Domain Specific Accelerators

Prof. Isabel Ojalvo

Spring 2025

What? **Why?** **How?** **Future?**


What?
Why?
How?
Future?

Specialized Hardware is Everywhere

Specialize hardware does most of the work, but it is mostly invisible

Cellphones

- Software on ARM cores
  - High-complexity, low compute work
- Accelerators do the heavy lifting
  - MODEMs
  - CODECs
  - Camera Image Processing
  - DNNs
  - Graphics

Graphic Processing Units (GPUs)

- Rasterizer
- Texture filter
- Compositing
- Compression/ Decompression
- Tensor computations

Others

- Neural Network Processors
- Processors for Software-Defined Networks (SDNs)

Specialized Hardware is Everywhere

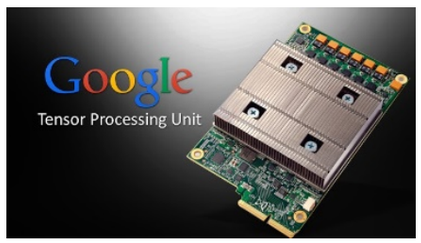
- Domain Specific Accelerator
  - Achieve higher efficiency by tailoring the architecture to characteristics of the domain
  - Not one application, but a domain of applications
- Different from strict ASIC since still runs software
- Specialized hardware? Yes or No?

Intel's 12th Gen "Alder Lake" 10nm Desktop CPU

In your desktop?

NVIDIA RTX A6000 Workstation Graphics Card

Xilinx Alveo U280 Data Center Accelerator Card



Google's Tensor Processing Unit (TPU)

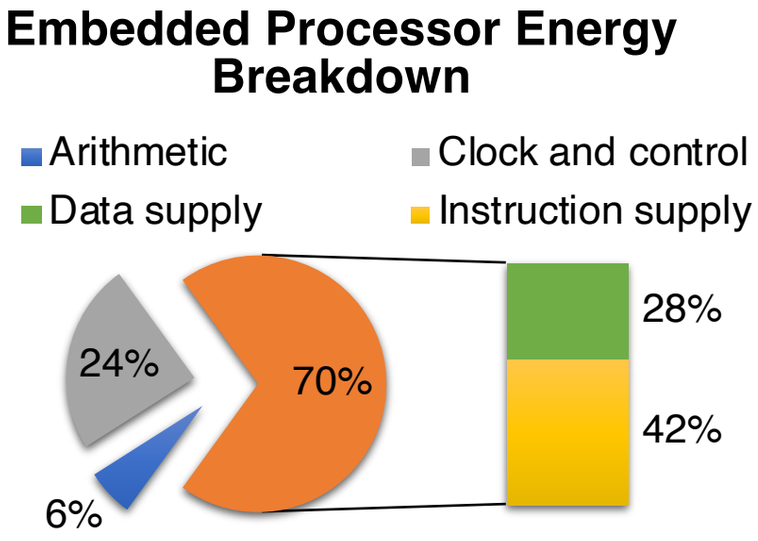
What? **Why?** How? Future?

Source of Computation Inefficiencies

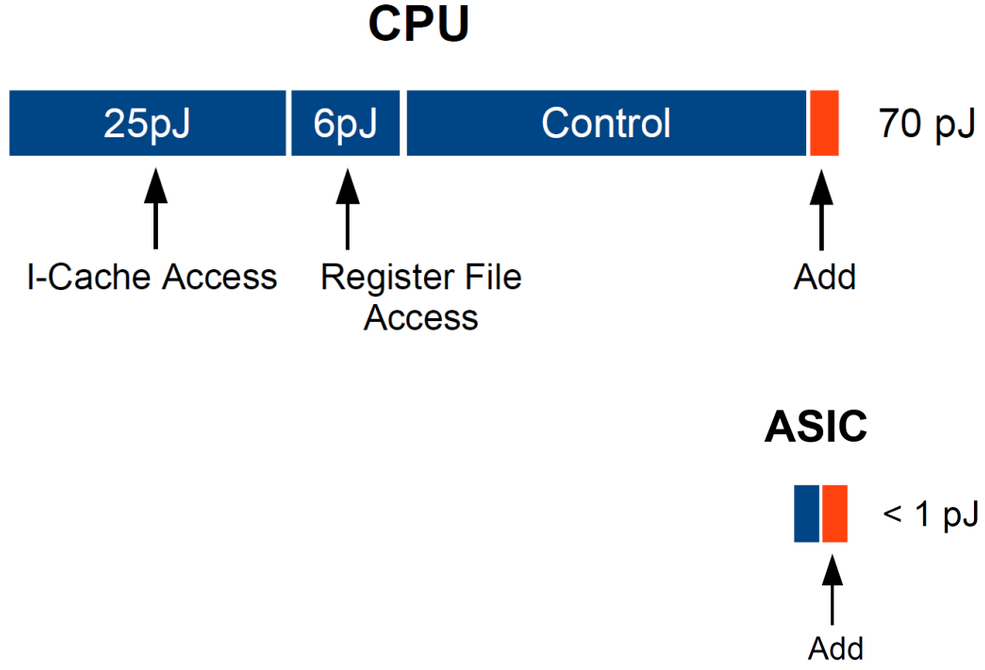
Inefficiency of General-Purpose (GP) computing

- Typical energy overhead for every 10pJ arithmetic operations
  - 70pJ on instruction supply
  - 47pJ on data supply

Only 59% of the instructions are arithmetic



Source: Dally et al. Efficient Embedded Computing, IEEE'o8

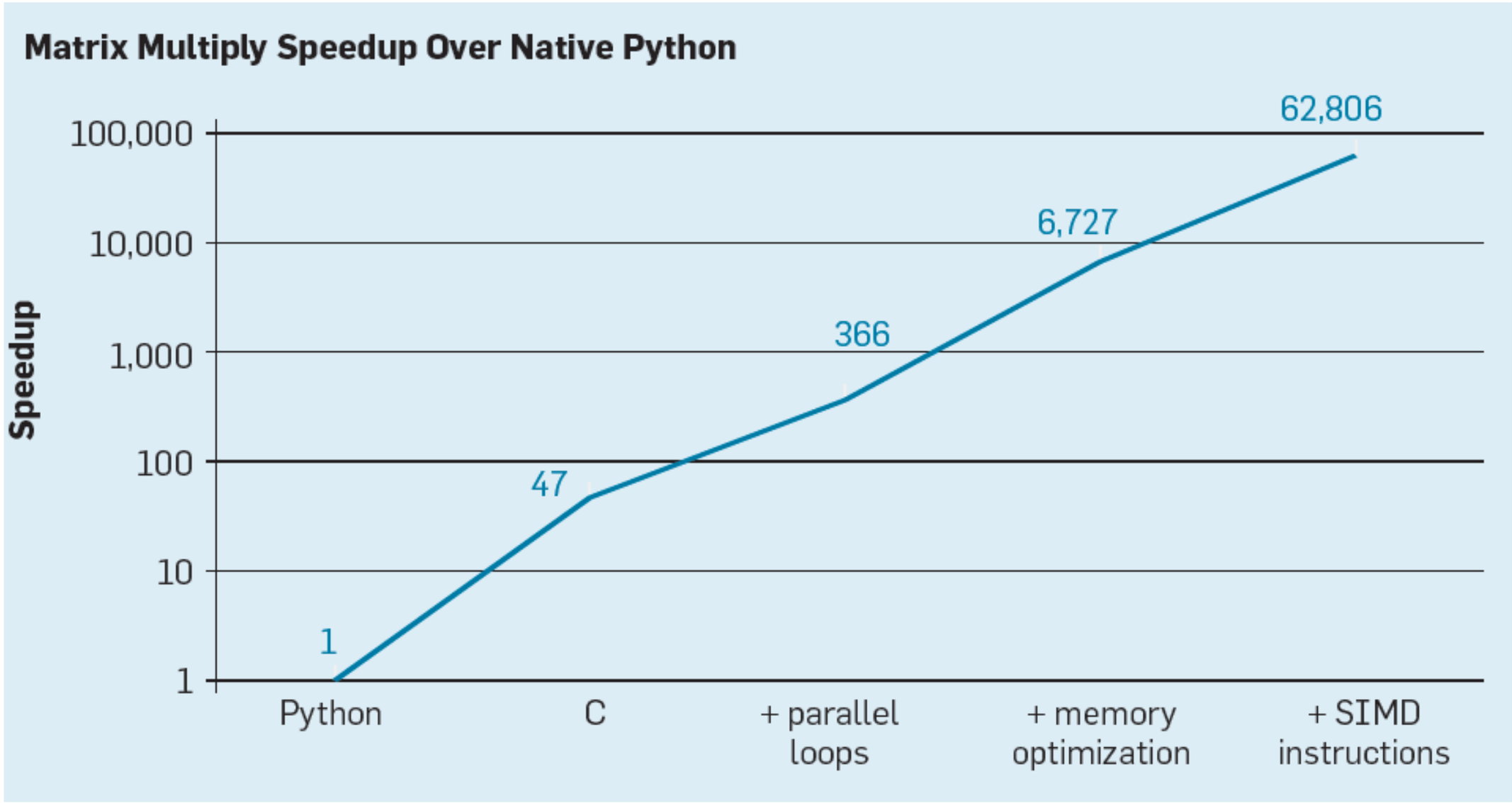


[M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," IEEE ISSCC, 2014, pp. 10-14]

Source of Computation Inefficiencies

Inefficiency of High-Level languages

- Better interpretability, lower efficiency



[Leiserson, C. et al. There's plenty of room at the top. Science, June 2020, Vol 368(6495)]

Productivity vs. Efficiency

There exists a big gap between

- Modern languages: emphasizing productivity
- Traditional languages: emphasizing performance

E.g., Verilog

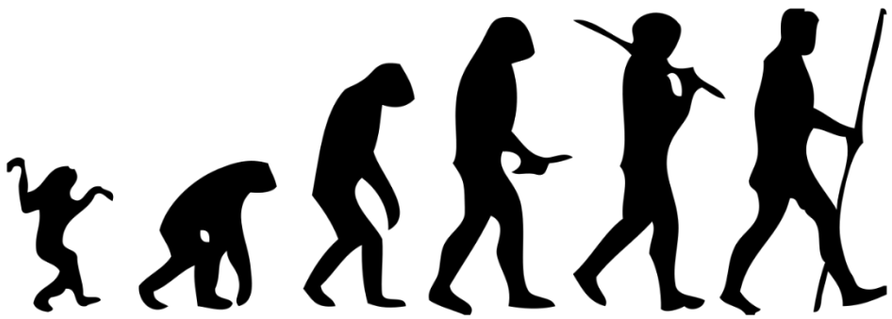
Traditional approaches: low productivity,
high efficiency (e.g., assembly)

Modern languages: high productivity,
low efficiency (e.g., Python)

Solutions to Computation Inefficiency

For humans, remarkable advancement of civilization via specialization

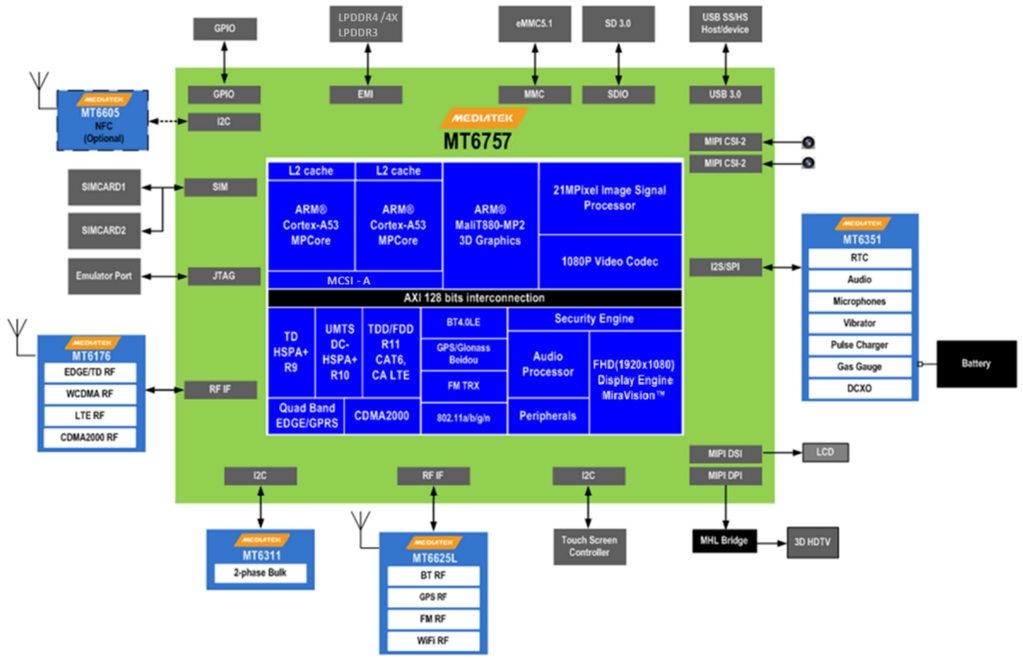
- Not through scaling – number of brain neurons and their firing rate did not change significantly



Specialization: Advance of Civilization

For computers

- Modern System-on-Chips (SoC) integrate a rich set of specialized accelerators
  - Speed up critical tasks
  - Reduce power consumption and cost



Solutions of Computation Inefficiency?

Hardware-centric approach: Domain-Specific Accelerators (DSAs)

Software-centric approach: Domain-Specific Languages (DSLs)

- Make vector, dense/sparse matrix operations explicit
- Help compiler to map operations to the processor (general? DSA?) efficiently
- DSL Examples
  - Matlab: for operating on matrices
  - TensorFlow: dataflow language for programming DNNs
  - P4: for programming SDNs
  - Halide: for image processing specifying high-level transformations

[J. Rexford, "p4: Programming Protocol Independent Pack Processors" https://www.cs.princeton.edu/~jrex/papers/P4-ccr14.pdf

Best combination: DSL + DSA + Automation

Best of Times for Specialized computing

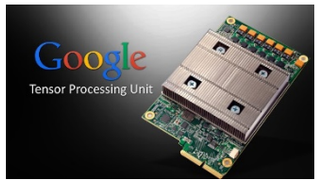
Pressing demand to efficiently accelerate a growing array of datacenter & embedded workloads

- Multicore performance scaling significantly slowed
- Machine learning spread + big data + sophisticated algorithms + …

Intel's 12th Gen "Alder Lake" 10nm Desktop CPU

NVIDIA RTX A6000 Workstation Graphics Card (in my lab)

Xilinx Alveo U280 Data Center Accelerator Card



Google's Tensor Processing Unit

Flexibility (easy programming) ⟷ **Efficiency (high performance)**

Implementation of Specialized Computing

Pressing demand to efficiently accelerate a growing array of datacenter & embedded workloads

- Multicore performance scaling significantly slowed
- Machine learning spread + big data + sophisticated algorithms + …

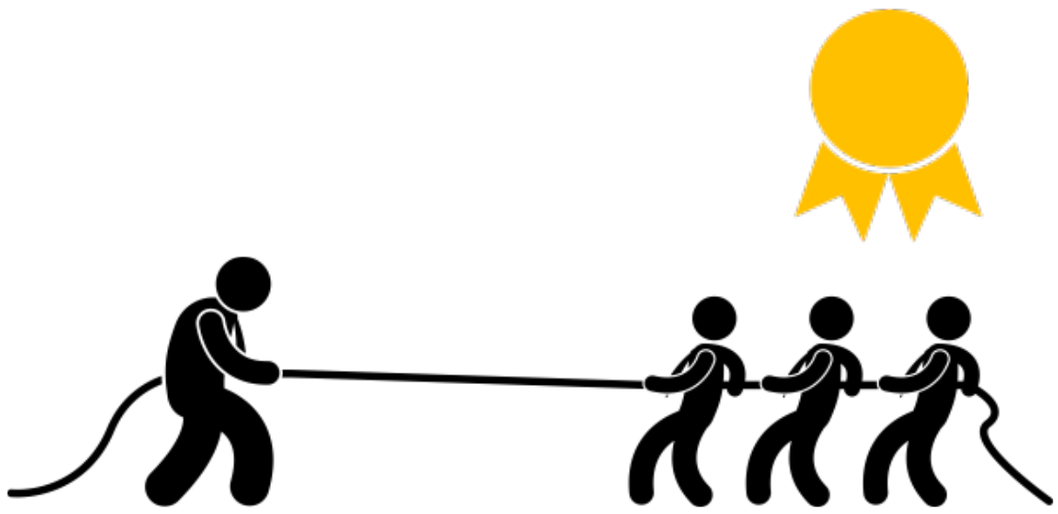
But – also worst of times? Conventional hardware design practice requires extensive hand-coding in RTL and manual tuning

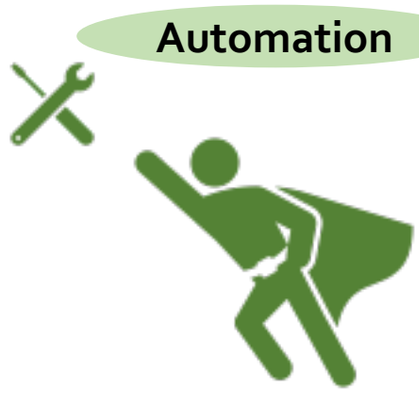
Platform	Normalized Speed-Up	Normalized Performance/Watt	Development Time in Days
FPGA*	545:1	1090:1	60
GPU	50:1	21:1	3
GPP	1:1	1:1	1

Verilog is hard…

Productivity vs. Efficiency

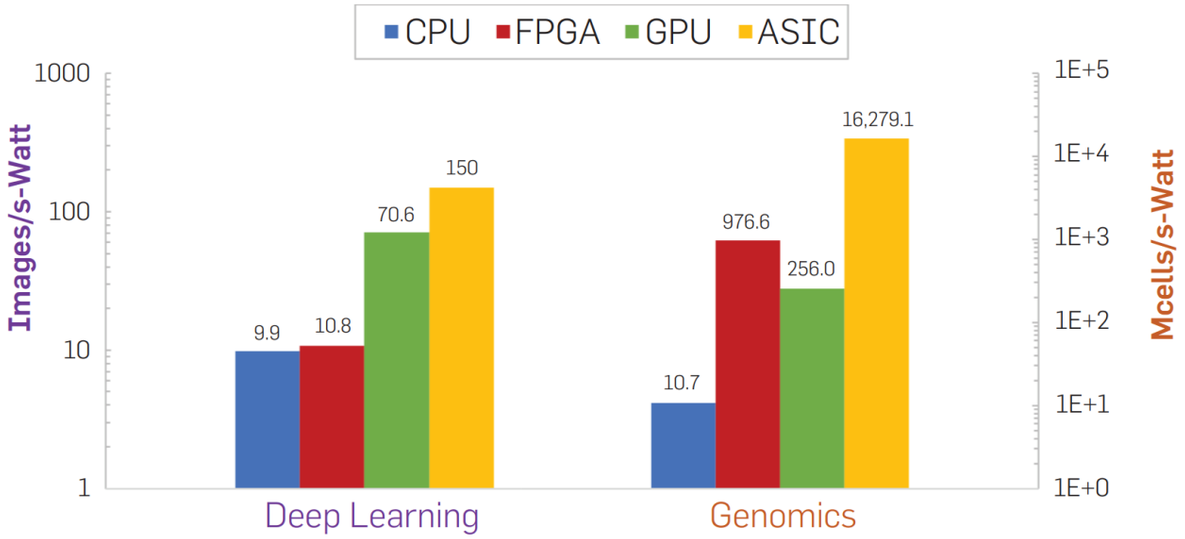
Gap can be largely narrowed by DSL + Automation





Examples of DSAs how good are they?

Deep learning and genomics



[W. J. Dally, et al., Domain-Specific Hardware Accelerators. Communications of the ACM, 63(7), pp. 48-57, July 2020.]

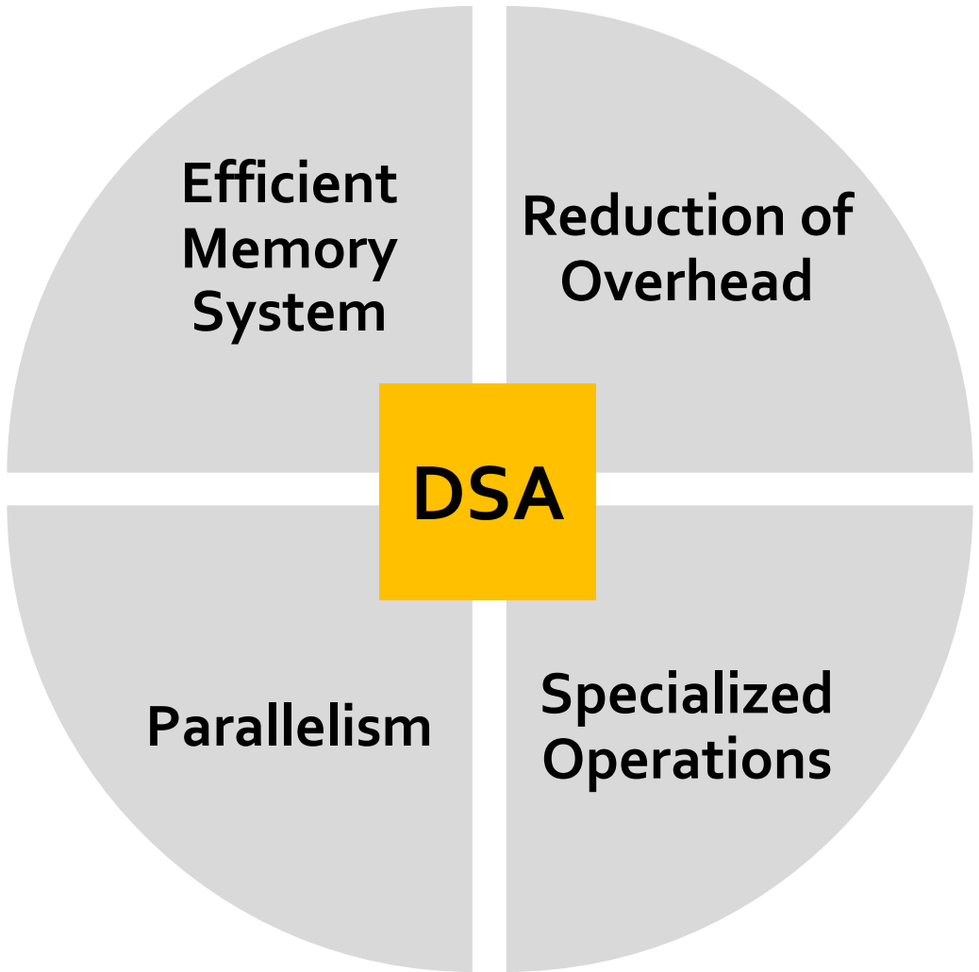
Another genomic algorithm (Banded Smith-Waterman) example

- In CUDA for the GPU in one day
  - 25x improvement in efficiency over the CPU
- On an FPGA in two months of RTL design and performance tuning
  - 4x the efficiency of the GPU
- RTL into an ASIC gives
  - 16x the efficiency of the FPGA but with significant nonrecurring costs and lack of flexibility

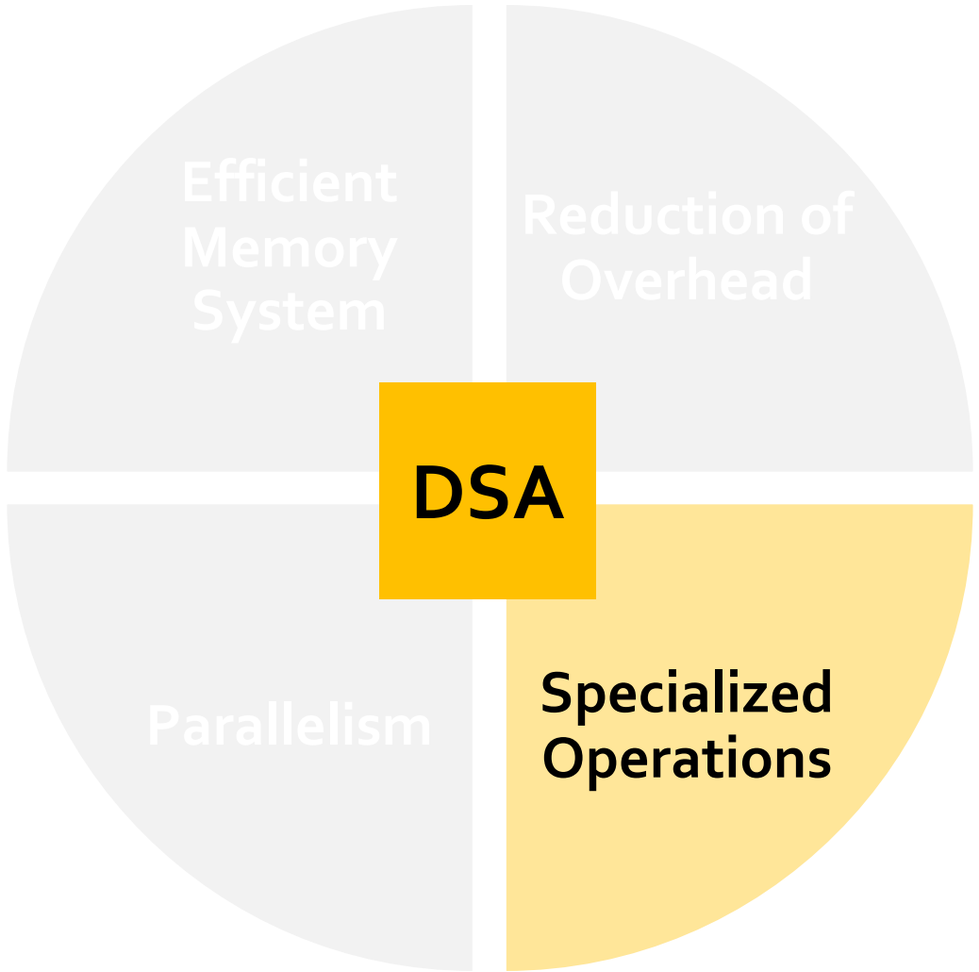
[Turakhia, Y., et al. "Darwin-WGA: A co-processor provides increased sensitivity in whole genome alignments with high speedup. HPCA (2019)]

Domaine Specific Accelerators

Principles of DSAs



Principles of DSAs



Specialized Operations and Data Types

Specialized operations result in orders of magnitude efficiency and moderate speedup

- William Dally, Keynote, “The Future of Computing: Domain-Specific Accelerators”, MICRO52
  - **An example:** Smith-Waterman algorithm for gene sequence alignment

$$I(i,j)=\max\{H(i,j-1)-o,I(i,j-1)-e\} \quad (1)$$

$$D(i,j)=max\{H(i-1,j)-o,D(i-1,j)-e\} \quad (2)$$

$$H(i,j)=max\begin{cases}0\\ I(i,j)\\ D(i,j)\\ H(i-1,j-1)+W(r_i,q_j)\end{cases} \quad (3)$$

On 14nm CPU (x86)	On 40nm DSA
35 ALU ops, 15 load/store	
37 cycles	**1 cycle** (37x speedup)
81 nJ (mostly for fetching, decoding, and reordering instructions)	**3.1 pJ** (26,000x efficiency) (0.3 pJ for computing logic, remainder for memory)

Turakhia, Y., Bejerano, G. and Dally, W.J., Darwin: A Genomics Co-processor Provides up to 15,000 X Acceleration on Long Read Assembly. *ASPLOS, 2018*

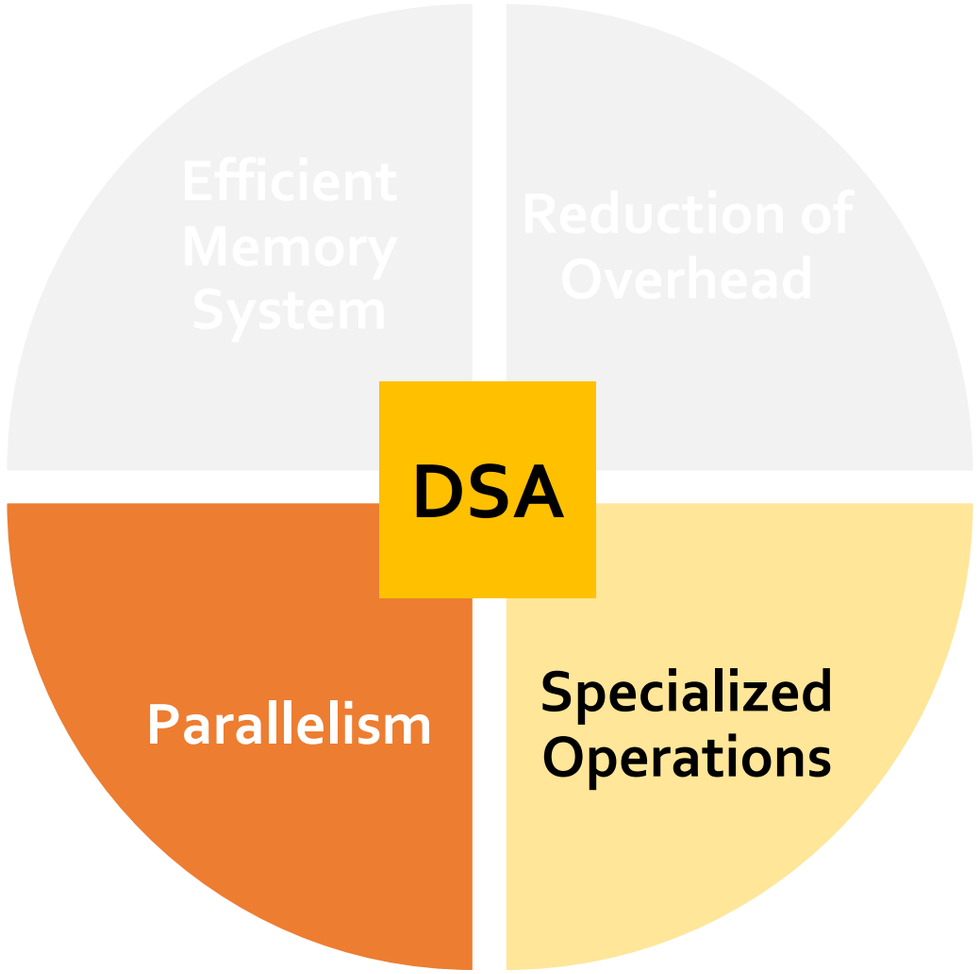
Specialized Operations and Data Types

From moderate speedup to magnificent speedup?

Specialization → Efficiency → Parallelization → Speedup

- Specialization → Efficiency
  - 37x speedup, 26,000x efficiency, 270,000x for logic
- Efficiency → Parallelization
  - Parallelism 64 PE arrays x 64 PEs per array, 4,096x total
- Parallelization → Speedup
  - 37 (Specialization) x 4,034 (Parallelism) = 150,000x total

Principles of DSAs



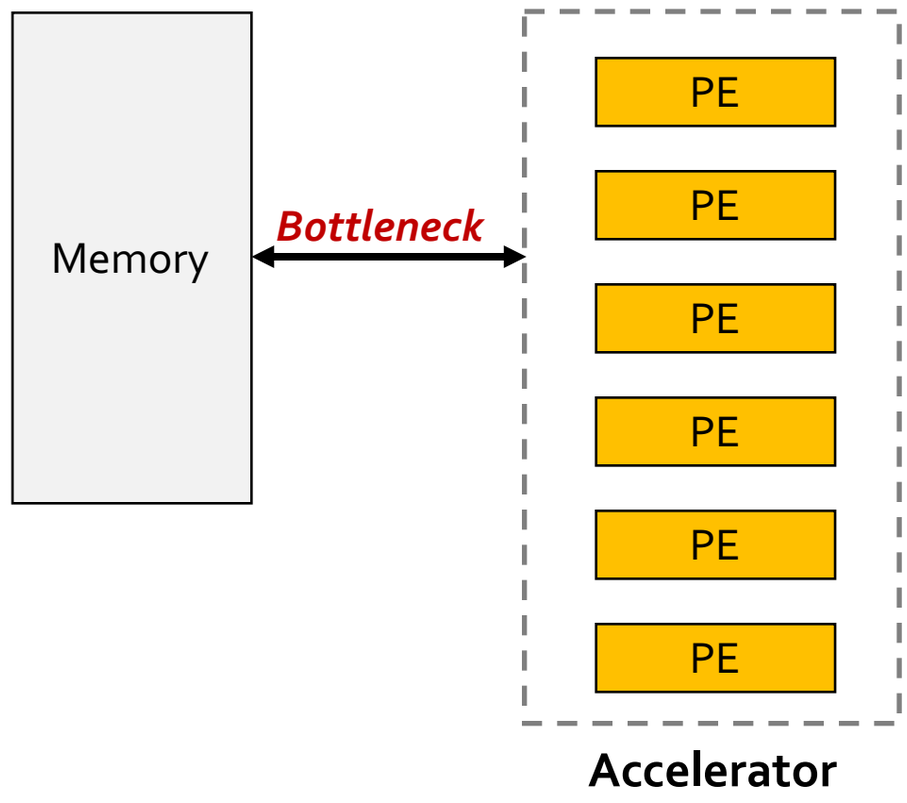
Parallelism

High degrees of parallelism provide gains in performance

- Many levels of parallelism
  - Multiply-accumulate (MAC) level
  - Processing element (PE) level (in some literature PE == MAC)
  - Function level
  - …

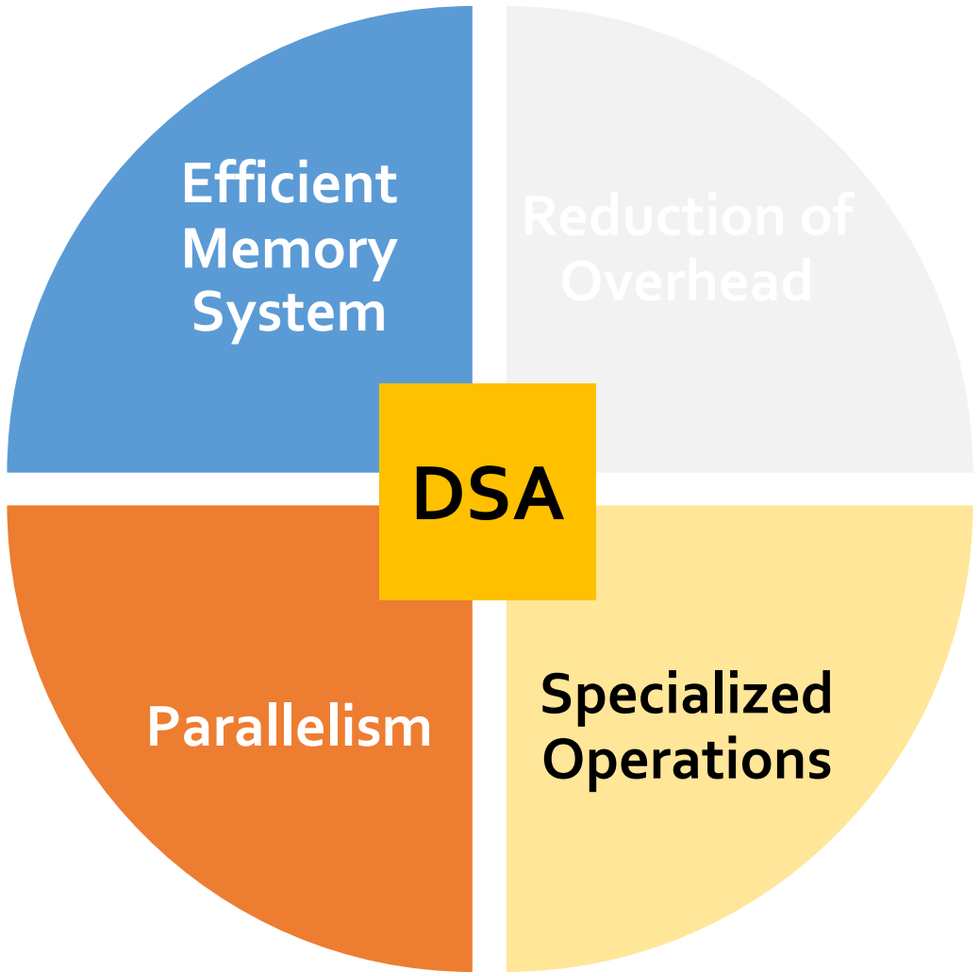
Parallel units must exploit locality

- Make very few global memory references; otherwise, performance will be memory-bound
- Make very few cross-PE data movement (ideally none)



Bottleneck: In many operations, the code needs to wait for all PE operations to be performed before it can continue.

Principles of DSAs

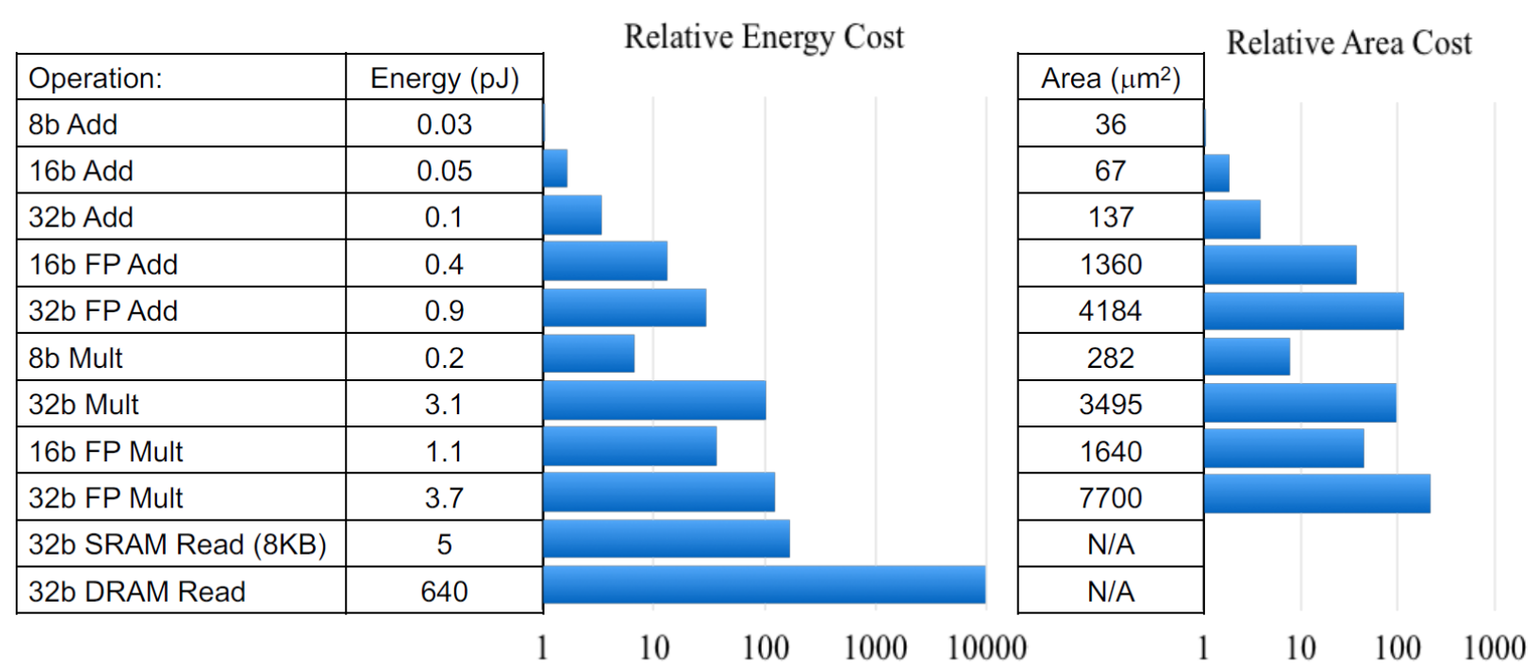


Local and Optimized Memory

Quote from William Dally: Accelerator Design is Guided by Cost

- Arithmetic is free (particularly low-precision)
- Memory is expensive
- Communication is prohibitively expensive

Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

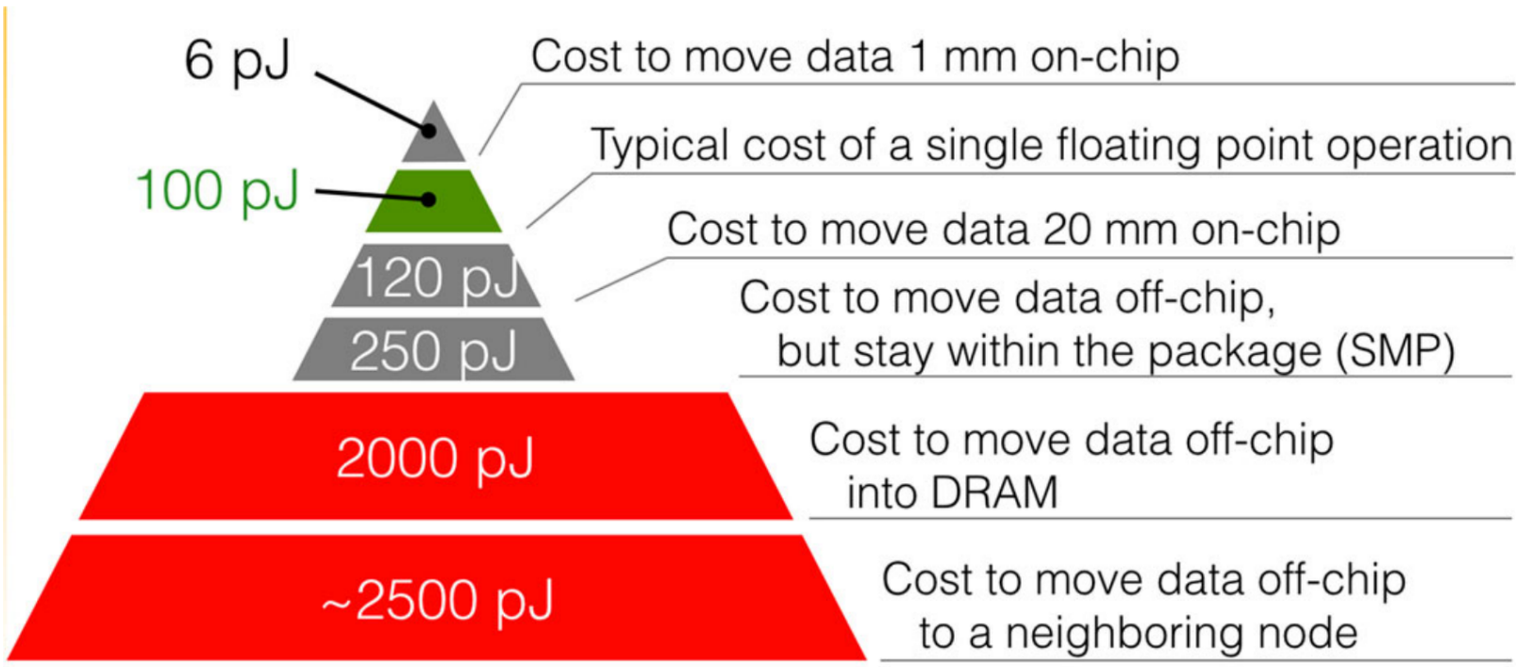


Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Local and Optimized Memory

Quote from William Dally: Accelerator Design is Guided by Cost

- Arithmetic is free (particularly low-precision)
- Memory is expensive
- Communication is prohibitively expensive



You, Y., et al. Fast LSTM by dynamic decomposition on cloud and distributed systems. *Knowledge and Information Systems*, 2020

Local and Optimized Memory

Storing key data structures in many small, local memories and try to reuse them

- Multiple levels of data buffers
- Very high memory bandwidth
- Low cost and energy

Data compression

- Increase the effective size of a local memory
- Increase the effective bandwidth of a memory interface
- “Quantization” – usually requires algorithm support (will discuss in a later lecture)

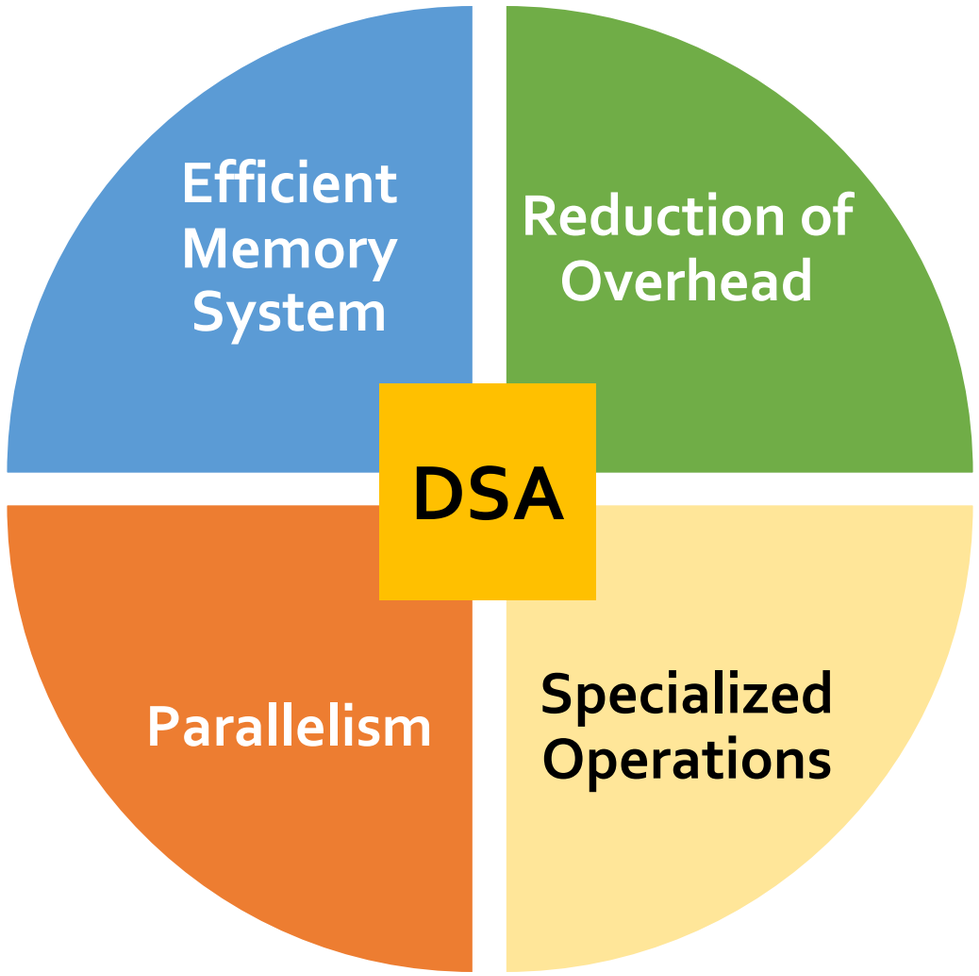
Compressed storage

- E.g., sparse matrix

- Data reuse
- Once stored in local memory from global (or lower-level) memory, use it as many times as possible

- …

Principles of DSAs



Reduced Overhead

Specializing hardware reduces the overhead of program interpretation

A simple in-order processor spends over 90% of its energy on overhead

- Instruction fetch, instruction decode, data supply, and control

A modern out-of-order processor spends over 99.9% of its energy on overhead

- Adding costs for branch prediction, speculation, register renaming, and instruction scheduling
- Example
  - 32-bit integer add @ 28 nm CMOS → 68 fJ
  - Integer add on 28 nm ARM A-15 → 250 pJ
    - 4000× the energy of the add itself!

[Dally, *et al.* "Efficient embedded computing", Computer 2008]

[Vasilakis, E. "An instruction level energy characterization of ARM processors", Tech. Rep. FORTHICS/TR-450, 2015]

Reduced Overhead

Overhead reduction in DSAs

- Most adds do not need full 32-bit precision
- No instructions to be fetched → no instructions fetch and decode energy
- No speculation → no work lost due to mis-speculation
- Most data is supplied directly from dedicated registers → no energy is required to read from a cache or from a large, multi-ported register file

When logic is “free”, memory and communication dominates!

	Unit	Area (mm²)	(%)	Power (W)	(%)
GACT	Logic	17.6	20.5	1.04	23.6
	Memory	68.0	79.5	3.36	76.4
D-SOFT	Logic	6.2	1.8	0.41	4.4
	Memory	320.3	98.2	8.80	95.6
EIE	Logic	2.8	6.9	0.23	40.3
	Memory	38.0	93.1	0.34	59.7

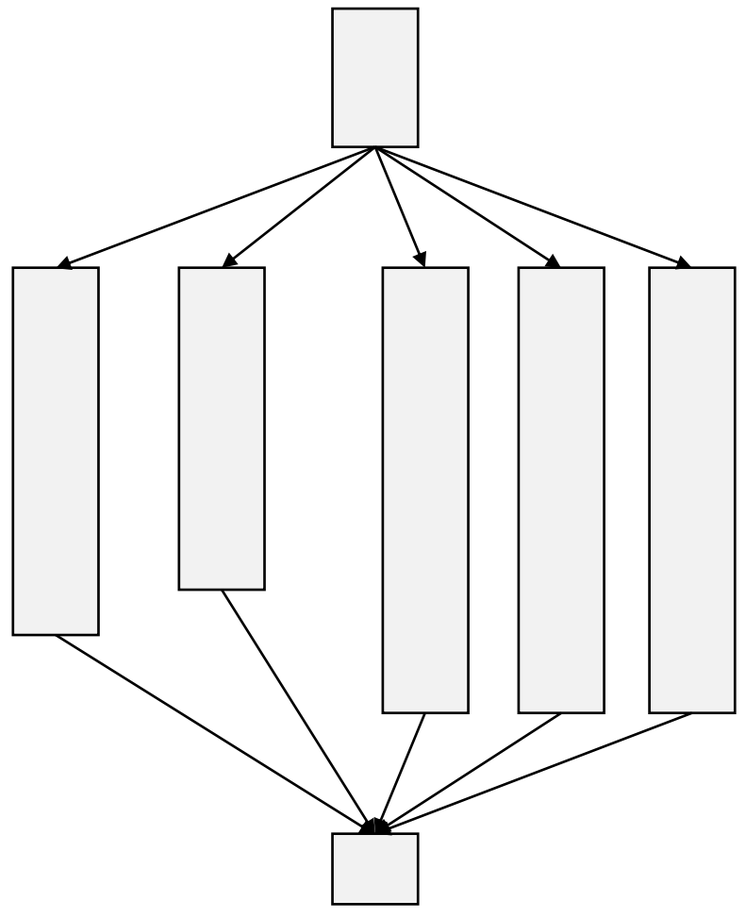
TSMC 40 nm technology

[Turakhia, Y. et al. , “Darwin: A genomics co-processor provides up to 15,000× acceleration on long read assembly”. ASPLOS 2018]

Other Principles

Need to Accelerate the Whole Problem

- Amdahl's Law!
- Watch for critical path and the components that are "hidden" by others



Simple parallelism often beats a "better" algorithm

- Algorithm-accelerator co-design is needed
- Is the algorithm hardware-friendly? Parallelism-friendly? Memory-friendly?

•Algorithms must be memory-optimized

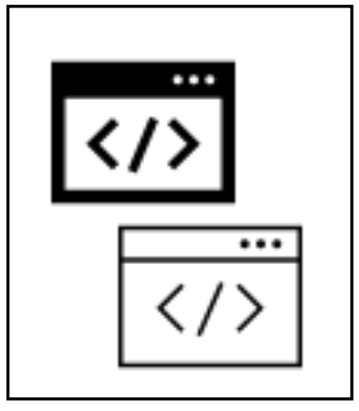
- Minimize global memory accesses
- Keep local memory footprint small

•

A Bigger Picture of HLS, DSA, and Compiler

High Level Synthesis (HLS) is essentially a compiler (automation tool) to design new domain-specific accelerators (DSAs)

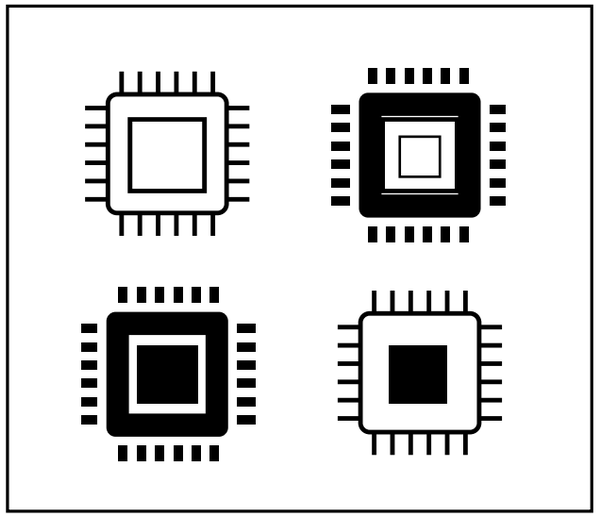
Strictly, DSA is different from ASIC since it still requires software



High-level programming languages

Compiler: to map programs to fixed DSAs

HLS: to design some new DSAs



Domani Specific Architectures (DSAs)

Summary

General-purpose computing is inefficient

Domain-Specific Accelerators (DSAs) are the future

- Specialization → efficiency
- Parallelism → speedup
- Co-design → is the algorithm DSA-friendly?

Memory and communication dominate

- Minimize global memory access
- Minimize memory footprint – new algorithms, sparsity, compression
- Lots of small, fast on-chip memories

FPGAs as accelerator platforms

- Highly flexible → highly specialized
- Highly parallelizable
- High local memory bandwidth

HLS as an easy-to-use compilation tool

Logic Gates

Logic Gates

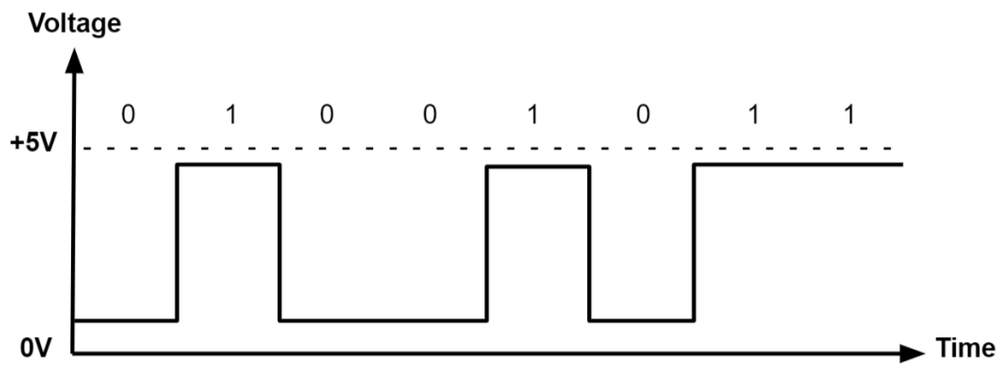
Digital logic is the building block of digital systems

- FPGAs rely on digital logic
- Computer hardware relies on digital logic
  - Every calculation, operation, pixel render relies on digital logic!!
- A circuit can be designed by drawing logic gates that are then mapped onto general-purpose gates on the FPGA
  - Also can be connected using Verilog or other HDL

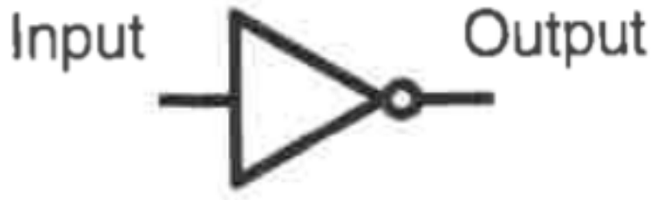
Logic gates have inputs and outputs

- Inputs are high or low
  - low is close to 0V
  - high is over half the supply voltage
    - typically 1.8, 3.3 or 5 V

Simplest Logic gate “NOT” shown to the right



NOT gate



NOT gate truth table

Input	Output
L	H
H	L

AND and OR

"AND" gates

- If either input is low then the output is low
  - similar to C++ && operation

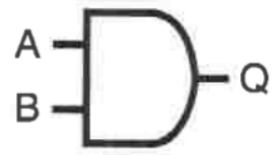


Figure 1-3 *An AND gate.*

Input A	Input B	Output Q
L	L	L
L	H	L
H	L	L
H	H	H

Table 1-2 *Truth Table for an AND Gate*

"OR" gates

- If either input is high then the output is High

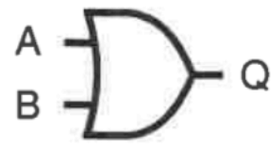


Figure 1-4 *An OR gate.*

Input A	Input B	Output Q
L	L	L
L	H	H
H	L	H
H	H	H

Table 1-3 *Truth Table for an OR Gate*

NAND and NOR

"NAND" gates

- AND gate with inverted output

"NOR" gates

- OR gate with inverted output

"XOR" gates

- One or the other but NOT both

Figure 1-5 *NAND and NOR gates.*

Input A	Input B	Output Q
L	L	H
L	H	H
H	L	H
H	H	L

Table 1-4 *Truth Table for a NAND Gate*

Input A	Input B	Output Q
L	L	H
L	H	L
H	L	L
H	H	L

Table 1-5 *Truth Table for a NOR Gate*

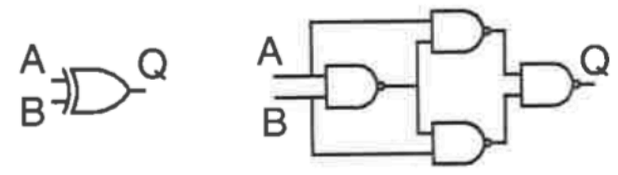


Figure 1-7 *Making an XOR gate from four NAND gates.*

Input A	Input B	Output Q
L	L	L
L	H	H
H	L	H
H	H	L

Table 1-7 *Truth Table for an XOR Gate*

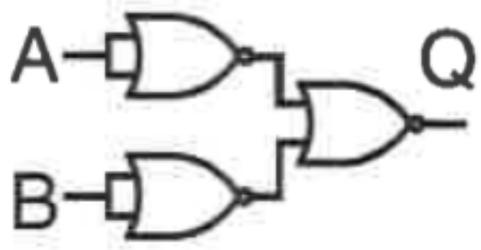
Making an AND gate with NOR gates

Input A	Input B	Output Q
L	L	L
L	H	L
H	L	L
H	H	H

Table 1-2 *Truth Table for an AND Gate*

Input A	Input B	NOT A	NOT B	Output Q
L	L	H	H	L
L	H	H	L	L
H	L	L	H	L
H	H	L	L	H

Table 1-6 *Truth Table for Making an AND Gate from NOR Gates*



Adding with Logic

First remember the binary representation of decimal numbers

Decimal Number	Binary Number
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table 1-8 *The Numbers 0 to 7 in Binary and Decimal*

If we want to add 2 bits then we need a third bit to represent the maximum value

- 001+001 = 010

Input A	Input B	Output Sum (A + B)	Output Carry-Out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1-9 *Truth Table for a Single-Bit Adder*

Adding with Logic

If we want to add 2 bits then we need a third bit to represent the maximum value

- 001+001 = 010

Input A	Input B	Output Sum (A + B)	Output Carry-Out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1-9 *Truth Table for a Single-Bit Adder*

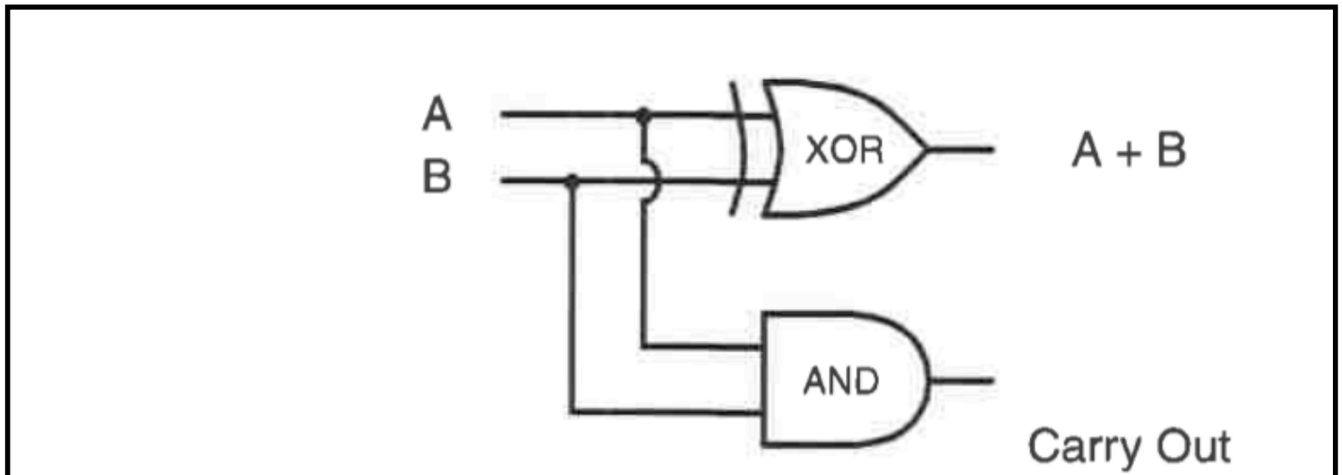


Figure 1-8 *A single-bit adder.*

That's all for today!

On your own:

Read about Flip Flops, Shift Registers and Counters

Complete homework problems