Beyond von Neumann: fundamental limitations of current compute architectures for real-time image processing

Sergio Goma

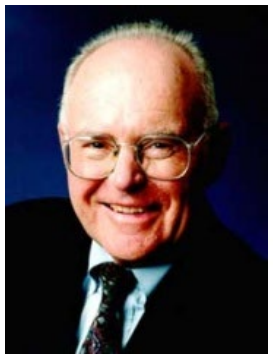
Alkemista IND

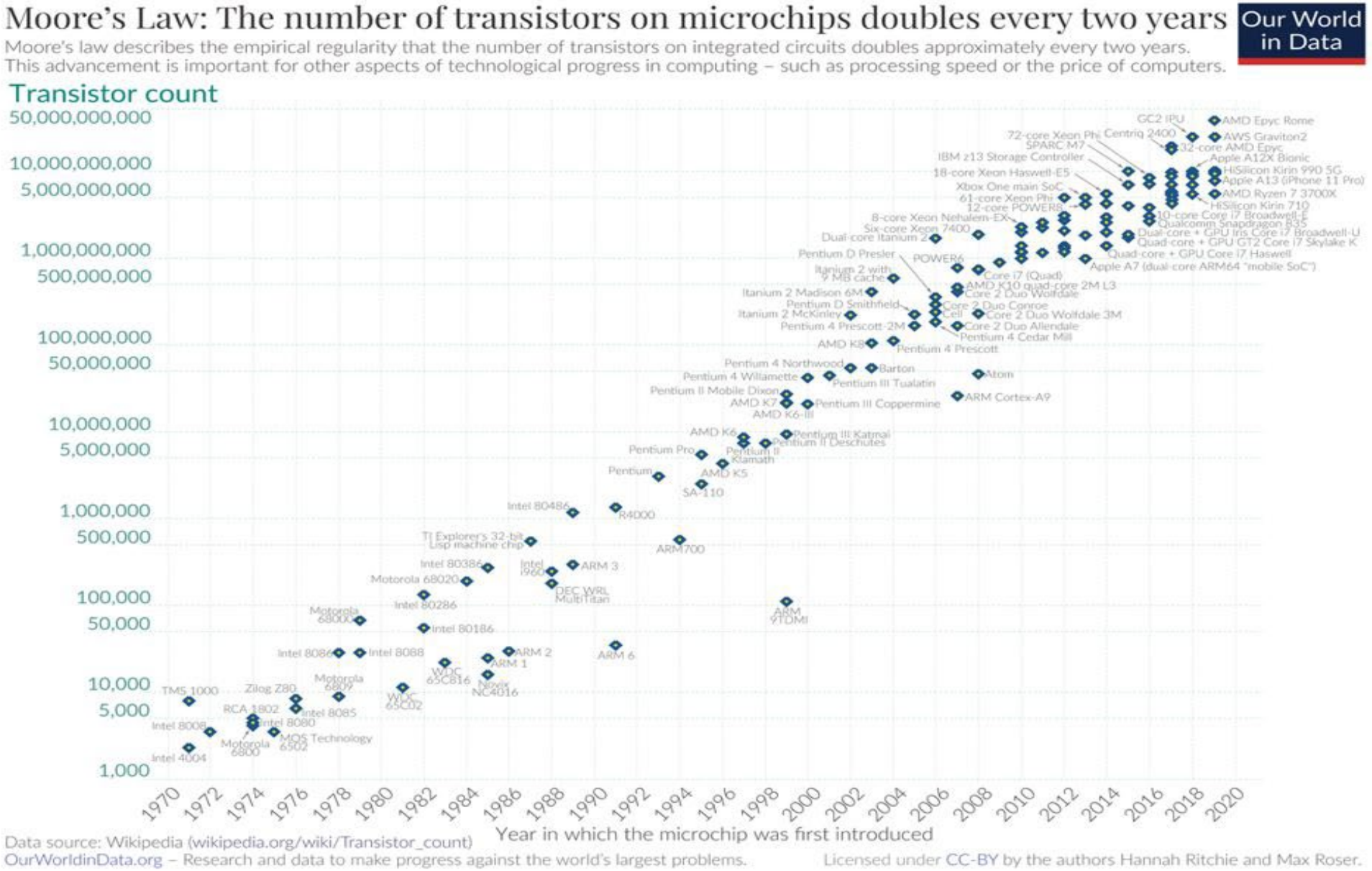
Acknowledgments and contributions on slide 33

Agenda

Main purpose of this work is threefold:

1. Raise awareness that data bottleneck (memory wall) is fundamental
2. Identify the main bottlenecks of von Neumann architecture creating the memory wall
3. Show that a revolutionary change is needed and provide an example of such a change – ‘fluid datapaths’





This Photo by Unknown Author is licensed under CC BY-SA

Figure 1 Moore's Law for Microchips

Moore’s law – pixel size

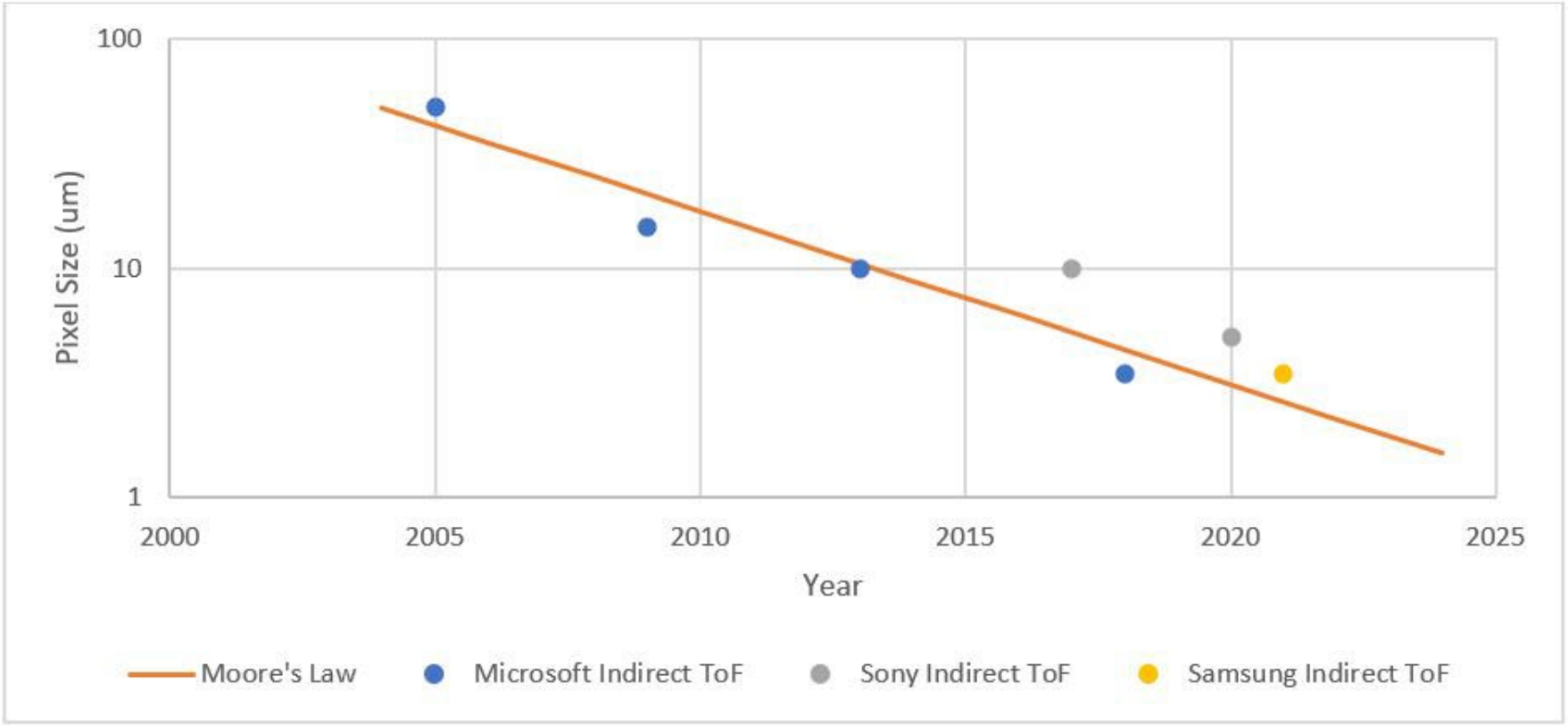


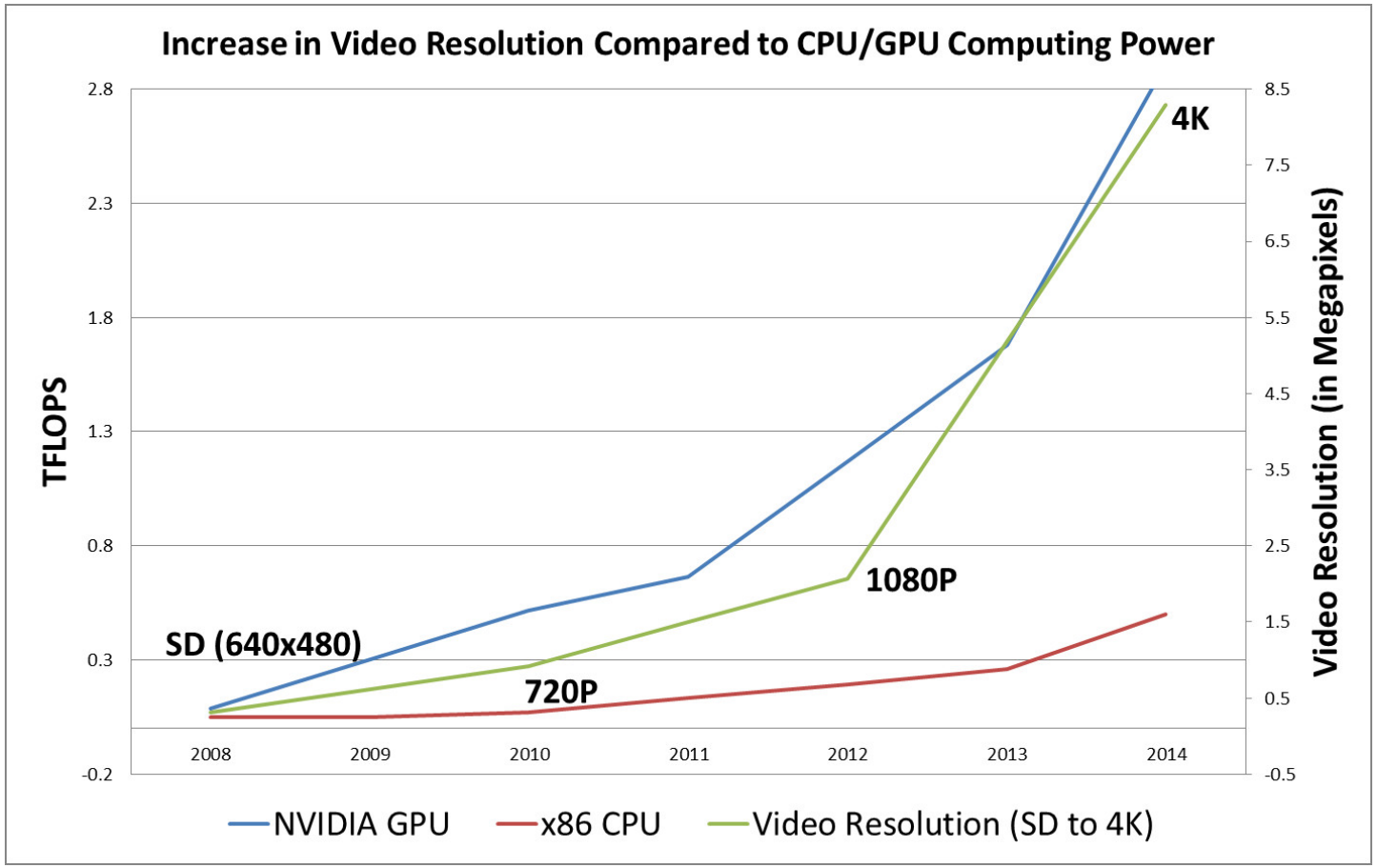
Figure 3 Moore's Law vs ToF Pixel Size Progression

Ref: https://devblogs.microsoft.com/azure-depth-platform/moores-law-for-tof-pixels/

Video resolutions have surpassed Moore's Law

- **Resolutions are going up faster than Moore's Law**

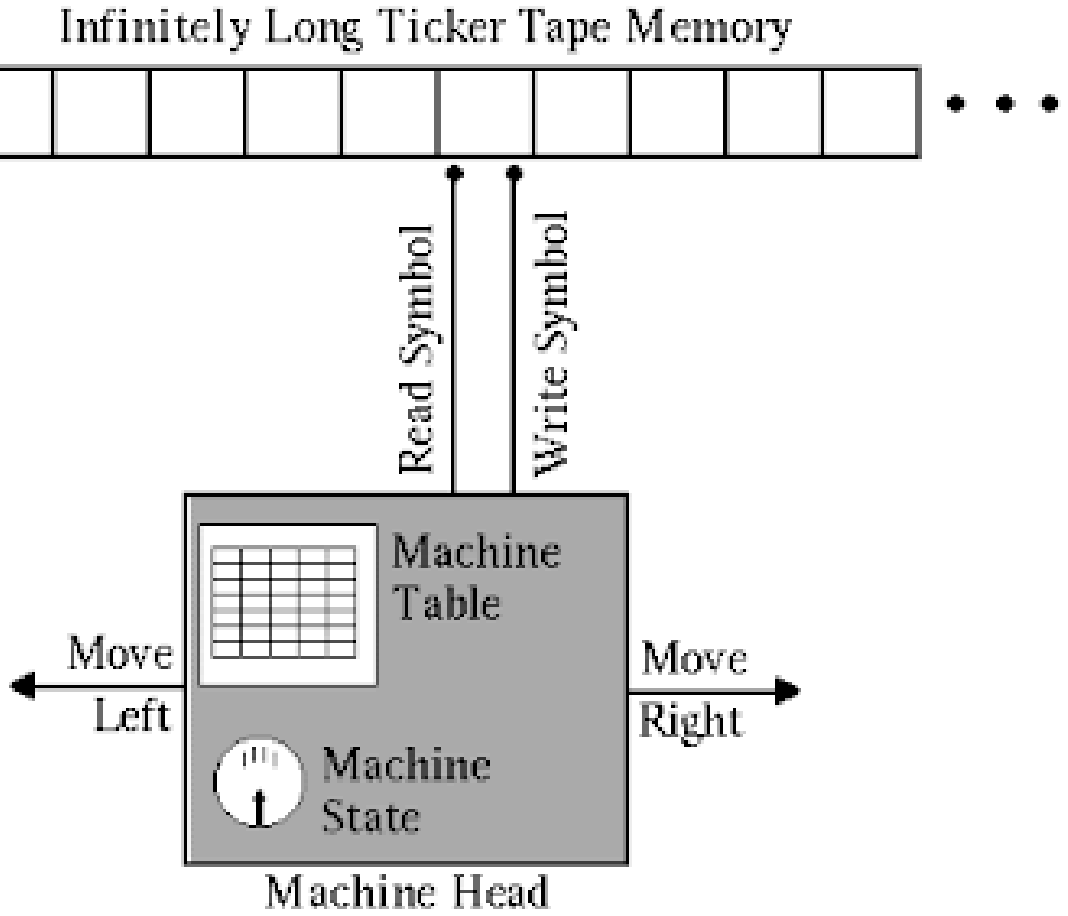
- From 2009 to 2014, video resolutions increased from SD (640x480) to 4K (3840x2160). That's 27 times more raw pixels. Over that same period of time, GPUs increased their processing power by 33x, but Moore's Law saw CPUs increase by only 10.3x.
- Sensor resolutions are going even farther. **Harris's CorvusEye** is 115 megapixels (~10,720 x 10,720) and **PV Labs' PSi-ViSiON** 3000 system is 300+ megapixels (~17,320 x 17,320).
- If your application requires real-time processing (as in on-the-fly processing of a live video or high-resolution image stream in an aircraft or sensor), then CPUs will not work.



https://www.linkedin.com/pulse/video-resolutions-have-surpassed-moores-law-what-does-sean-varah

The origins: Turing

The original concept was intended for off-line data processing.



Turing Machine

1950 Turing, Computing machinery and intelligence

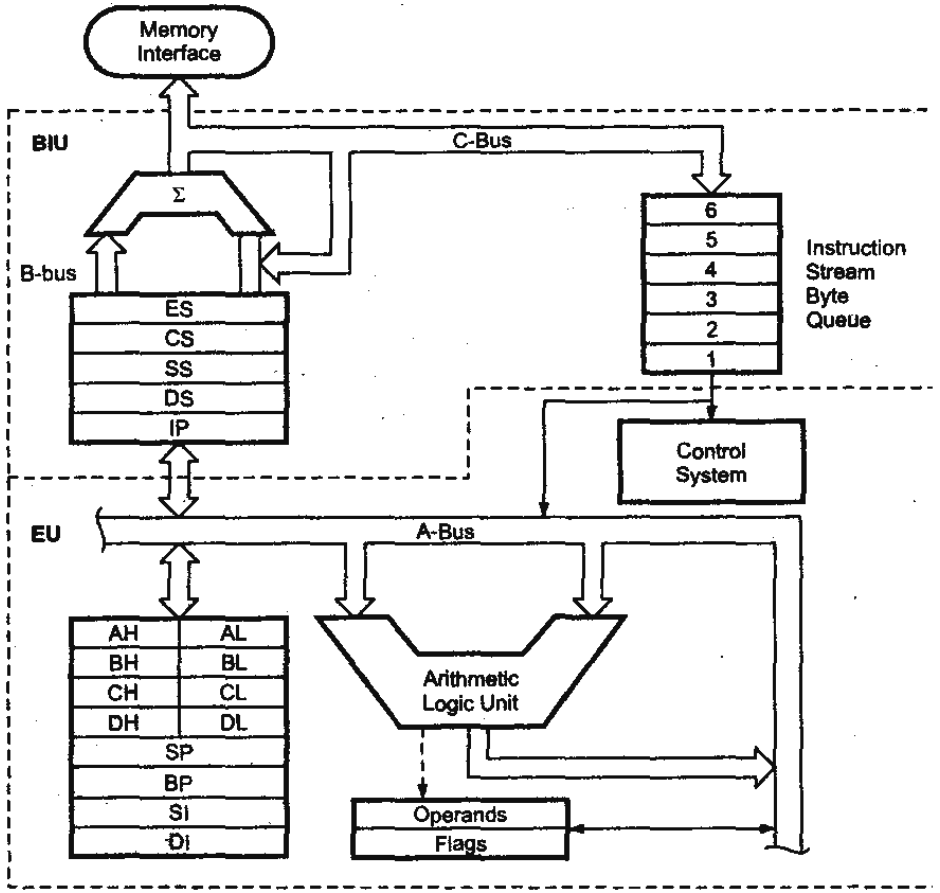
The problem: Computing Data Bottleneck

The “memory wall” problem or so-called von Neumann bottleneck limits the efficiency of conventional computer architectures, which move data from memory to CPU for computation; these architectures cannot meet the demands of the emerging memory-intensive applications. *

The staple of this architecture is load-store or register-register data transfers:

Any ALU operation occurs between preloaded registers and is motivated by off-line processing.

Applies to both Princeton and Harvard architectures (we assume instruction fetch issue is solved)



2021 Zou, X., Xu, S., Chen, X. *et al.* Breaking the von Neumann bottleneck: architecture-level processing-in-memory technology

x86

Off-line architectures - RISC

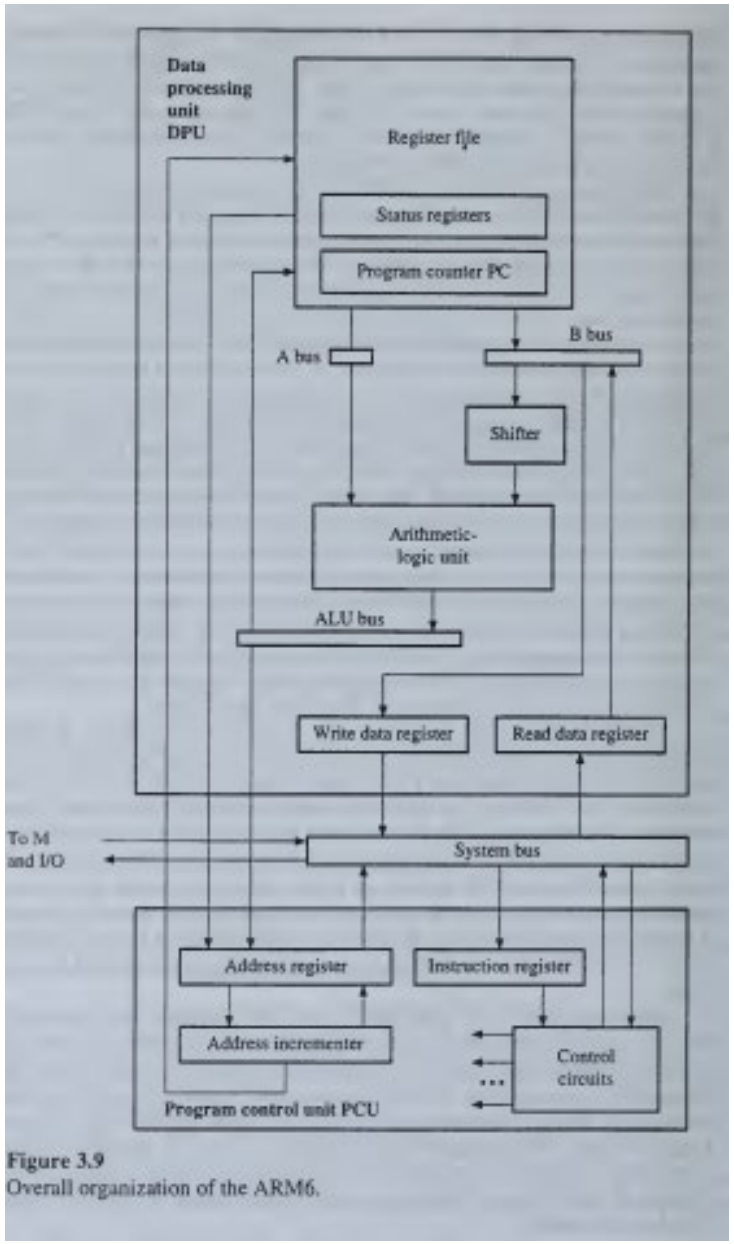
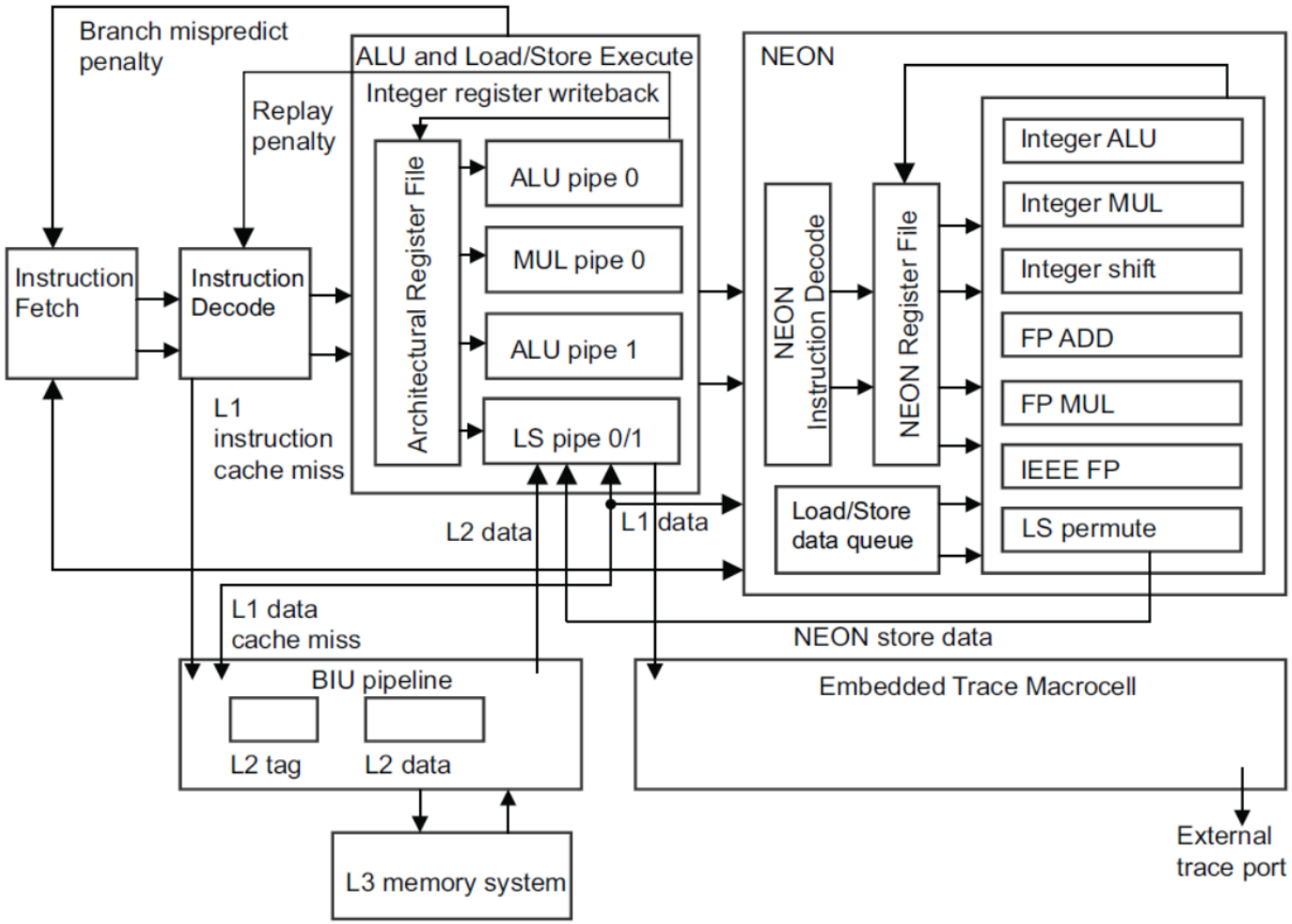


Figure 3.9
Overall organization of the ARM6.

ARM6



ARM+NEON (Cortex-A8)

Computing Data Bottleneck – vonNeumann Assumptions

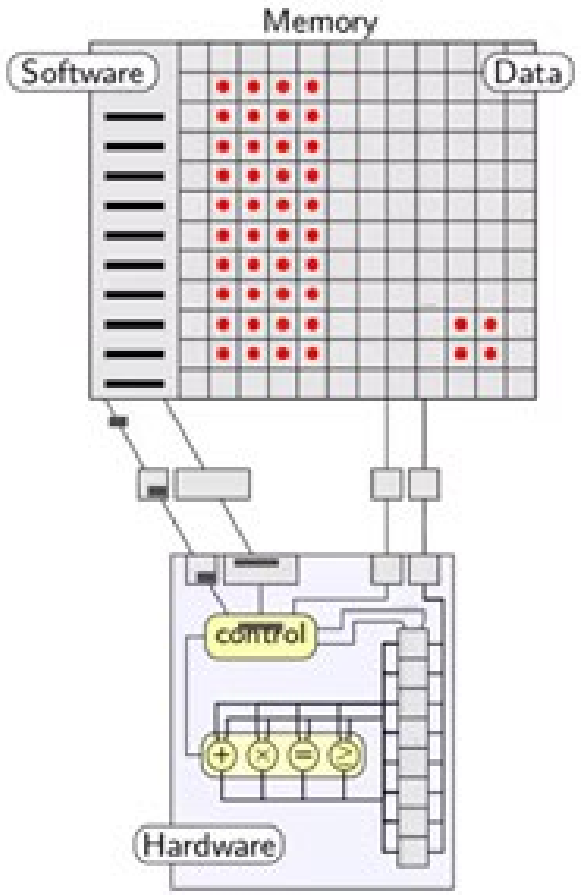
Microsequence Everything – this is a use-case assumption, as execution time was not critical (off-line processing)
– Instructions for compute are sequenced
- Data movement to and from register file is sequenced

• Correction needed due to real-time processing use-cases

Minimize Hardware – this is a technology assumption based on the fact that hardware was expensive to make

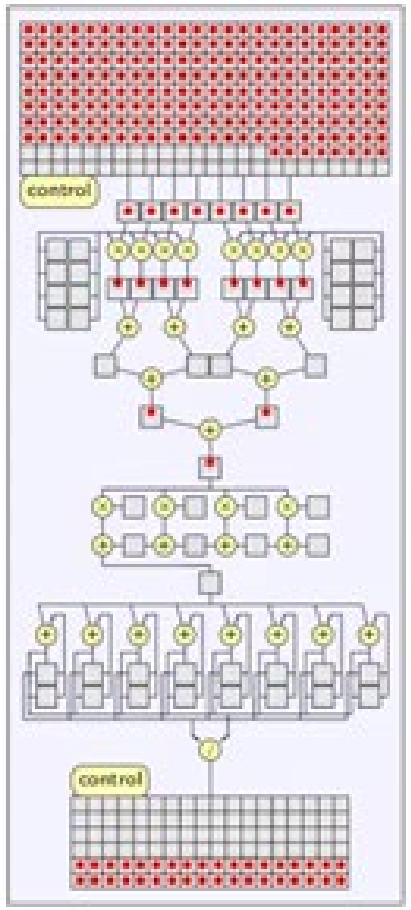
• Correction needed due to high cost of software development, availability of hardware design and increased efficiency of distributed systems

Computing Data Bottleneck - video



Normal processor

Software controls computation process



FPGA

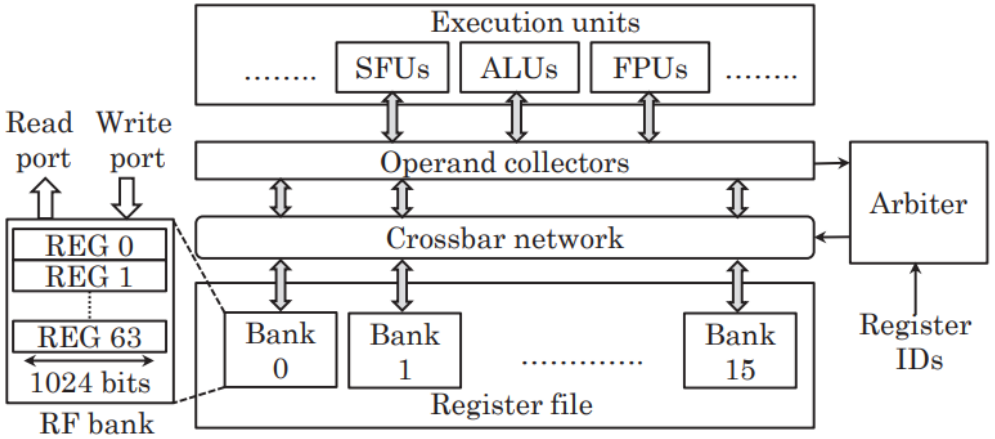
Computation integrated in hardware

https://youtu.be/BML1YHZpx2o

Off-line vs On-line architectures (data flow)

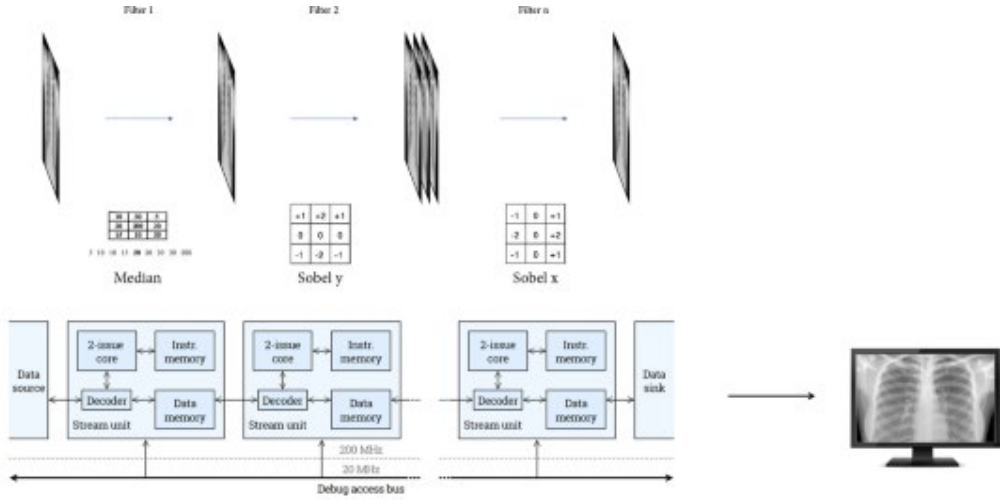
Off-line:
- all data is stored and available
- latency is irrelevant
- no dominant data flow
- processing time is not critical

On-line:
- relevant data is always in the present
- latency is critical (to close loop)
- data has one dominant direction
- processing rate needs to be higher than acquisition rate



RISC-V





FPGA/Custom real-time

Off-line architectures - data bottleneck causes

1. **Load/store** (register-register architecture): Data is fetched in registers (register file), aligned and preprocessed before fed to compute elements, then stored back in registers – off-line processing is well supported.

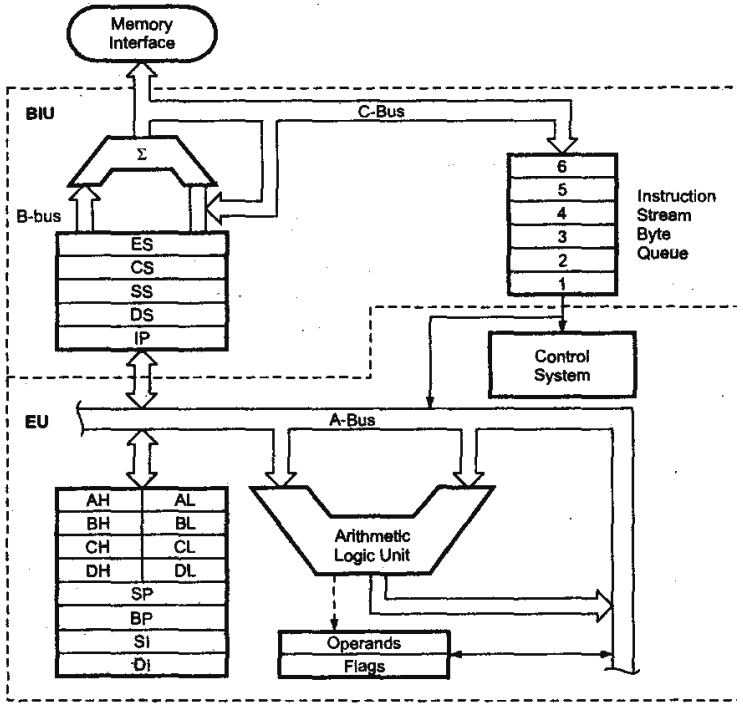
2. **Data rearrangement** – data in memory needs to be arranged before reaching the inputs of compute elements (the compute element function is fixed -> input/out format is fixed)

3. **Address calculation** – next data address calculation is part of the data path, the compute units are used for address calculation.

4. **Compute Elements -** (ALU/MACs) – delimiting compute elements at register interface exacerbates the data bottleneck

Off-line architectures and data bottleneck – Load/Store

Load/store (register-register architecture): Data is fetched in registers (register file), aligned and preprocessed before fed to compute elements, then stored back in registers - off-line processing is well supported.



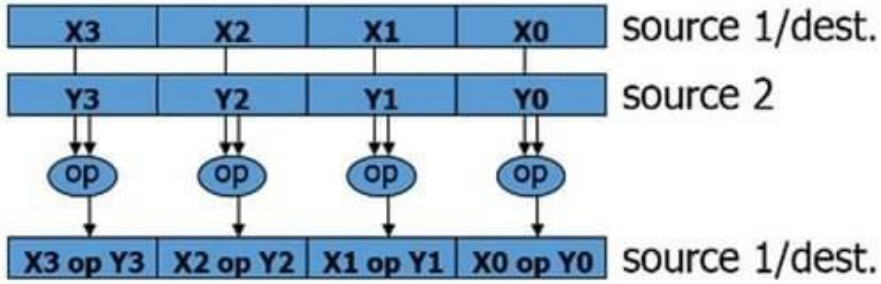
CPU

SIMD

1 instruction – multiple data

SSE2/3/4 – Neon – Altivec

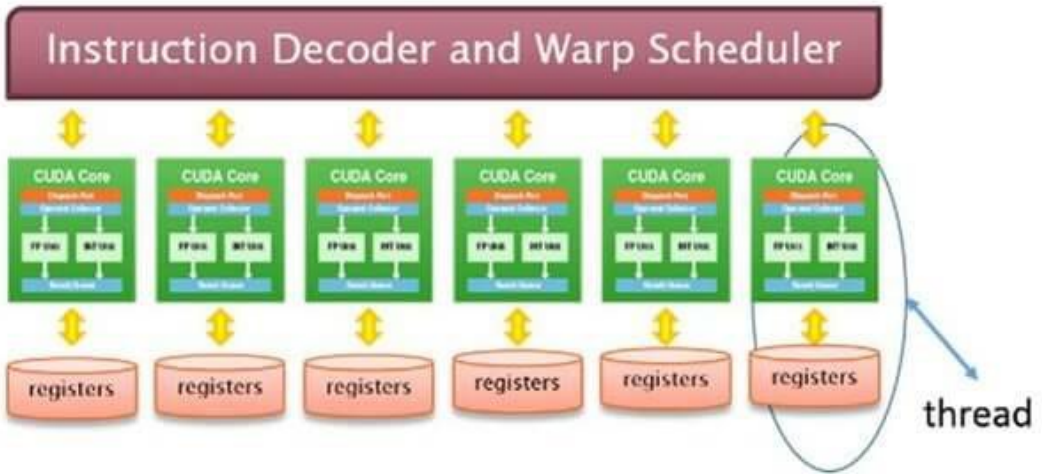
AVX – AVX2...



GPU

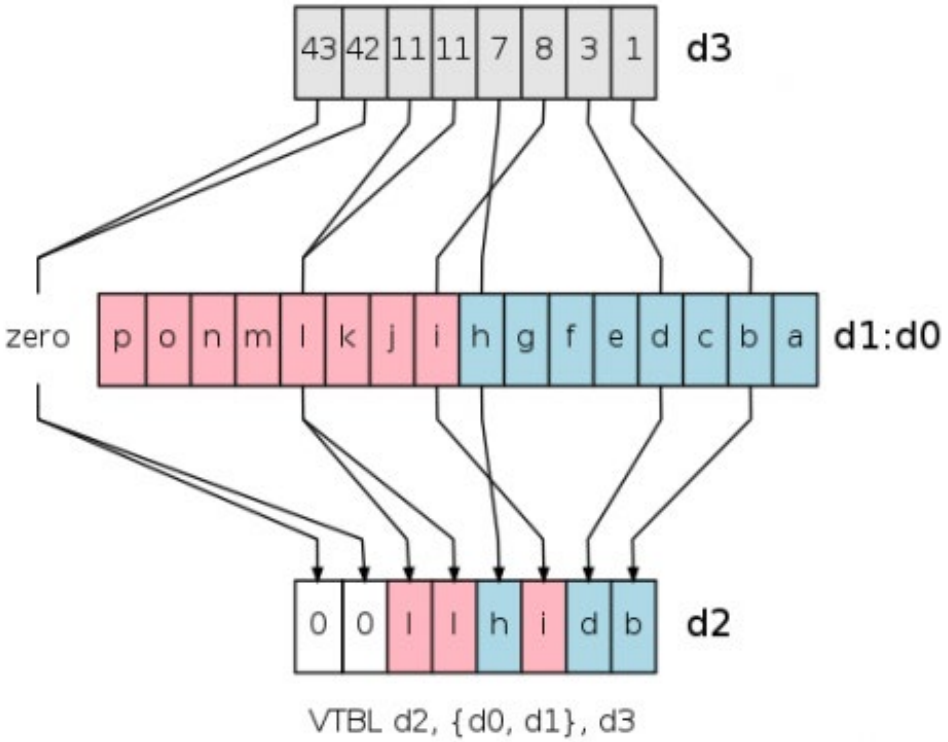
SIMT

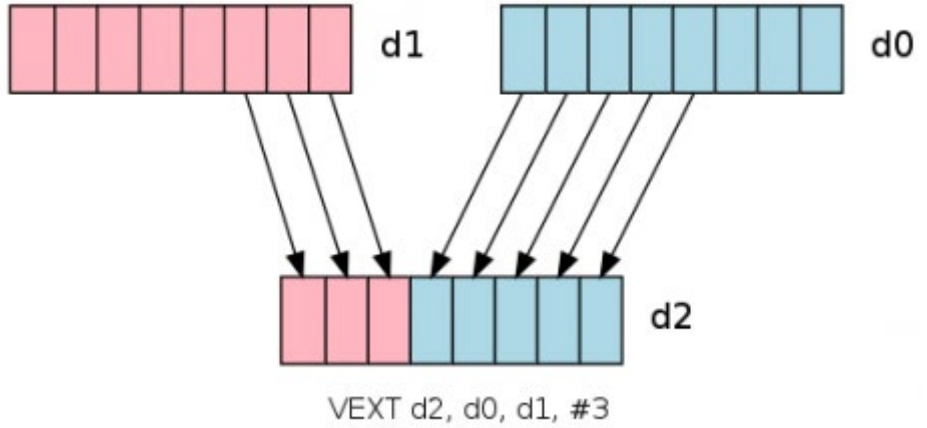
1 instruction – multiple threads

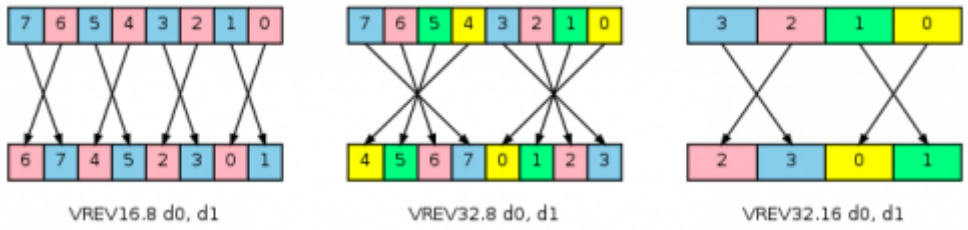


Off-line architectures and data bottleneck – Data Rearrangement

Data rearrangement - data in memory needs to be arranged before reaching the inputs of compute elements (the compute element function is fixed -> input/out format is fixed)



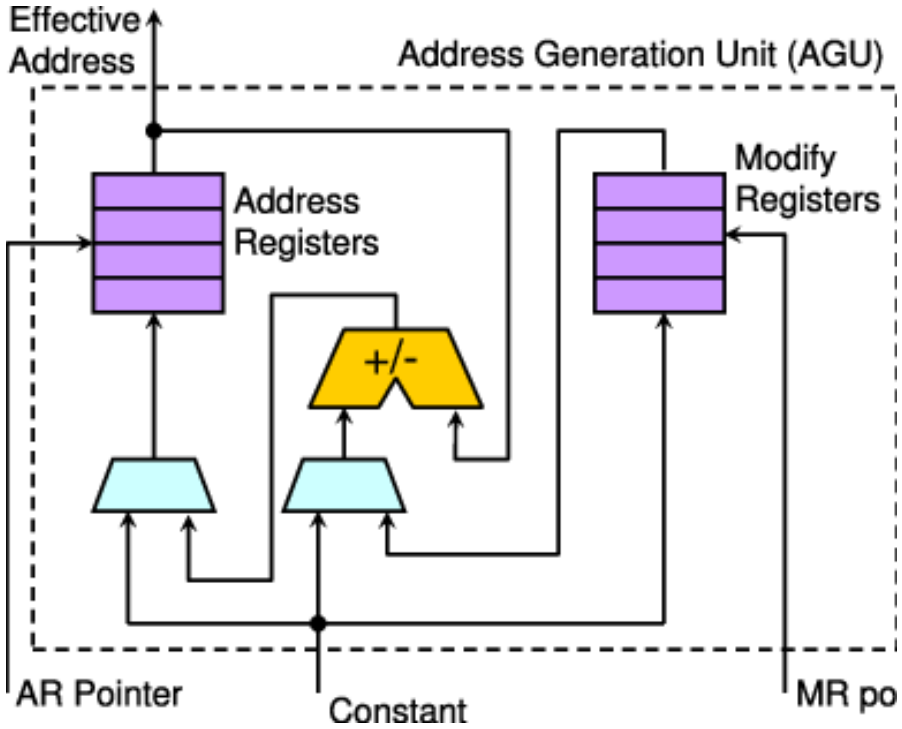




https://developer.arm.com/documentation/102159/0400/Permutation---rearranging-vectors

Off-line architectures and data bottleneck – Address Calculation

Address calculation – next data address calculation is part of the data path, the compute units are used for address calculation



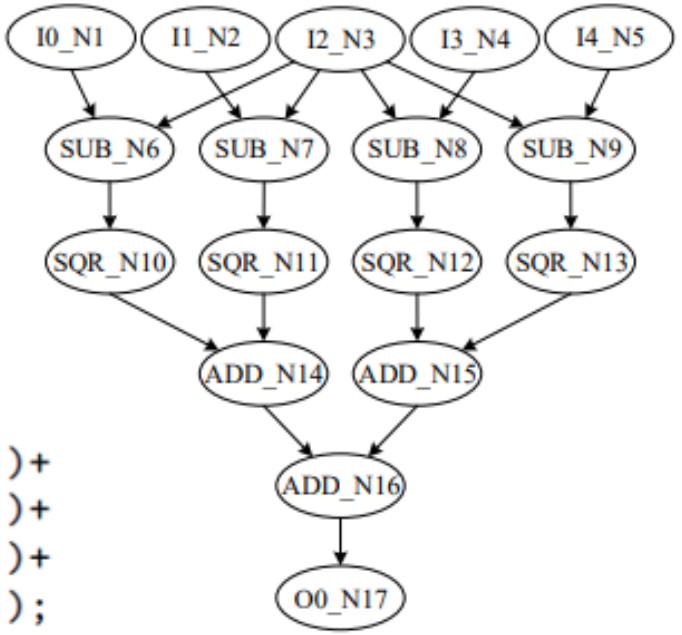
2006 Ozturk, Ozcan all. An ILP based approach to address code generation for digital signal processors

Off-line architectures and data bottleneck – Compute Elements

Compute Elements - (ALU/MACs) - delimiting compute elements at register interface exacerbates the data bottleneck

```
#define SQR(x) ((x)*(x))
for(int i=0;i<64;i++)
  for(int j=1;j<63;j++)
    for(int k=1;k<63;k++)
      b[i][j][k] =
      SQR(a[i][j][k]-a[i][j][k-1])+
      SQR(a[i][j][k]-a[i][j][k+1])+
      SQR(a[i][j][k]-a[i][j-1][k])+
      SQR(a[i][j][k]-a[i][j+1][k]);
```

(a) C Source Code



(b) Data Flow Graph

Fig. 1: The 'gradient' benchmark.

2016 Li, Xiangwei & all An Area-Efficient FPGA Overlay using DSP Block based Time-multiplexed Functional Units.

Image Processing - Hardware Compute

Image processing – hardware

Custom pixel/point-based processing - no memory



Image processing – hardware

Custom 2D convolution – fixed size, programmable coefficients – x lines memory

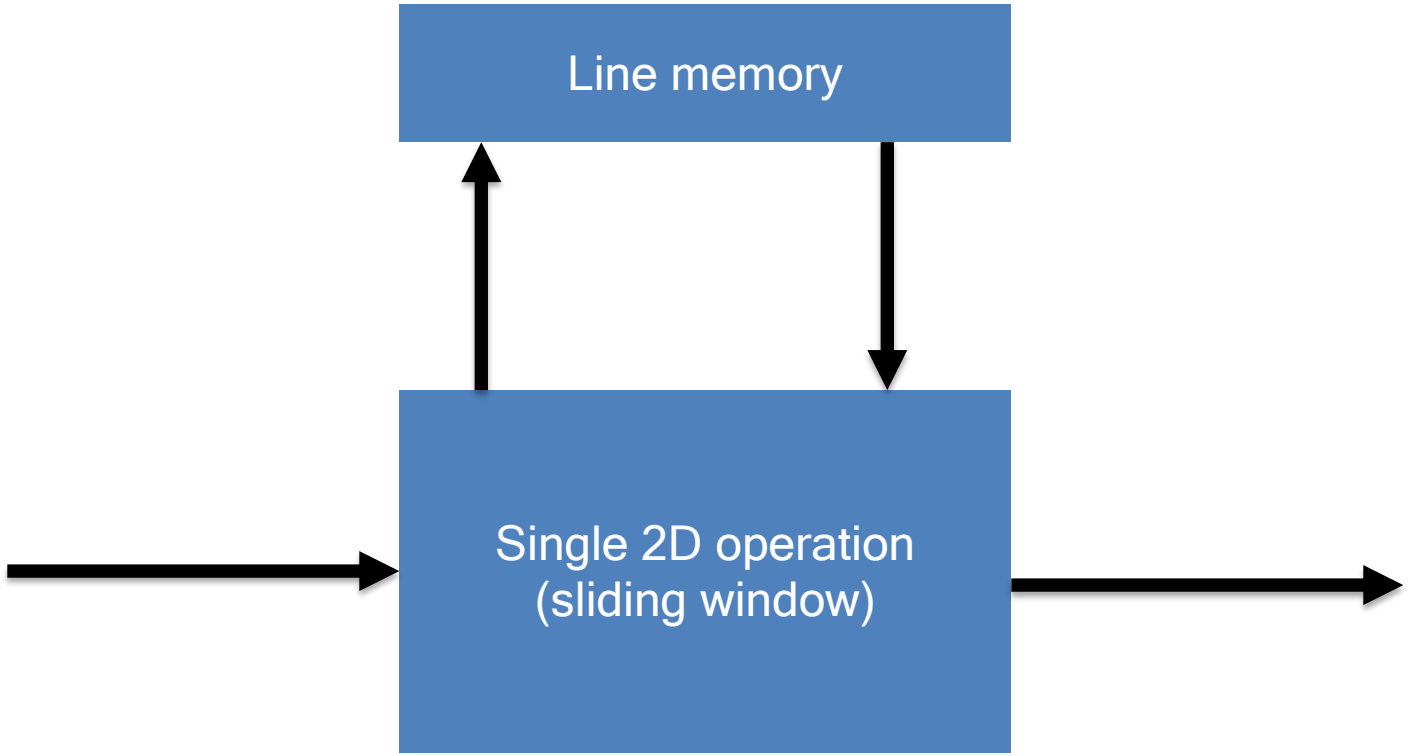


Image processing – hardware (S)

Custom 2D convolution – fixed size, programmable coefficients – x lines memory

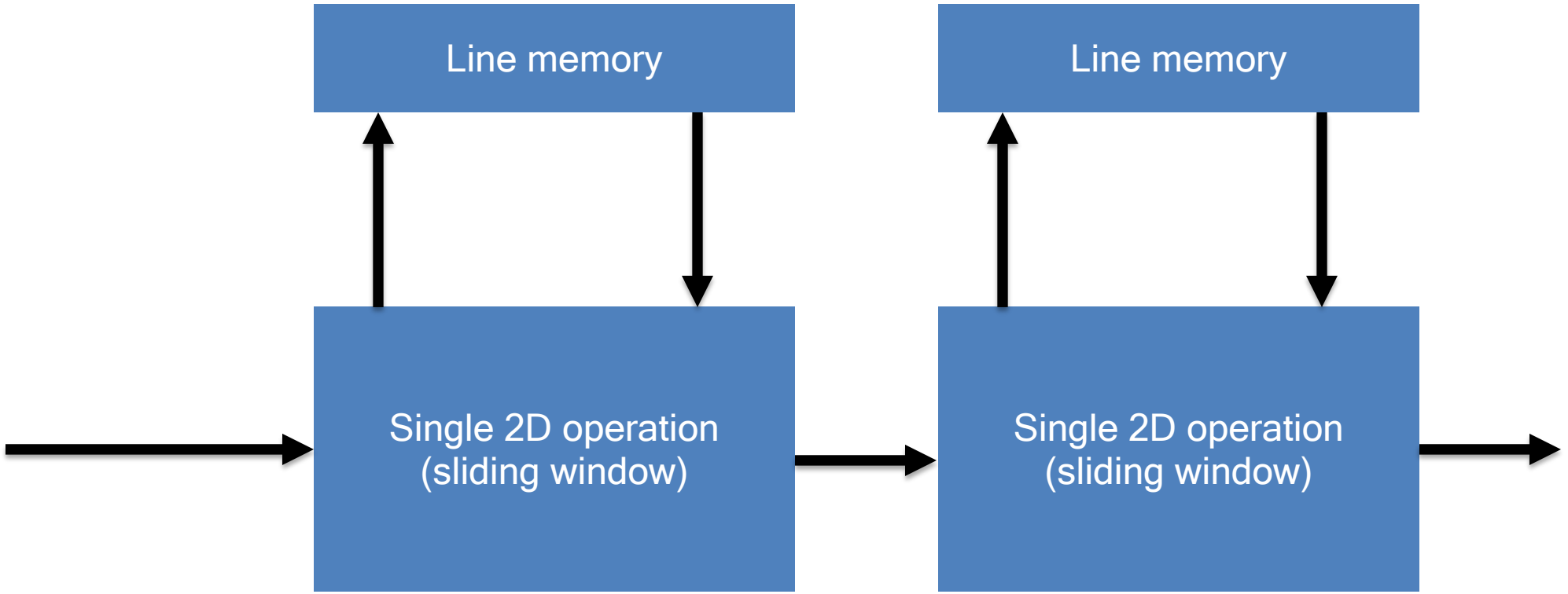
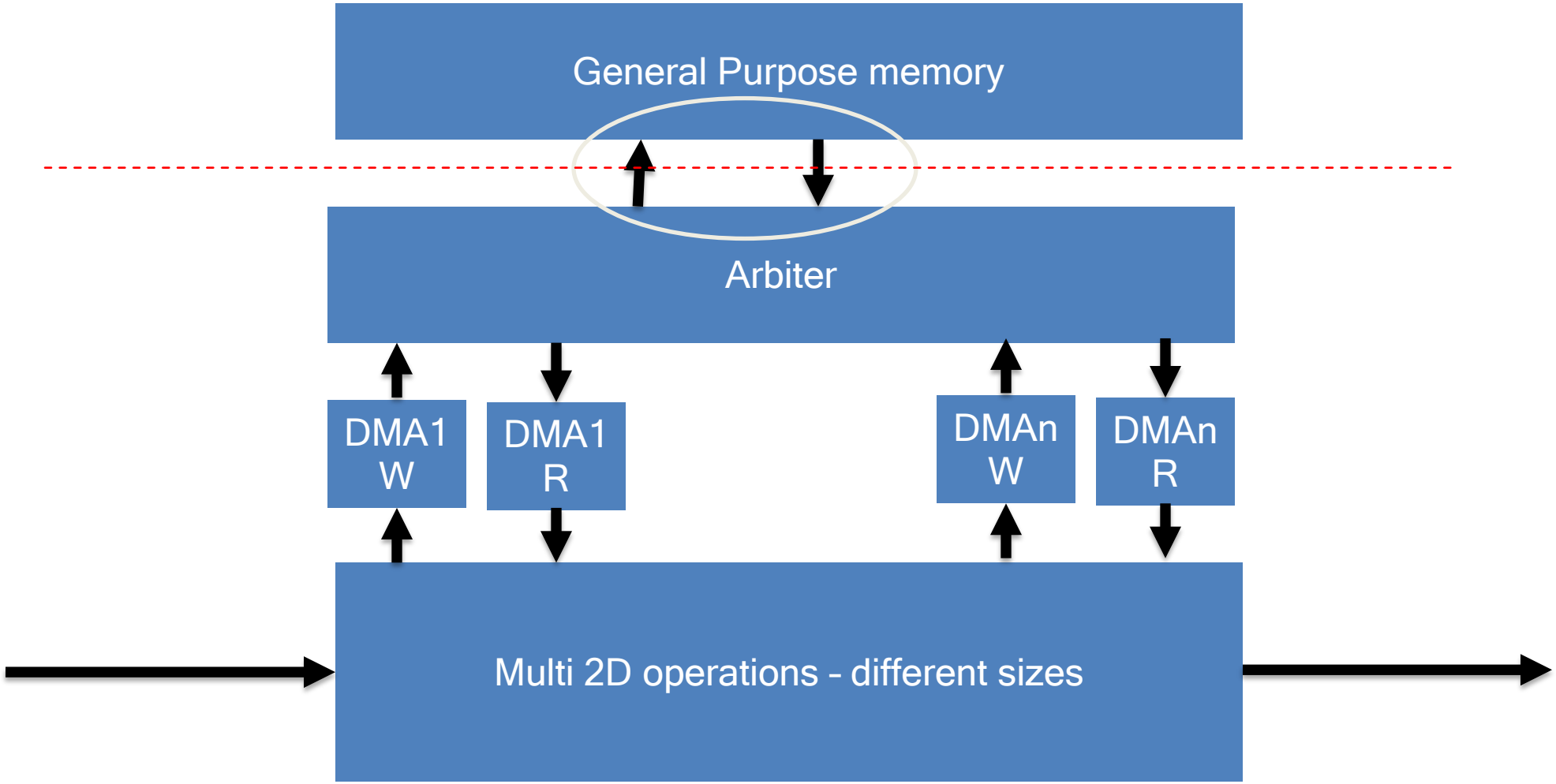


Image processing – hardware (D)

Custom n x 2D convolutions - fixed sizes (different), programmable coefficients - general purpose memory

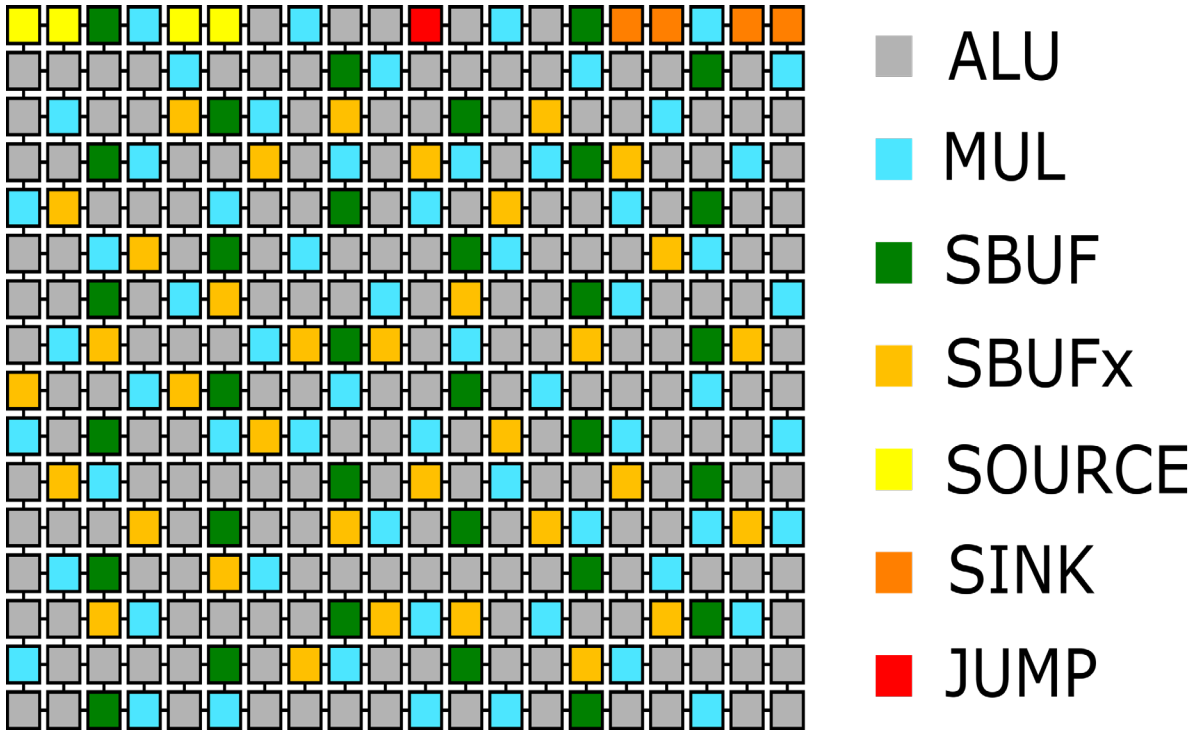


Example: Reconfigurable Data Paths Compute

Example: Reconfigurable Instruction Cell Array (RICA)

coarse heterogeneous FPGA-like, C-programmable - interconnect conditionally updated in one clock

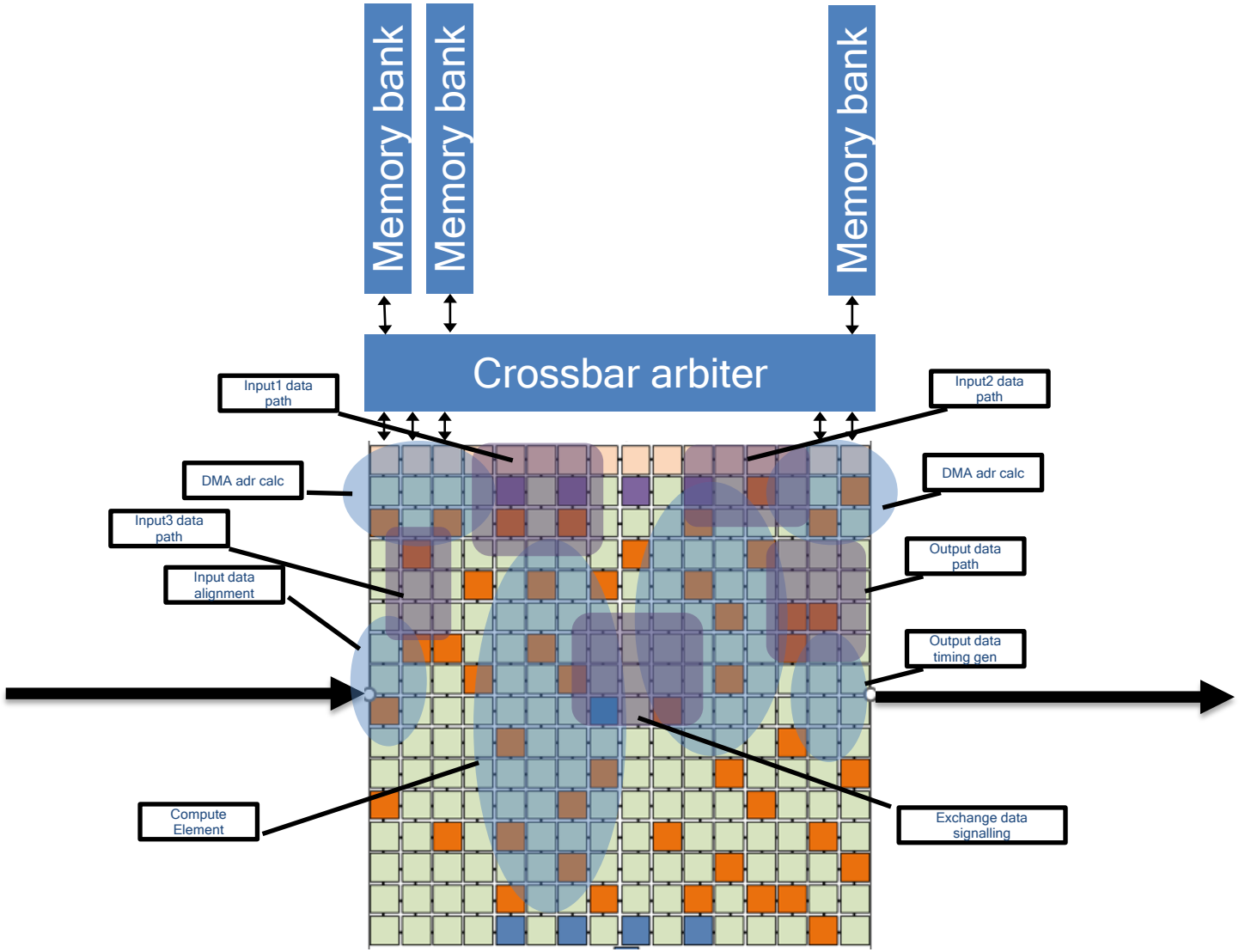
- Coarse-Grain reconfigurable array
- Like FPGA with 16-bit CLBs w ALU
- Instruction cells:
  - ALU, MUL, SBUF, SINK, SOURCE
- Mostly ALU/MUL (83%)
- Computational cells are combinatorial.
  - Timing <1ns
- Vectorized instructions and interconnects x8, x16, x32
- Maps data-paths



2008 S. Khawam. *et al.* The Reconfigurable Instruction Cell Array

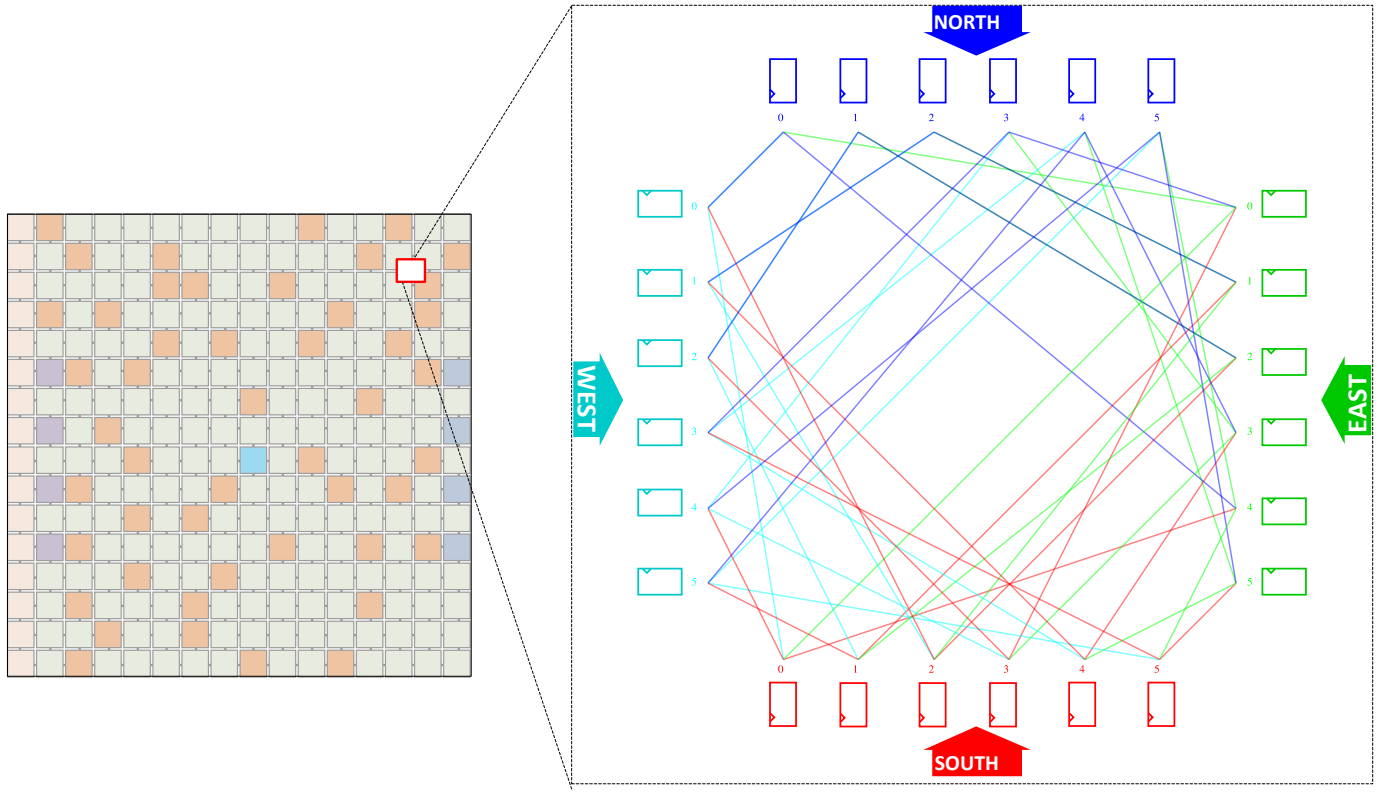
Example: Image processing on configurable hardware

Custom n x 2D convolutions - fixed sizes (different), programmable coefficients - general purpose memory



2008 S. Khawam. *et al.* The Reconfigurable Instruction Cell Array

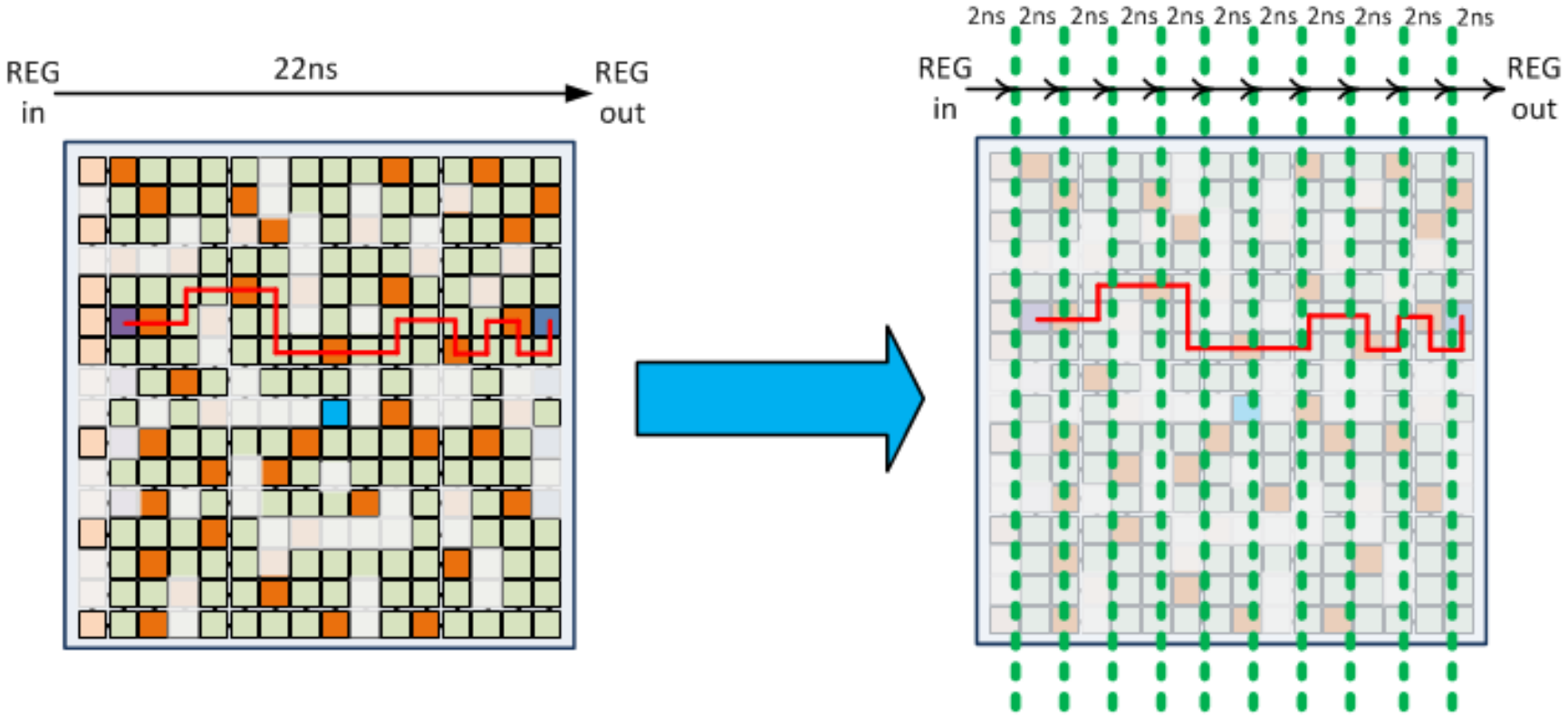
Example: Configurable Interconnect



- Compute cells connected by a sea of interconnect
- SBOXes connect cells to their neighbors
- 6x8-bit channels per side for scalability (Rent's Rule)

2008 S. Khawam. *et al.* The Reconfigurable Instruction Cell Array

Example – Data Path Architecture



- Arbitrary data paths executed in a single clock
- Automatic pipelining for ASIC-like performance
- All instructions execute concurrently
- Data dependencies pose no limitations

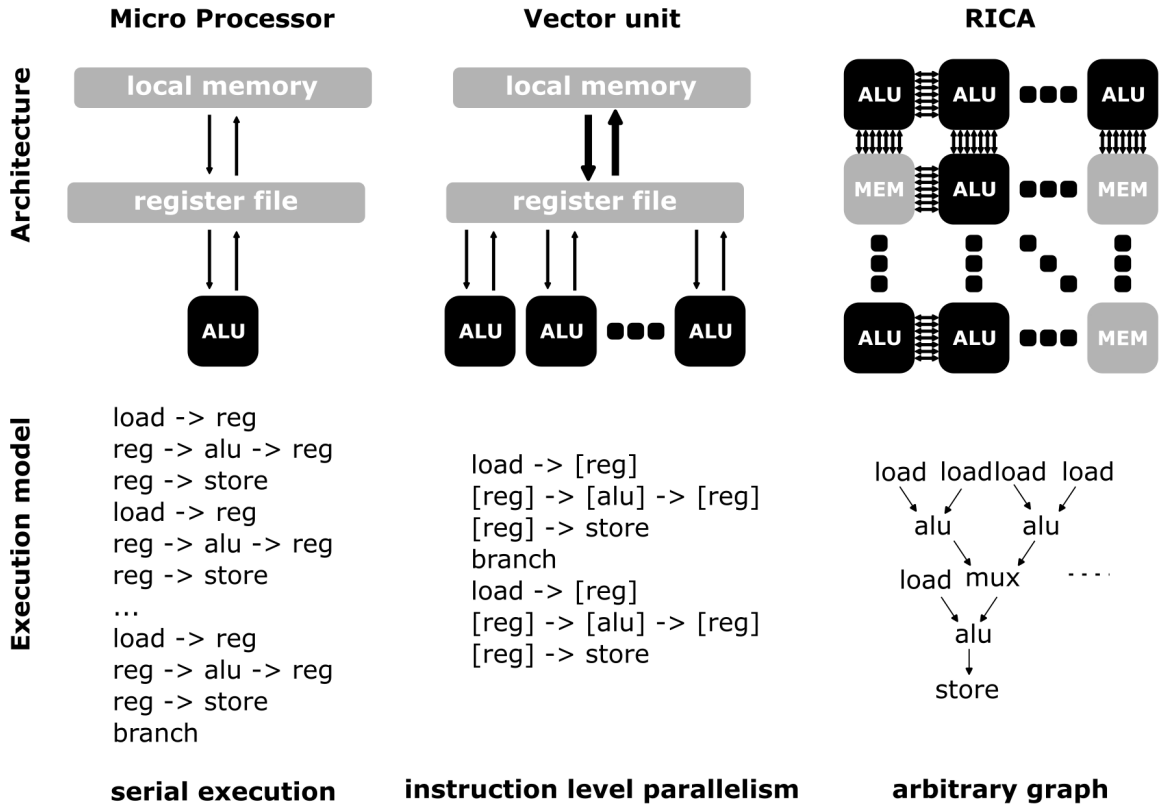
2008 S. Khawam. *et al.* The Reconfigurable Instruction Cell Array

Configurable Compute Element - advantages

Example: Compute Element Architectural Advantages

- Evolution of compute: from single operand to graph execution

- Micro processor (fixed interconnects)
  - One instruction at a time = slow
  - Instruction fetch overheads = power
  - Centralized register file = power
- Vectorized (fixed interconnects)
  - Instruction level parallelism (SIMD or MIMD) = fast
  - Instruction need to be data independent = low util
  - Centralized register file = power
- Reconfigurable Compute/Interconnects
  - Arbitrary graph execution, both dependent and independent instructions = fast, high util
  - Minimal instruction fetches = efficient
  - Programable pipelining = fast
  - Distributed registers = efficient, fast
  - Distributed memory load/store = efficient, fast



2008 S. Khawam. *et al.* The Reconfigurable Instruction Cell Array

Architectural Advantages

Instruction-Level Parallelism (ILP)

Configurable Interconnects:

- Every operation in a filter kernel is executed in parallel, one pixel per clock.
- High-throughput can be achieved with a low operating frequency => low power.
- Algorithms become memory bandwidth limited.
  - Entire filter clocked at the same frequency as the memory.
- Instruction cache is mostly idle => very low power

Microsequencing (RISC+SIMD or VLIW):

- Parallelism available via a vector of similar execution units which can execute in lock-step, each on different data.
- Clocked at a high frequency, to achieve high-throughput => higher power.
- Each cycle performs different operation types, so instruction cache is highly active => higher power.

Architectural Advantages

Resource Utilization

Configurable Interconnects:

- Typical filters are upside-down triangles:
  - Operations are 2-input -> 1-output.
  - Each level has fewer intermediate results than the last.
  - Filters: Many inputs, gradually reducing to one output.
- Interconnect fabric allows data to flow in any direction through the pipeline.
- Utilization is NOT dependent on the 'shape' of the filter kernel.
- Therefore, high utilization achievable with real-world applications.

Microsequencing (RISC+SIMD or VLIW):

- Early levels of operations achieve high utilization of ALUs.
  - But subsequent levels achieve gradually less utilization.
- Partially mitigated by thread-level parallelism (TLP):
  - Unused sections of the vector can be used by other threads.
  - Challenges with memory bandwidth and intermediate results.
- Full parallelism (utilization) is hard to achieve in real-world applications.

Architectural Advantages

Intermediate Results

Configurable Interconnects:

- Sent combinatorially to their consumer via the interconnect.
- Interconnect can reach multiple destinations - extended via multiple hops - with no switching activity on unused destinations => low power.
- Each hop of the interconnect can be optionally pipelined via a programmable register to reduce critical path. Single register => low power.
- Native support for fan-out and re-ordering => no additional power.

Microsequencing (RISC+SIMD or VLIW):

- Stored in a register-file, on every operation boundary.
- Register-file depth proportional to how far away the next operation can be (delay).
- Register-file access-power scales with depth, irrespective of how many slots are actually used by a program.
- Limited ability to pass results from one lane to another => more cycles needed.

Architectural Advantages

Memory Bandwidth

Configurable Interconnects:

- Memories are typically used only for input history and look-up tables.
- All other data is in-flight in the interconnect.
- One copy of each filter => no multiplication of memory bandwidth.
- Large filters can be split into multiple configuration contexts => introduces new input history.
- Minimal memory activity => low power.

Microsequencing (RISC+SIMD or VLIW):

- Tiling multiplies the memory bandwidth requirement.
- Operations of a filter kernel are spread across many clock cycles.
- This means lifetime of intermediate results can also be many clock cycles.
- Fan-out increases register pressure.
- If register-file depth is exceeded, must spill into memory (L2 cache).
- Extra memory accesses => higher power.

Conclusions

Von Neumann architecture was envisioned to operate on off-line data – data persistency and processing latency were not a constrain

Today's CPUs (CISC, RISC, GPU, DSP, VLIW) inherit the fundamental constrains present in von Neumann architecture and are not optimal in real-time image processing applications and require a revolutionary change.

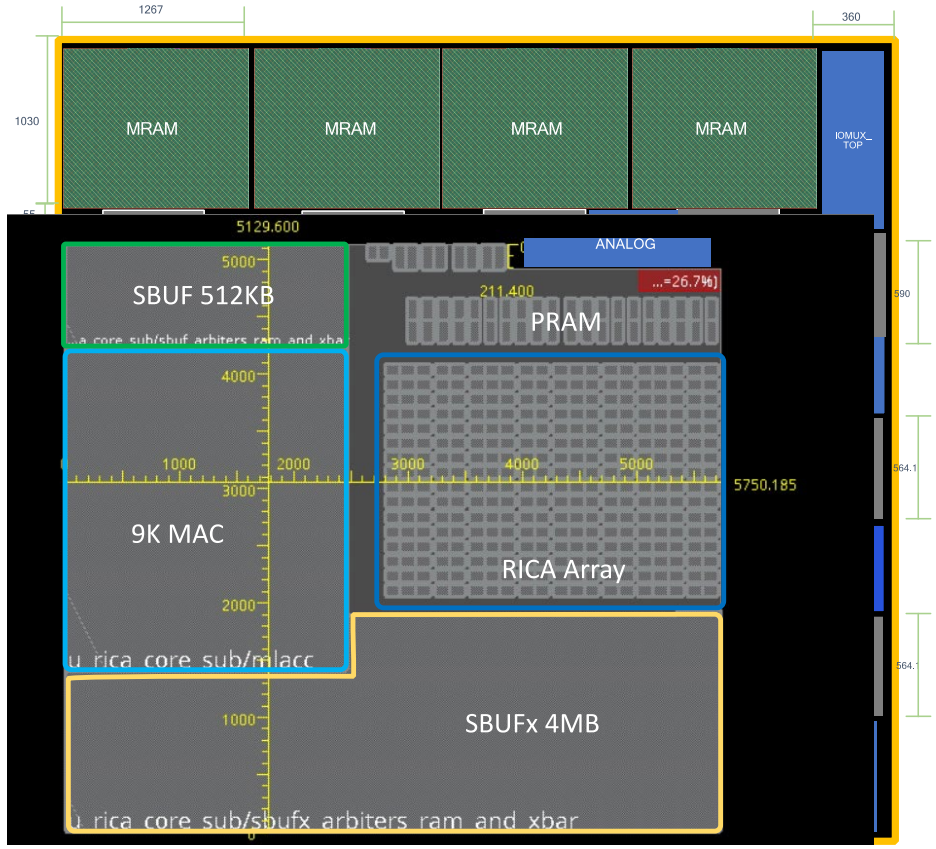
Programmable interconnects are successful in solving the data bottleneck by removing the need of multi pass data rerouting.

Additional Credits and Acknowledgments

Programmable Architecture Example (RICA-ML):

Mark Muir, Ioannis Nousias, Sami Khawam, Biay-Cheng Hseih

Qualcomm Technologies, Inc.



Thank you – Q&A

Back-up slides

Configurable Compute Element

Example: Algorithm mapping

Compute blocks: hardcoded, optimized and programmable in function

Data paths: fully programmable as netlists

Control paths: microsequenced – high-level language

Example: Source Code -> Compiler -> Mapper -> Place and Route -> Bitstream

Source code

```
input = source(0);

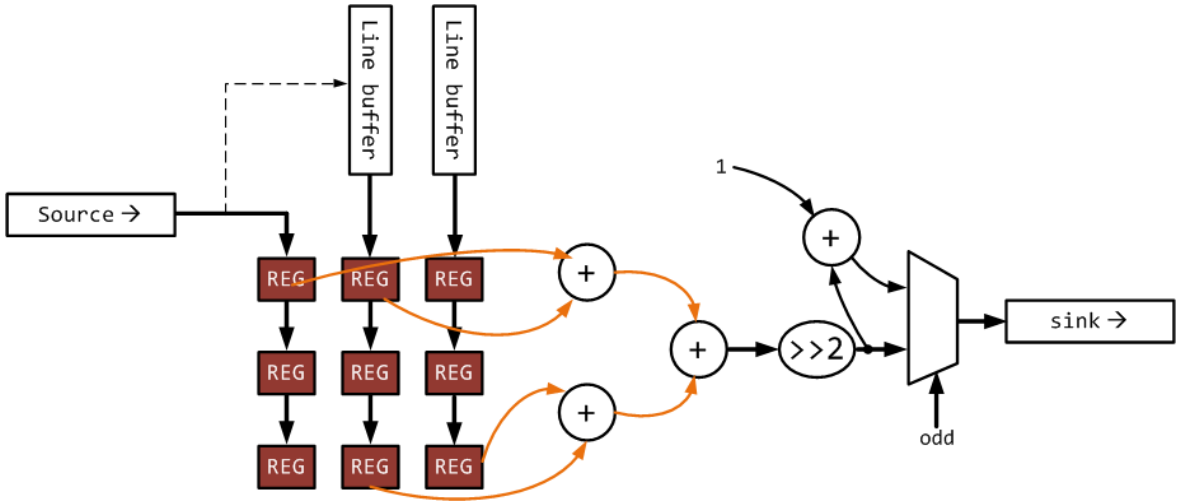
shift_window(block, input);

output =(block[0][0] +
         block[0][1] +
         block[2][1] +
         block[2][2]) / 4;

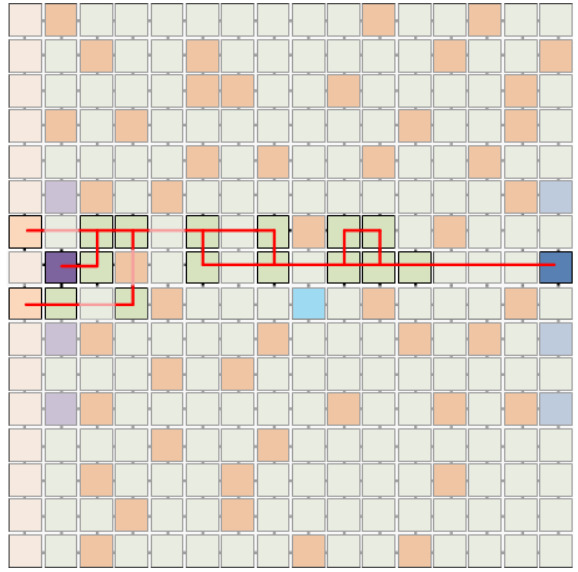
output = odd ? output + 1 :
               output;

sink(output);
```

Compiled data flow

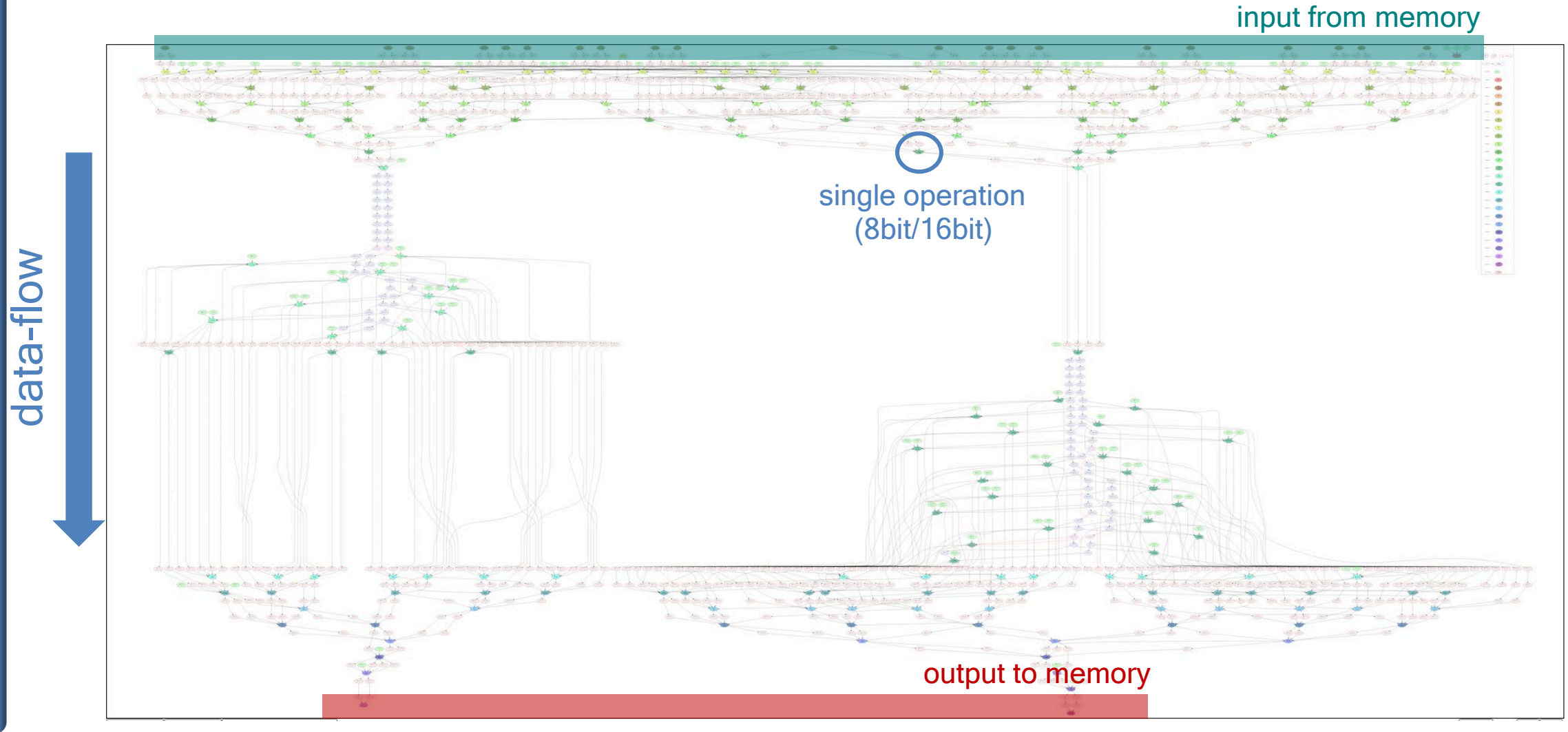


Place & Route



Computation Example - Arbitrary Graph Execution

- 6DoF gaussian pyramid. Two scales in parallel: 11x11 and 21x21 gaussian filters.



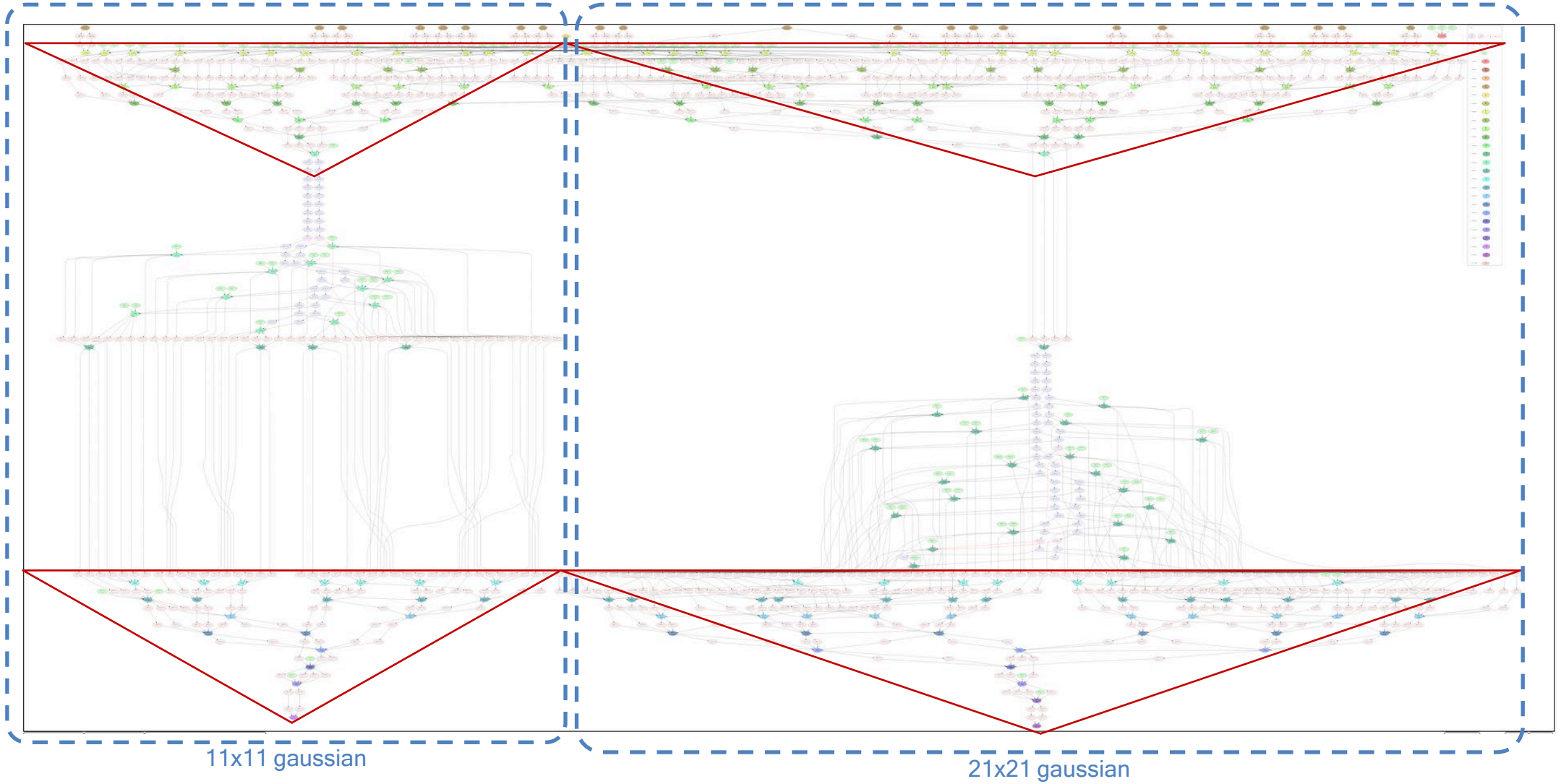
Computation Example - Arbitrary Graph Execution (Cont)

- 6DoF gaussian pyramid. Two scales in parallel: 11x11 and 21x21 gaussian filters.



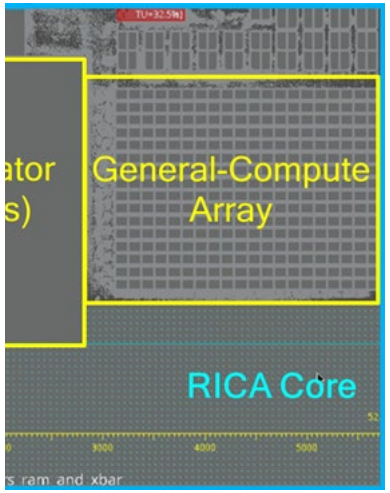
Computation Example - Arbitrary Graph Execution (Cont)

- 6DoF gaussian pyramid. Two scales in parallel: 11x11 and 21x21 gaussian filters.

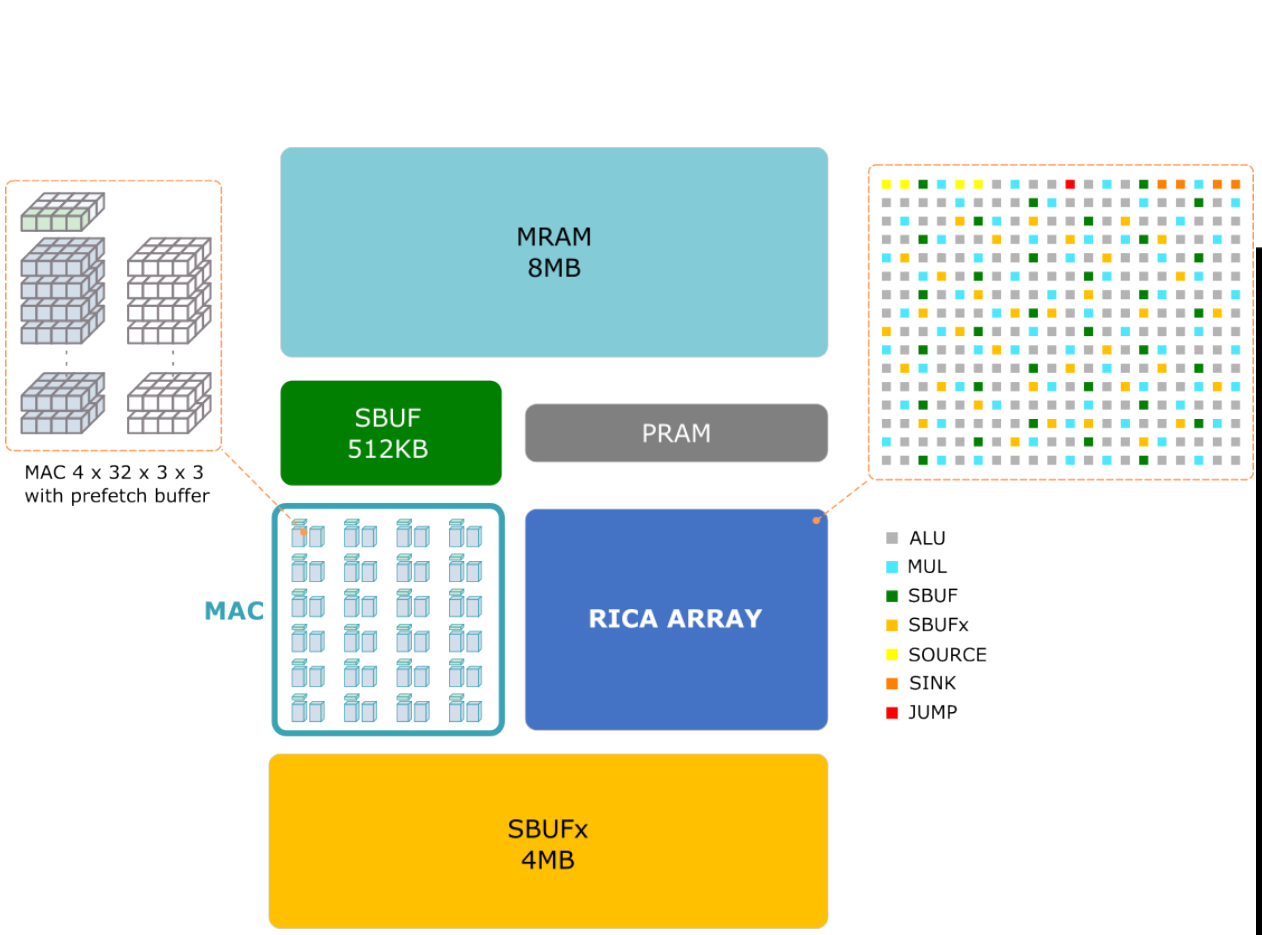


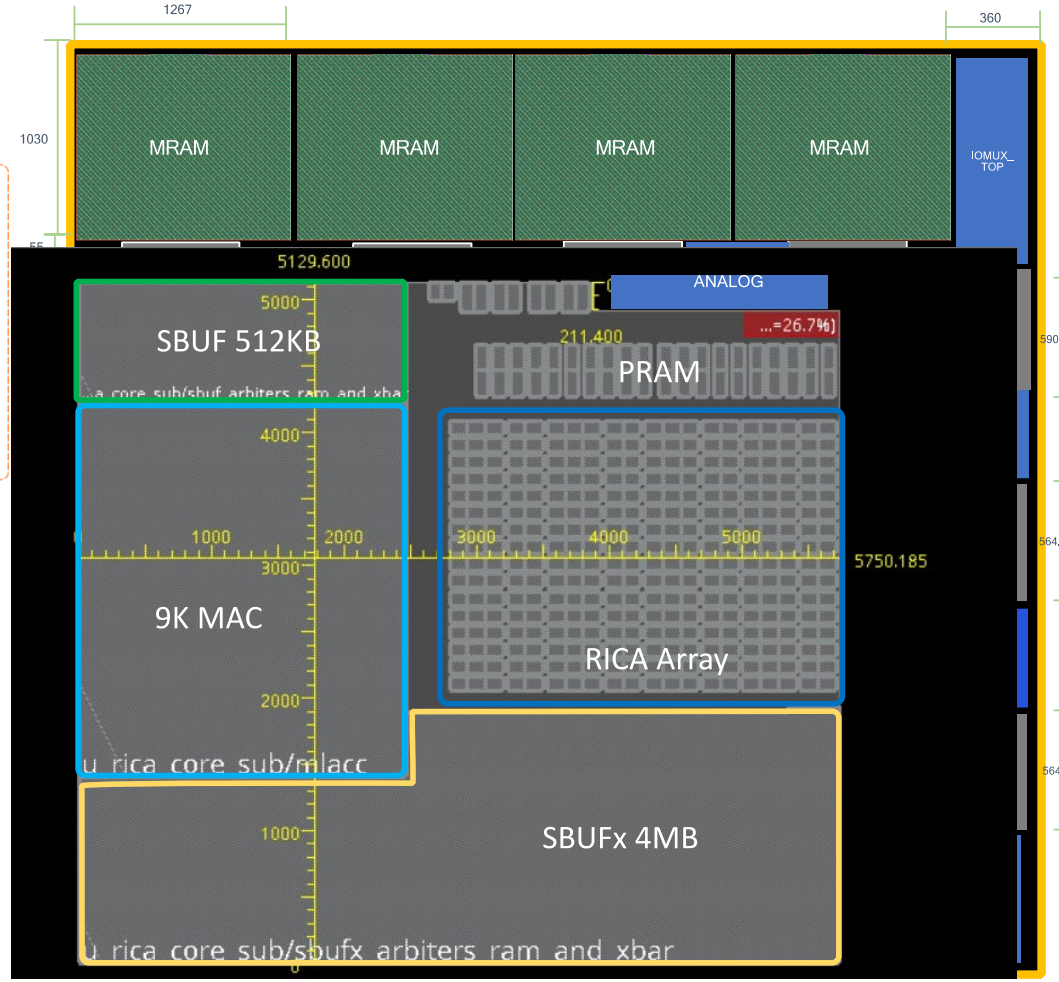
Compute Core with ML - layout and performance

Spec	RICAML - Core
Technologies	22nm ULL, 0.8V Vdd
Memory	250K-byte PRAM, 4.2M-byte data RAM 8M-byte e-NVM (MRAM)
Op Freq.	330 MHz
Data Input	MIPI RX & LVDS
Data Output	MIPI TX & LVDS
Equiv. Compute Capability	180 GOPS baseline, MAC Extension to 5TOPS with ML Accelerator
Power	LP Always-ON-mode: ~5mW @5fps
Key Features & Appl.	True image sensor edge-computing; Always-On LP operation; MobileNet2, CNN face-detection, hand/gesture detection, CV, SLAM,....



Compute Core – ML architecture and SoC integration





Compute Core – ML benchmarks:
MobileNetV2 1.0 scale, 224x224 input, 3.4M parameters, 300M MAC per inference.

