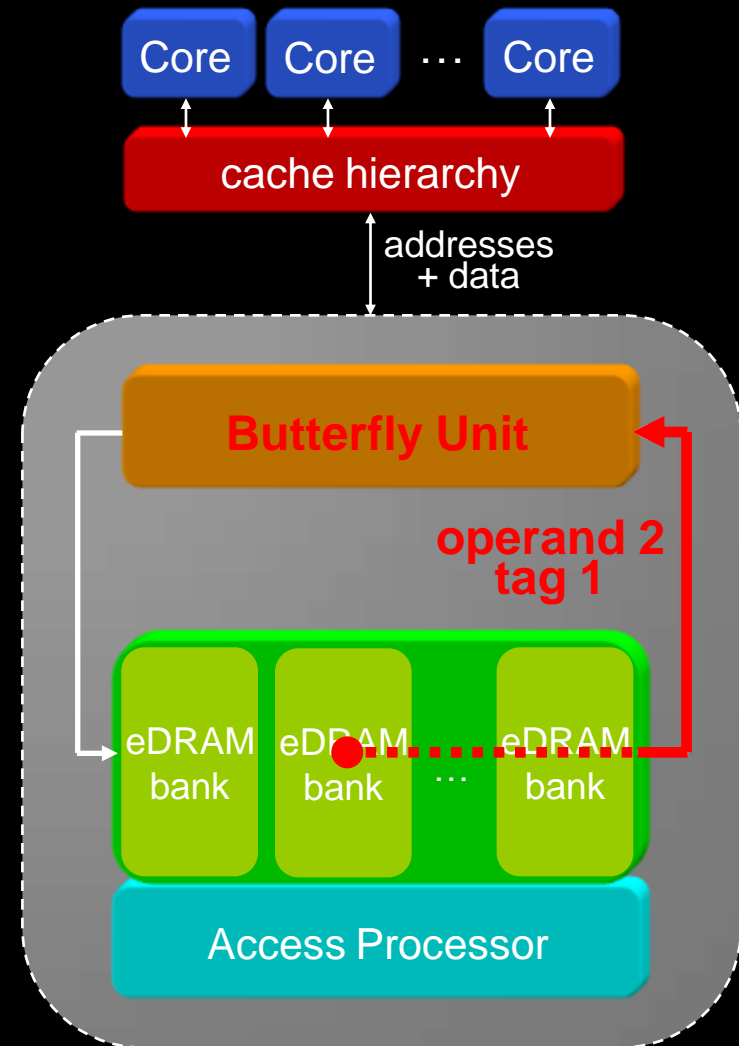
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



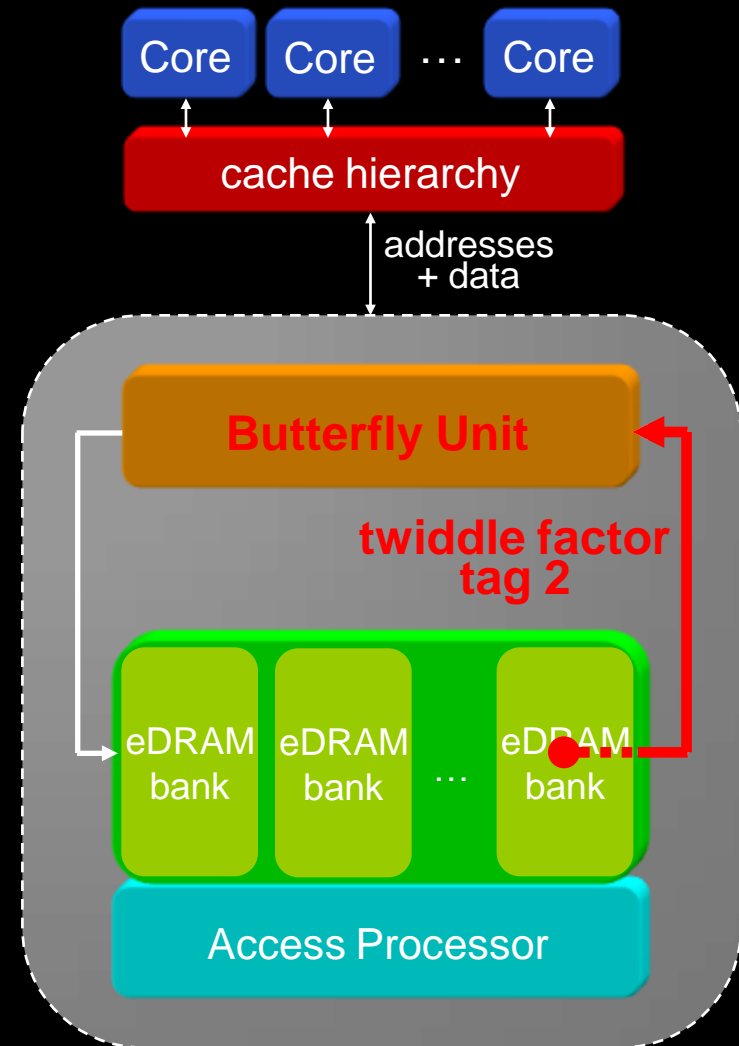
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



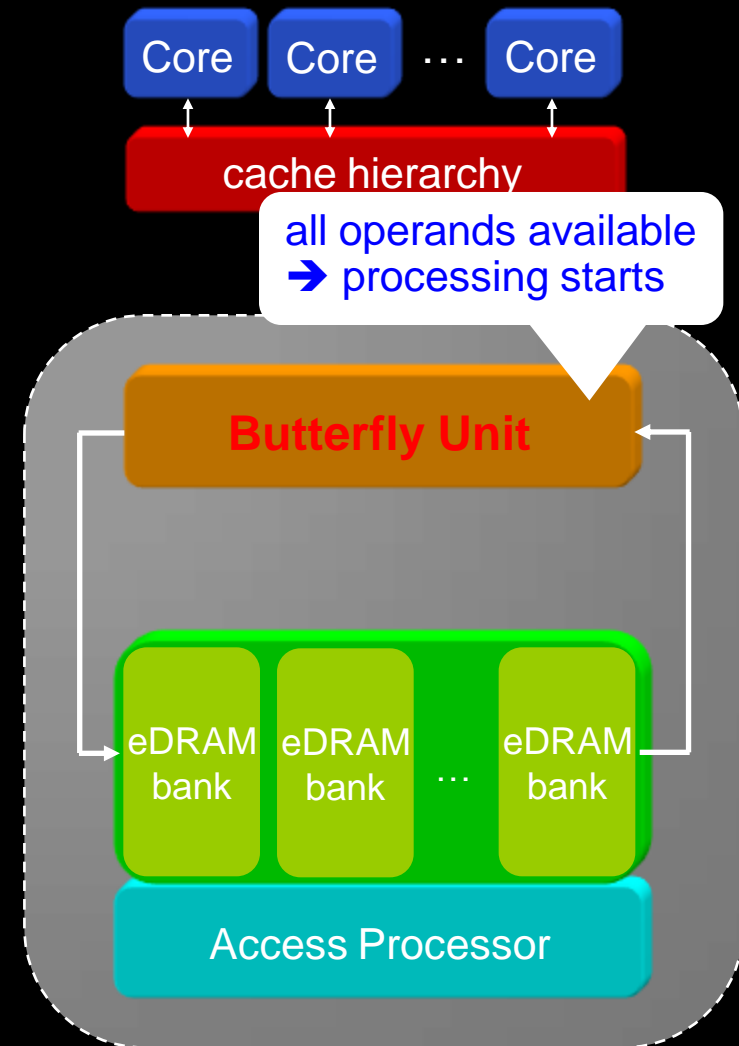
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



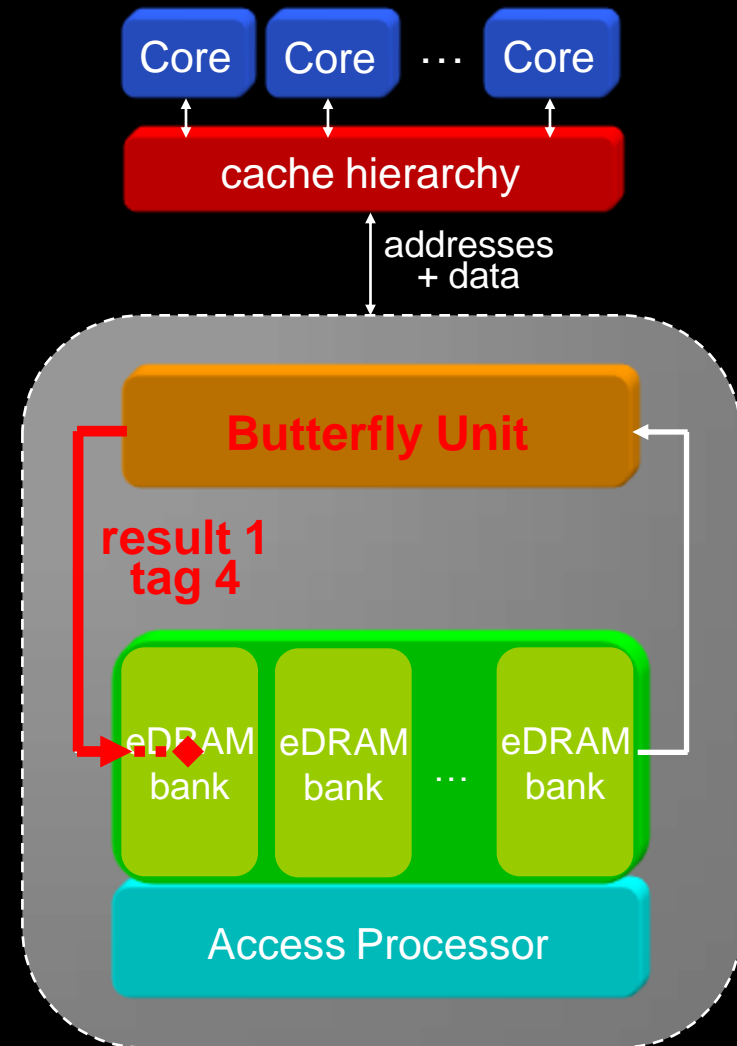
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



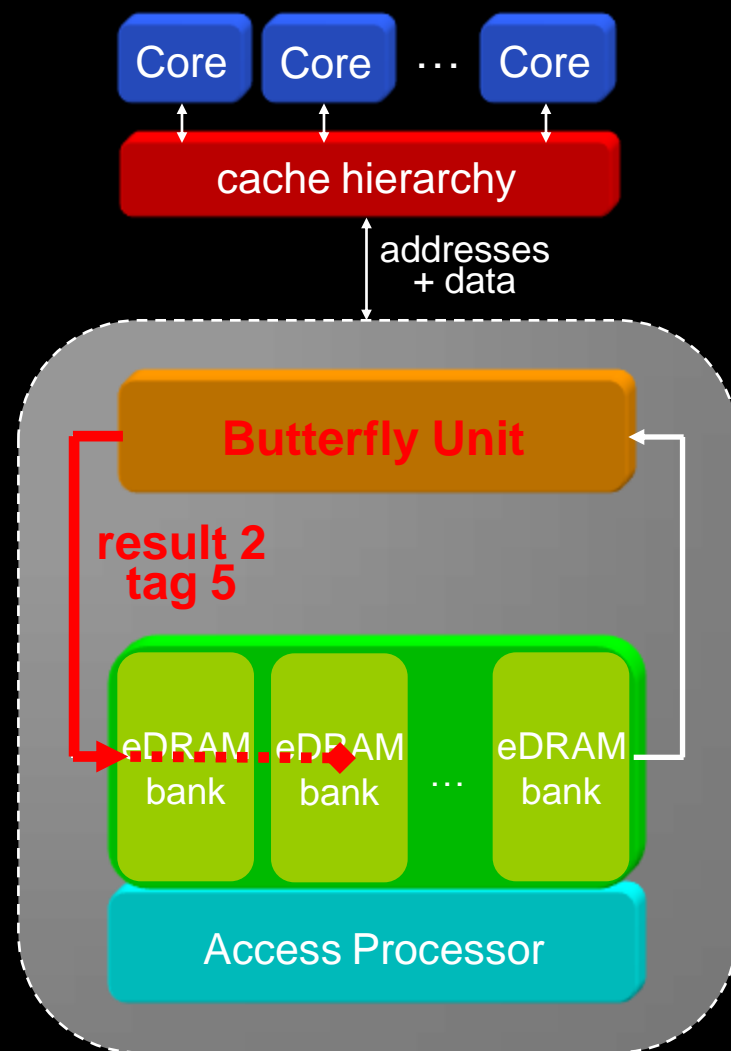
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available

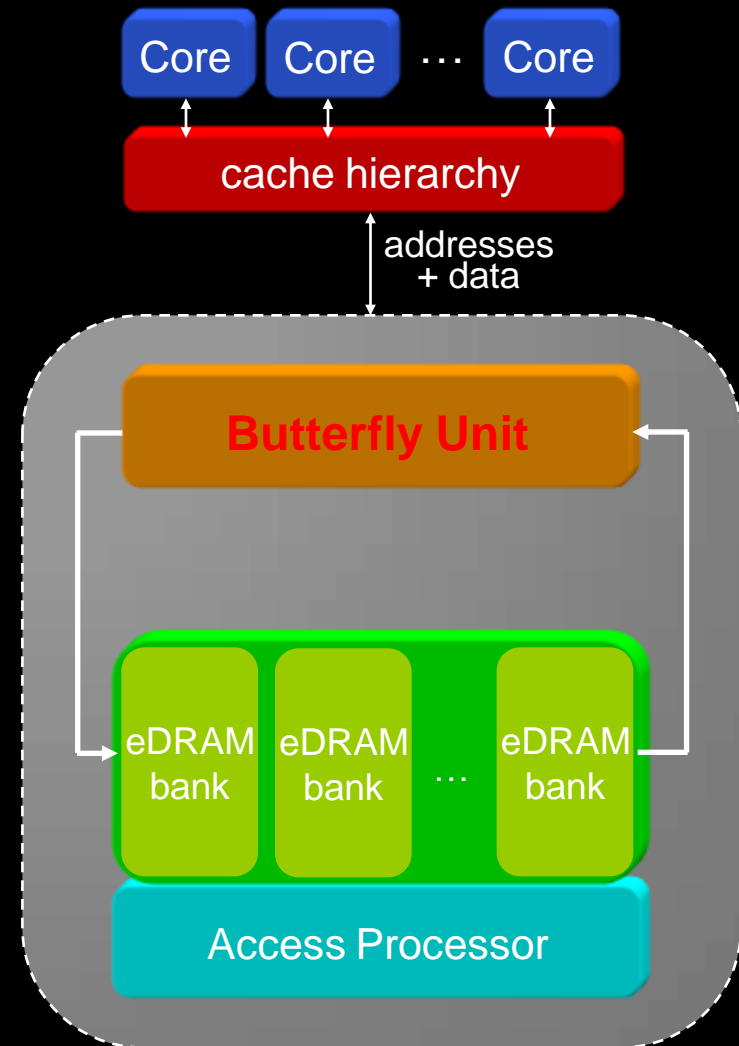


Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available

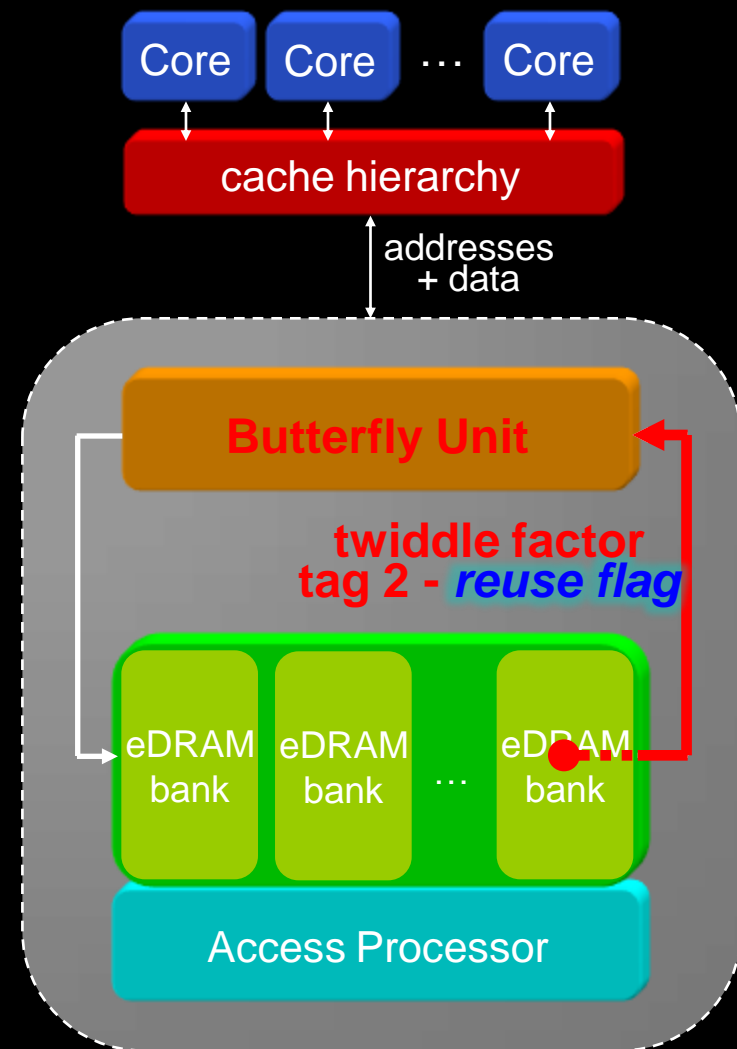
stage configuration

butterfly calculations



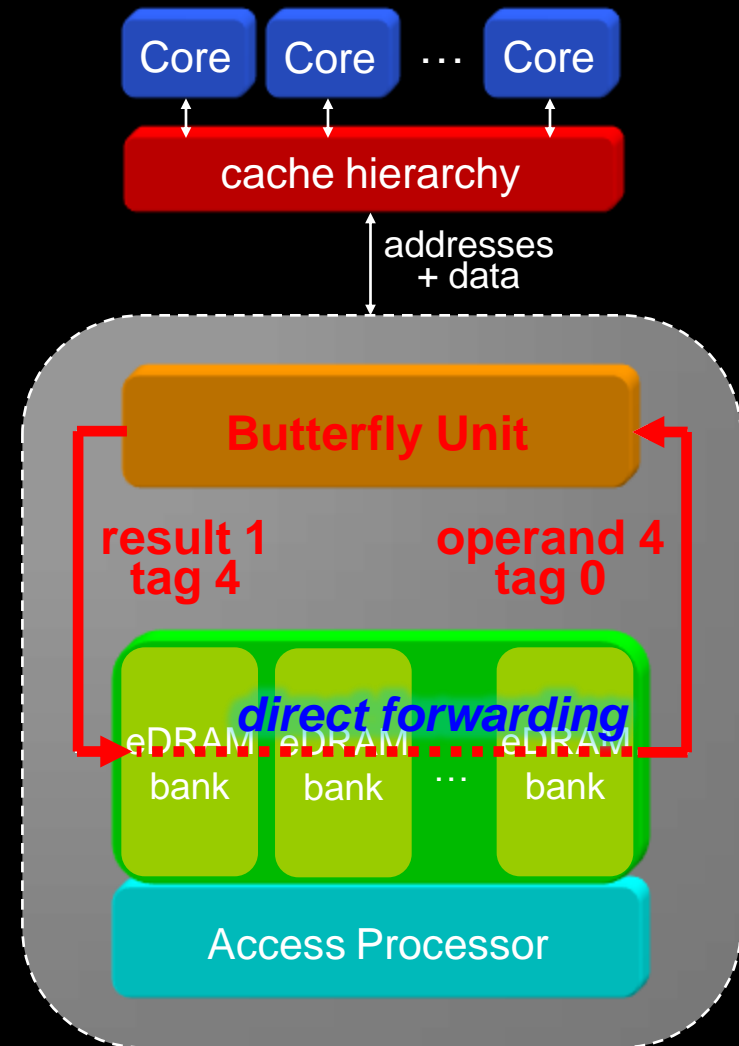
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



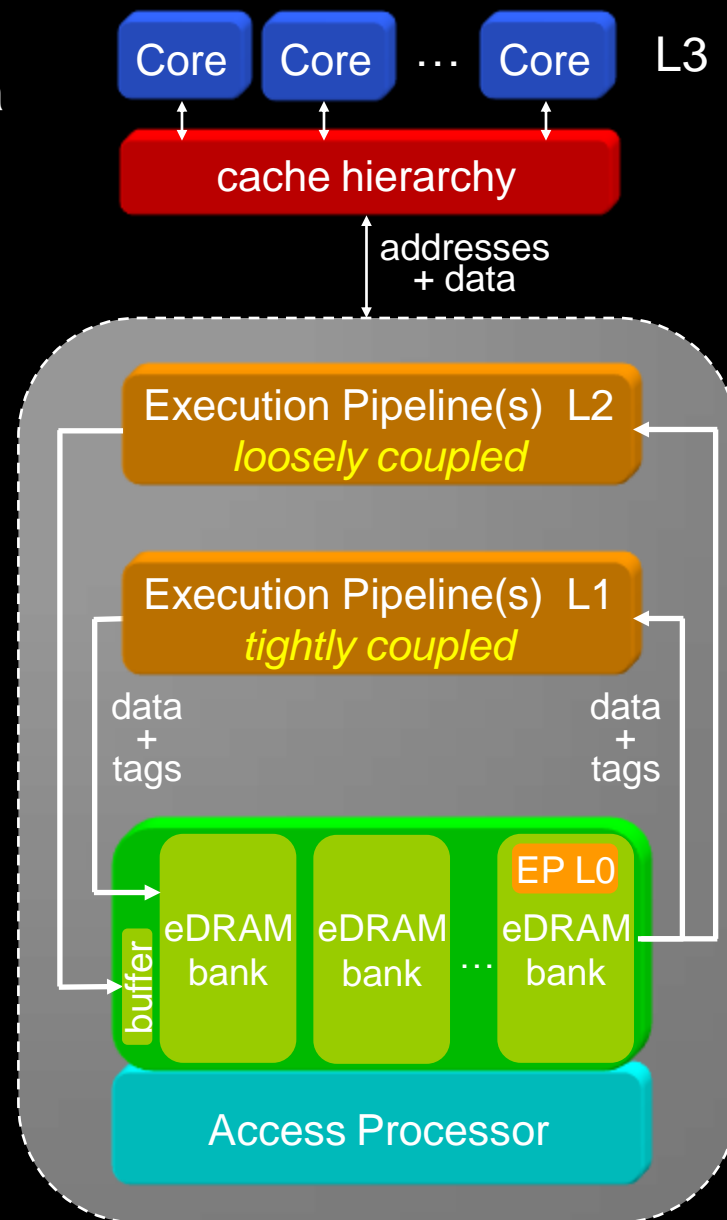
Example: FFT

- 1) Application selects function and initializes near-memory accelerator
- 2) Access Processor configures Execution pipeline
- 3a) Access Processor generates addresses, schedules accesses and assigns tags
 - read operand data for butterflies
 - write butterfly results (pre-calculate addresses)
- 3b) Execution pipeline performs butterfly calculation each time a complete set of operands is available



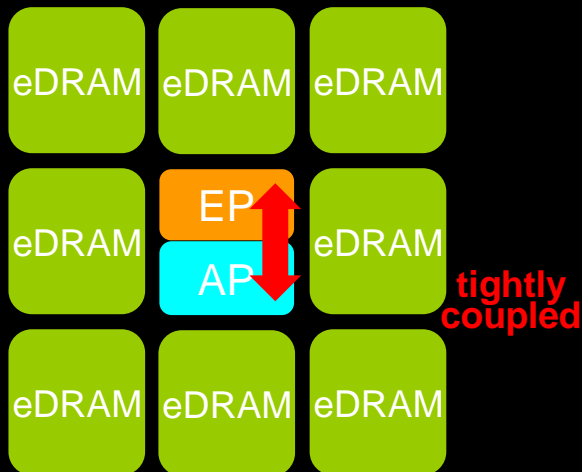
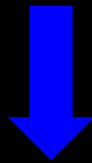
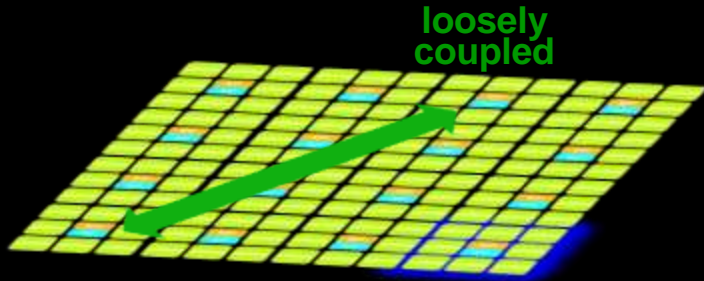
Execution Pipeline Hierarchy

- Different levels of coupling between access/data transfer scheduling and operation execution
- **L2** - loosely coupled
 - timing of transfers, execution and write accesses (execution results) not exactly known in advance to AP
 - requires write buffer
 - rate control based on #tags being “in flight”
- **L1** – tightly coupled
 - AP “knows” execution pipeline length
 - reserves slots for write execution results
 - minimizes buffer requirements
 - optimized access scheduling
“*just in time*” / “*just enough*”
- For completeness
 - **L0** – table lookup (pre-calculated results)
 - **L3** – host CPU or GPU

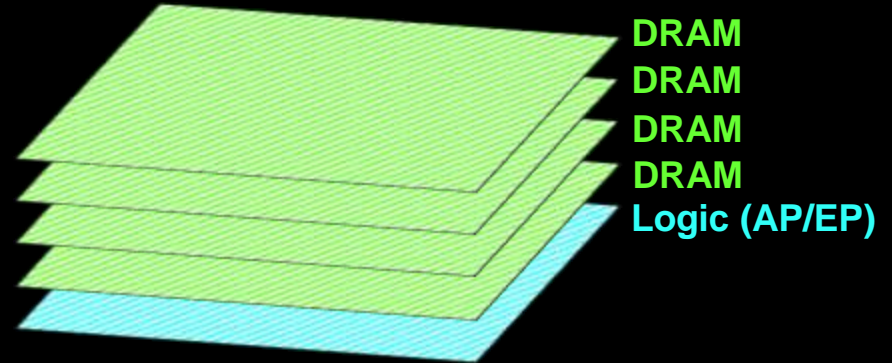


Implementation Examples

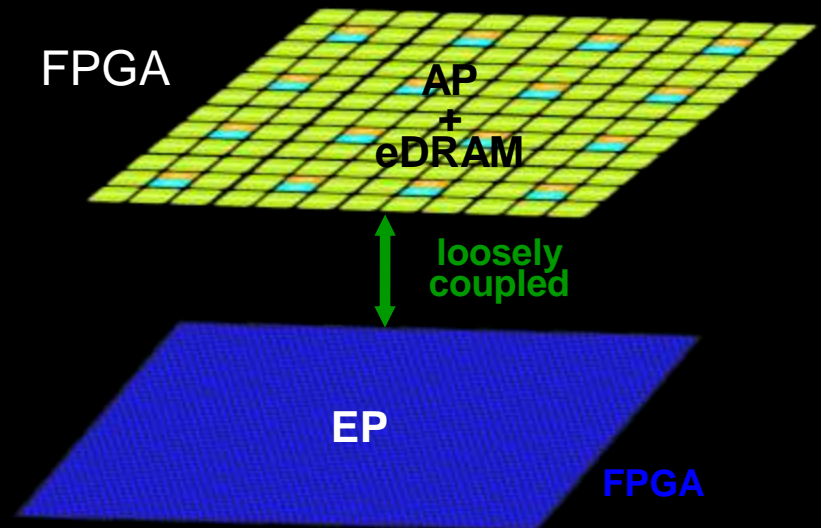
- On the same die with eDRAM



- 3D stack

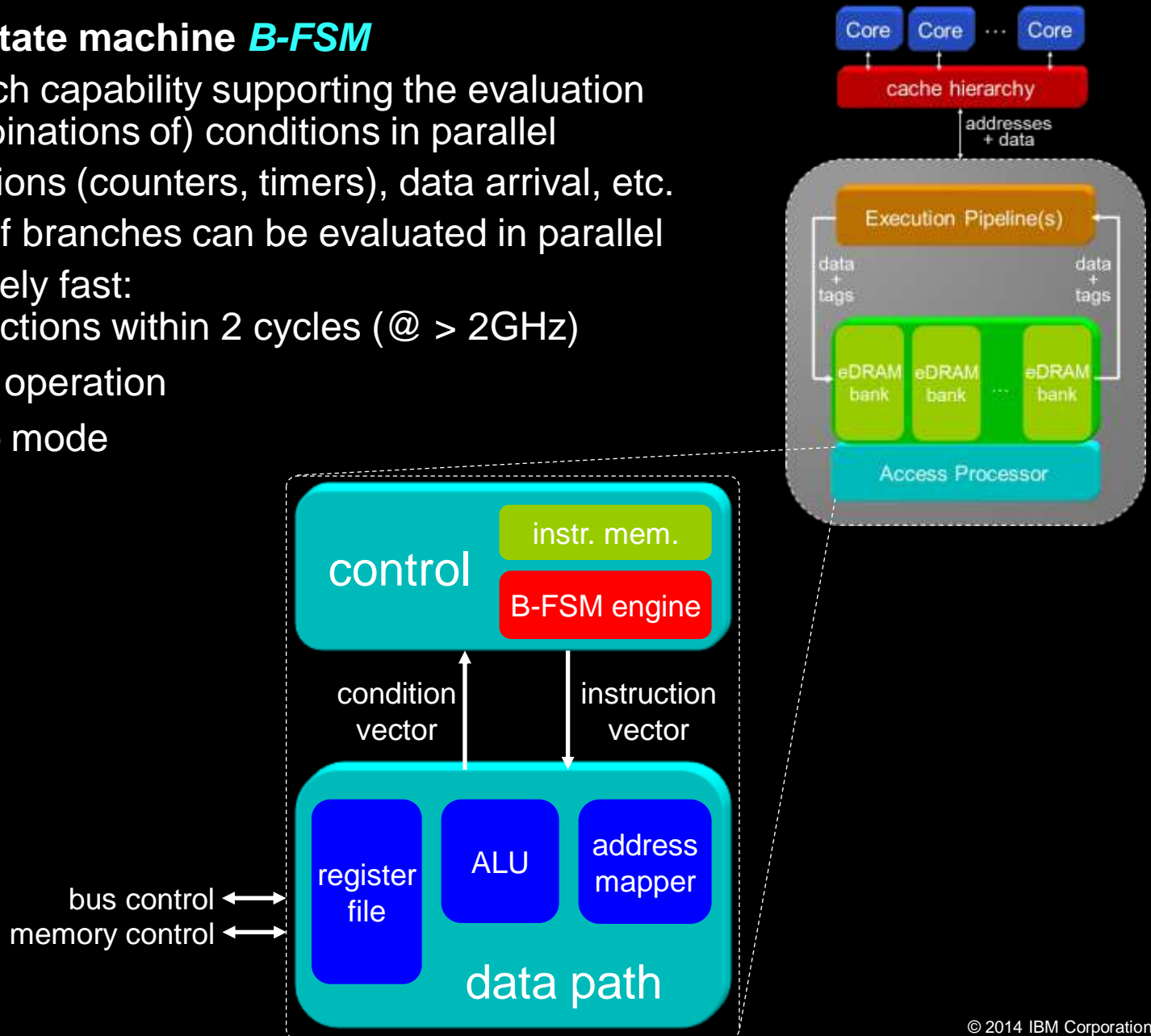


- FPGA

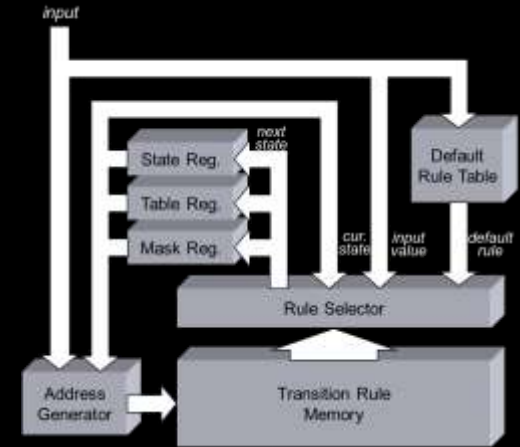


Programmable state machine *B-FSM*

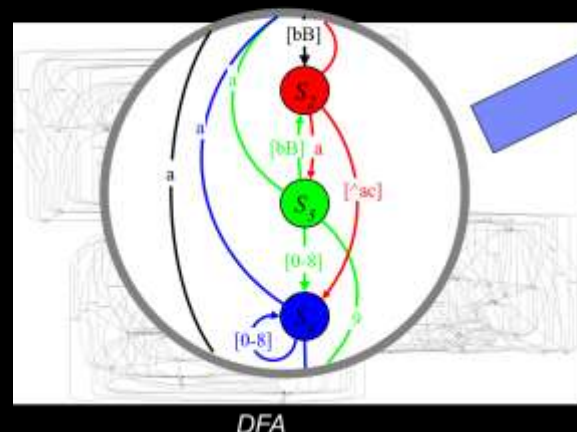
- Multiway branch capability supporting the evaluation of many (combinations of) conditions in parallel
 - loop conditions (counters, timers), data arrival, etc.
 - *hundreds* of branches can be evaluated in parallel
- Reacts extremely fast: dispatch instructions within 2 cycles (@ > 2GHz)
- Multi-threaded operation
- Fast sleep/nap mode



- Novel HW-based programmable state machine
 - deterministic rate of 1 transition/cycle @ >2 GHz
 - storage grows approx. *linear* with DFA size
 - 1K transitions fit in ~5KB, 1M transitions fit in ~5MB
 - supports wide input vectors (8 – 32 bits) and flexible branch conditions: e.g., exact-, range-, and *ternary-match*, negation, case-insensitive
- TCAM emulation



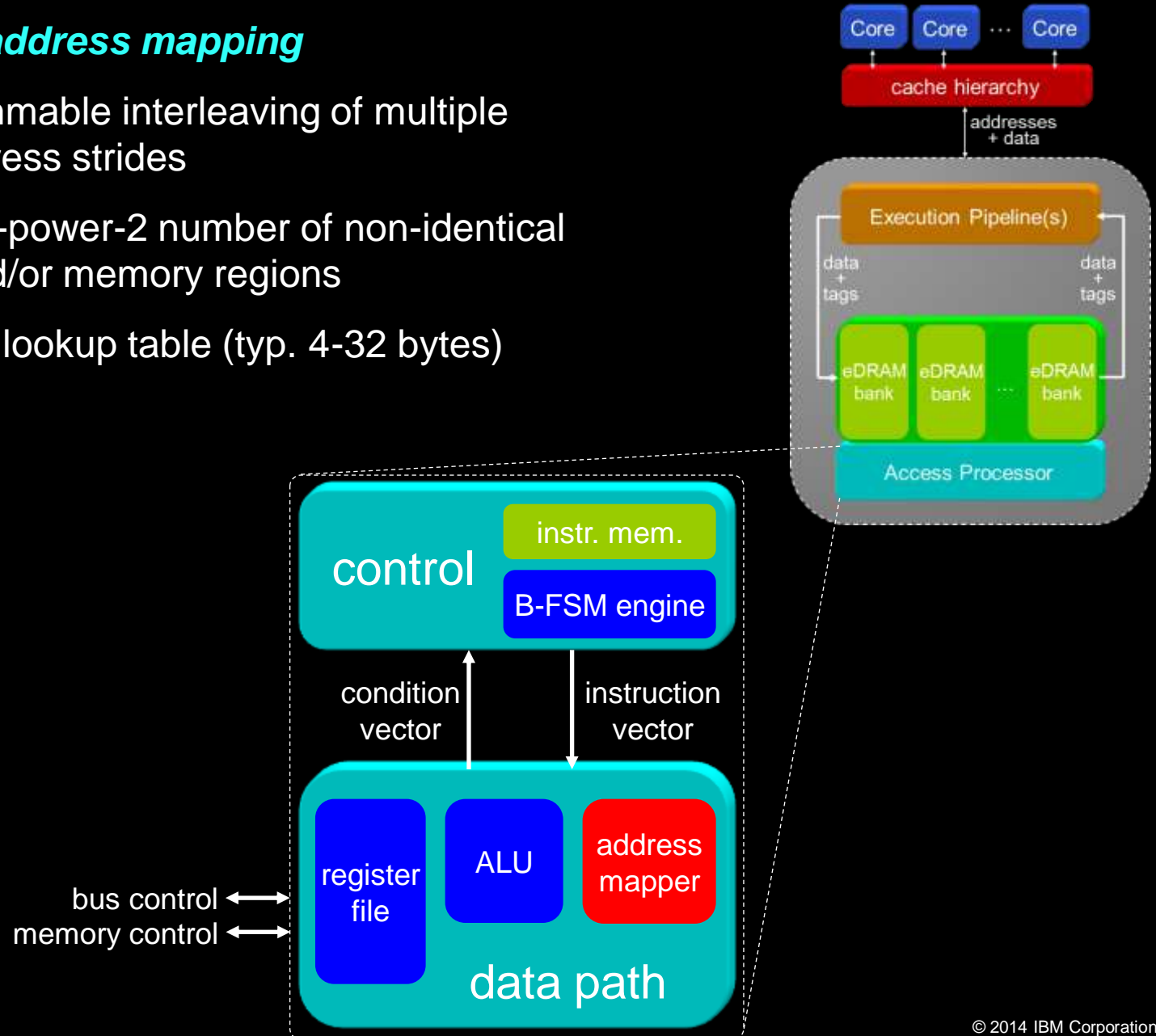
- Successfully applied to a range of accelerators
 - regular expression scanners, protocol engines, XML parsers
 - processing rates of *~20Gbit/s* for single B-FSM in 45nm
 - small area cost enables scaling to extremely high aggregate processing rates

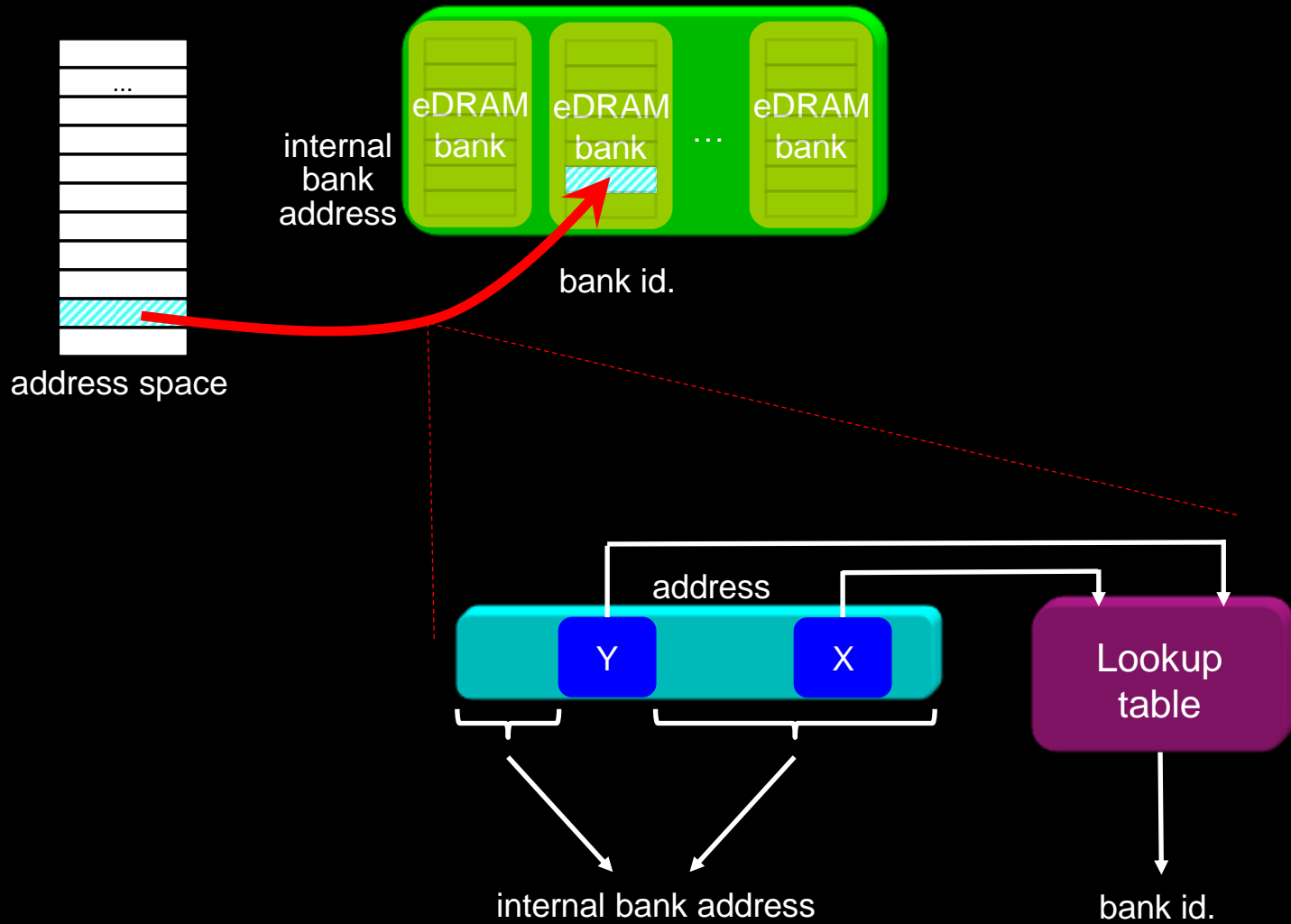


B-FSM data structure

Programmable *address mapping*

- Unique programmable interleaving of multiple power-of-2 address strides
- Support for non-power-2 number of non-identical sized banks and/or memory regions
- Based on small lookup table (typ. 4-32 bytes)





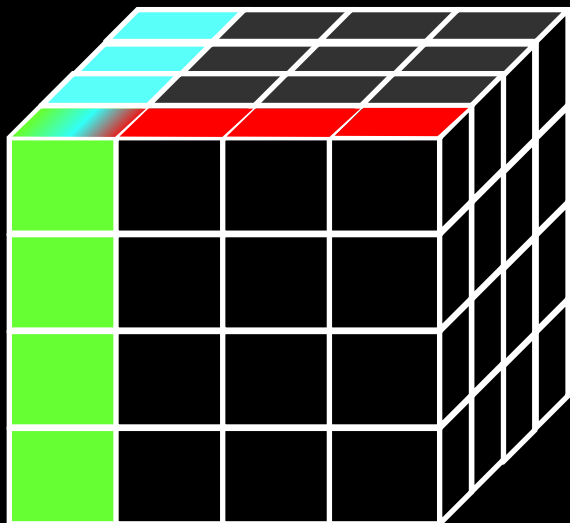
- LUT size = 4 bytes



...
256	1024	1280	1536	1792
...
7	263	519	775	7
6	518	774	6	262
5	773	5	261	517
4	4	260	516	772
3	259	515	771	3
2	514	770	2	258
1	769	1	257	513
0	0	256	512	768
	0	1	2	3
	memory banks			

- Simultaneous interleaving of row and column accesses
- Example: $n=256$
 - two power-of-2 strides: **1** and **256**

- LUT size = 16 bytes



- Simultaneous interleaving of row, column, and “vertical layer” accesses
- Example
 - three power-of-2 strides: **1**, **16** and **256**

...
256	1024	1280	1536	1792
...
64	64	320	576	832
63	575	831	63	319
...
48	304	560	816	48
47	815	47	303	559
...
32	544	800	32	288
31	31	287	543	799
...
16	784	16	272	528
15	271	527	783	15
...
5	773	5	261	517
4	4	260	516	772
3	259	515	771	3
2	514	770	2	258
1	769	1	257	513
0	0	256	512	768
	0	1	2	3

memory banks

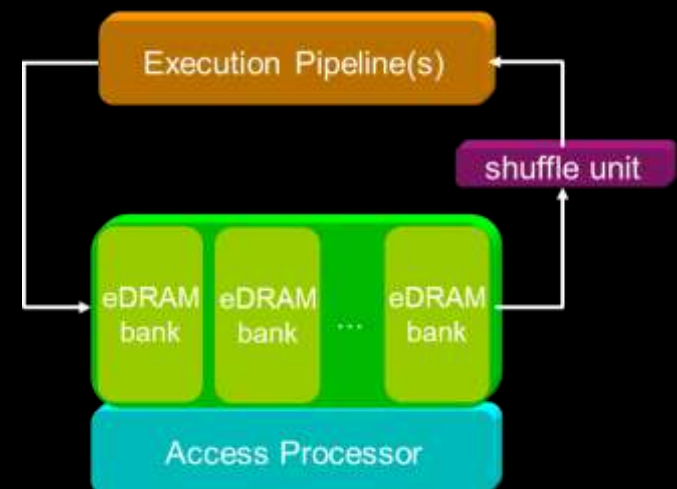
Power

- Access Processor power estimate for 14nm, > 2GHz: **25-30 mW**

Amortize Access Processor energy over large amount of data

- Maximize memory bandwidth utilization (basic objective of the accelerator)
 - bank interleaving, row buffer locality
- Exploit wide access granularities
 - example: 512 bit accesses @ 500 MHz [eDRAM]
 - estimated AP energy overhead per accessed bit $\approx 0.1 \text{ pJ}$

- Make sure that all data is effectively used
- improved algorithms
 - data shuffle unit: swap data at various granularities within one/multiple lines (configured similar as EP)



New Programmable Near-Memory Accelerator

- Made feasible by novel state machine and address mapping technologies that enable programmable address generation, address mapping, and access scheduling operations in “real-time”
- Objective is to minimize the (energy) overhead that goes beyond the basic storage and processing needs (memory and execution units)
- Proof of concept for selected workloads using FPGA prototypes and initial compiler stacks
- More details to be published soon