Lecture 1:

Introduction

Sitao Huang

sitaoh@uci.edu

January 4, 2022

Instructor

Sitao Huang

Assistant Professor, EECS

Email: sitaoh@uci.edu
Website: sitaohuang.com

- Research Interests:
  - *Hardware accelerators*
  - *Programming languages and compilers for accelerators*
  - *High-level synthesis (HLS)*
  - *Heterogeneous computing*

- *Ph.D., University of Illinois at Urbana-Champaign (2021)*
- *M.S., University of Illinois at Urbana-Champaign (2017)*
- *B.S., Tsinghua University (2014)*

Recruiting students!

Languages and Compilers for Hardware Accelerators

- Topics
  - Hardware accelerators (FPGAs, GPUs, ASICs, etc.)
    - Types, design methodology, performance metrics, etc.
  - Programming languages for accelerators
    - Abstraction levels, language features, programming paradigms, etc.
  - Compilation flows for accelerators
    - Compiler construction, optimizations, hardware-specific considerations, etc.
  - Recent works in the field

- Goals
  - Get a better understanding on the hardware accelerator design flows
  - Know about various ways of designing languages and compilers for accelerators
  - Compiler optimization techniques for hardware accelerators
  - Get to know state-of-the-art research works in this field

Announcements, Time, and Location

- **Announcements**: Check ***Canvas*** and emails for the latest announcements
- **Lecture Time**:
  - **Tuesdays and Thursdays 8:00 – 9:20 am**

- **Lecture Location**:
  - First two weeks (1/3 – 1/14) or until further notice: *online*, Zoom link
  - Later weeks if in-person: SSTR 101

- **Office Hours**:
  - **Tuesdays 9:30 – 10:30 am** or by appointment (sitaoh@uci.edu)
  - Location: Lecture Zoom link, if lectures are online
    Engineering Hall 3215, if lectures are in-person

Grading Policy

- Homework – 30%
  - Four assignments, each 7.5%
- Midterm – 30%
  - February 10 (Thursday, Week 6), in class
- Course Project – 40%
  - Proposal (due Jan. 31), 10%
  - Final Presentation (Week 10), 15%
  - Final Report (Week 10), 15%

NOTE: Please check Canvas for the latest announcements

Tentative Schedule

- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compiler Optimizations for Accelerators
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

Languages and Compilers for Hardware Accelerators

- ***What?***
- ***Why?***
- ***How?***

Languages and Compilers for Hardware Accelerators

- ***What?***
- ***Why?***
- ***How?***

Languages and *Compilers* for *Hardware Accelerators*

Languages and Compilers for *Hardware Accelerators*

- **Hardware accelerator**: “computer hardware designed to perform *specific functions more efficiently* compared to software running on a CPU” (Wikipedia)
- Tradeoff between flexibility and efficiency
- Invest time and money for better performance and efficiency

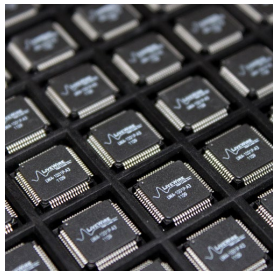
More flexible
More general

More specialized
More efficient

CPU

GPU

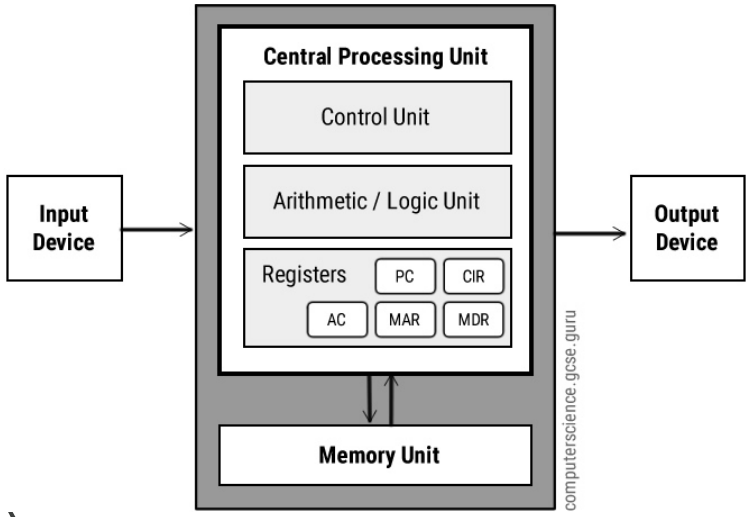
FPGA
(Field-Programmable Gate Array)



ASIC
(Application-Specific Integrated Circuit)

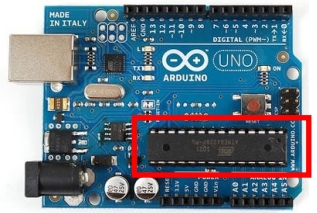
General Purpose Processor

- Programmable processors used in a variety of applications
- Not designed for any specialized purposes
- Central Processing Unit (CPU)
- Architecture (Von Neuman)
  - Arithmetic Logic Unit (ALU)
  - Control Unit
  - Registers
  - Memory management unit (MMU)
  - Cache
- Implemented on Integrated Circuit (IC)
  - one or more CPU cores in a single IC chip
- An IC that contains a CPU may also contain memory, peripheral interfaces
  - Microcontrollers and System-on-Chip (SoC)

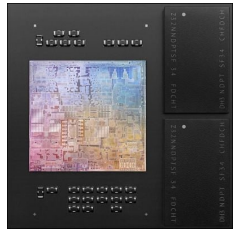


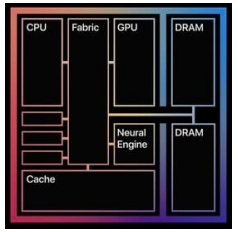
Intel i9-12900K CPU

Qualcomm Snapdragon SoC



Microcontroller Unit (MCU) on an Arduino board





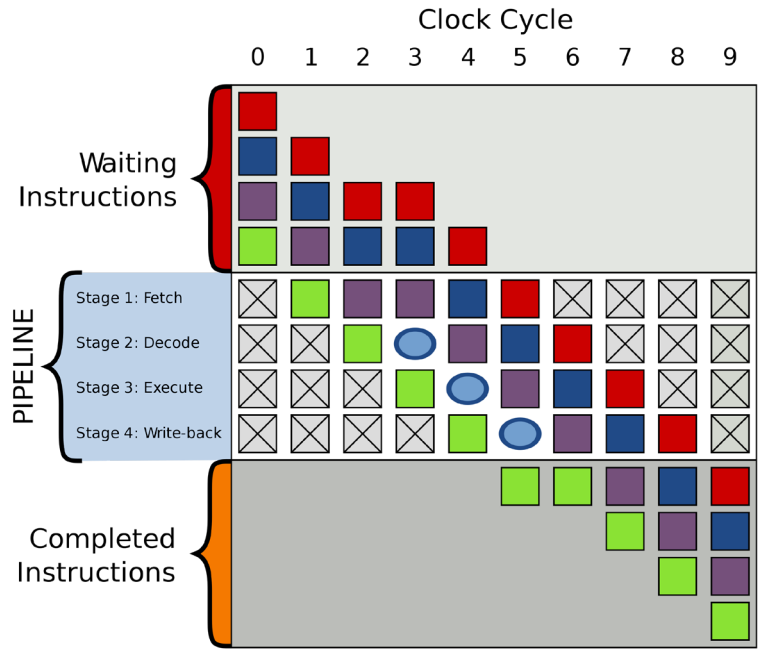
Apple M1 SoC

General Purpose Processor

- Instruction Pipelining
  - A technique for implementing instruction-level parallelism within a processor
  - Pipeline stages (five-stage pipeline case)
    - Instruction Fetch (IF)
    - Instruction Decode (ID)
    - Execute (EX)
    - Memory access (MEM)
    - Register write back (WB)
- Hazards
  - *Data hazard*: values produced from one instruction are not available when needed by a subsequent instruction
  - *Control hazard*: a branch in the control flow makes ambiguous what is the next instruction to fetch
- Pipeline Stall
  - Delay in execution of an instruction in order to resolve a hazard

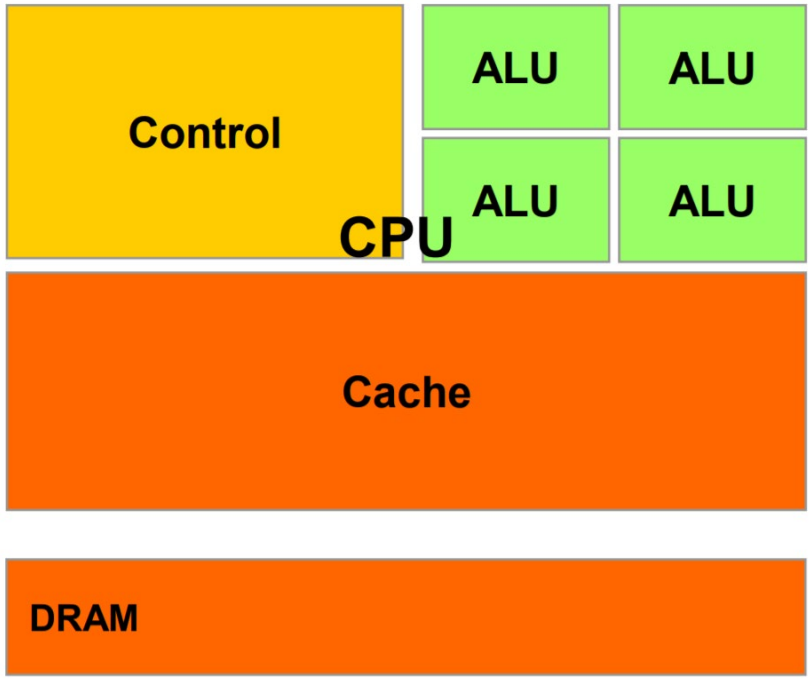
Basic five-stage pipeline

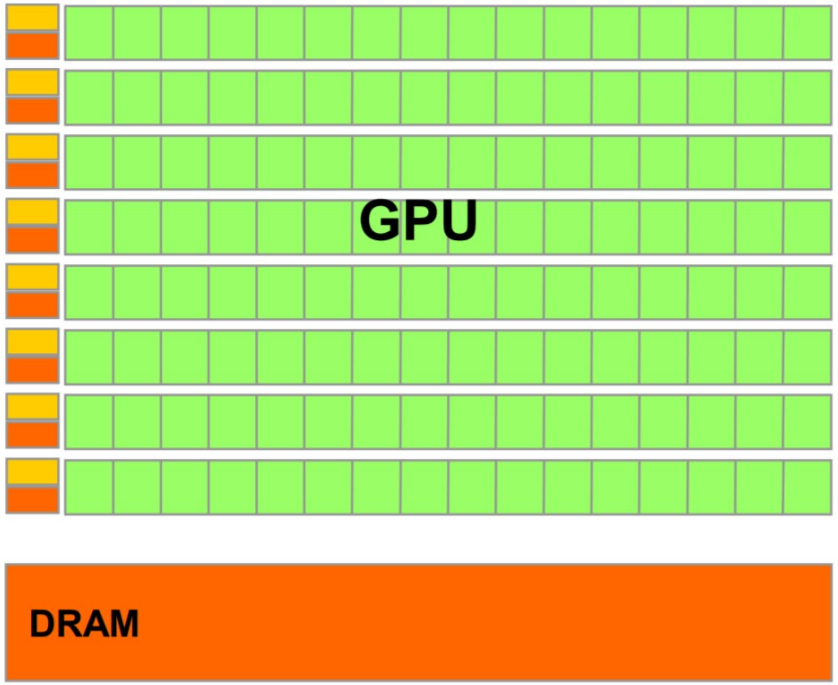
Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX



Graphics Processing Unit (GPU)

- CPUs and GPUs have fundamentally different design philosophies
  - CPUs: ***low***-latency, ***low***-throughput
    - high clock freq., large caches, sophisticated control, powerful ALUs
  - GPUs: ***high***-latency, ***high***-throughput
    - moderate clock freq., small caches, simple control, (many) energy efficient ALUs
    - *Require **massive** number of threads to tolerate latencies* ⇨ ***More specialized in specific applications***

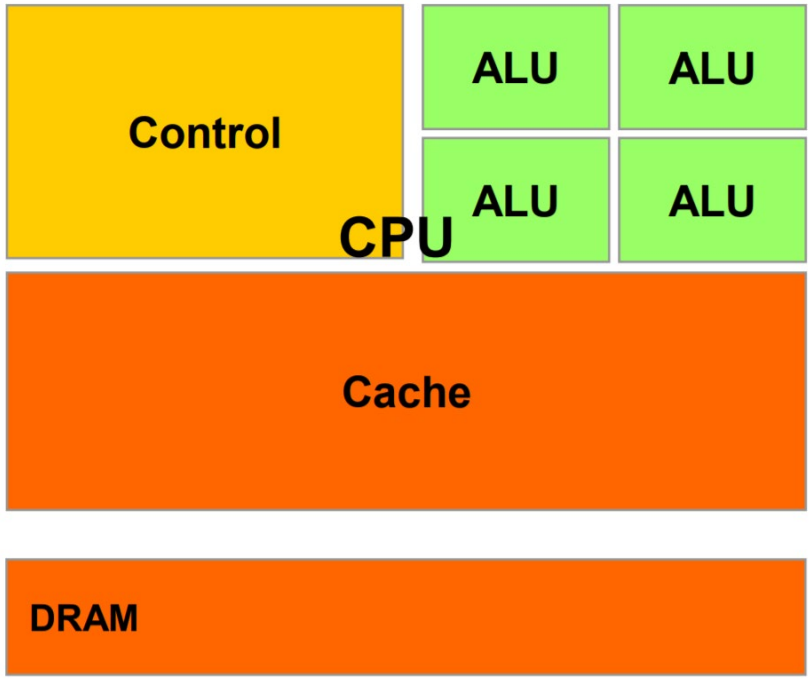


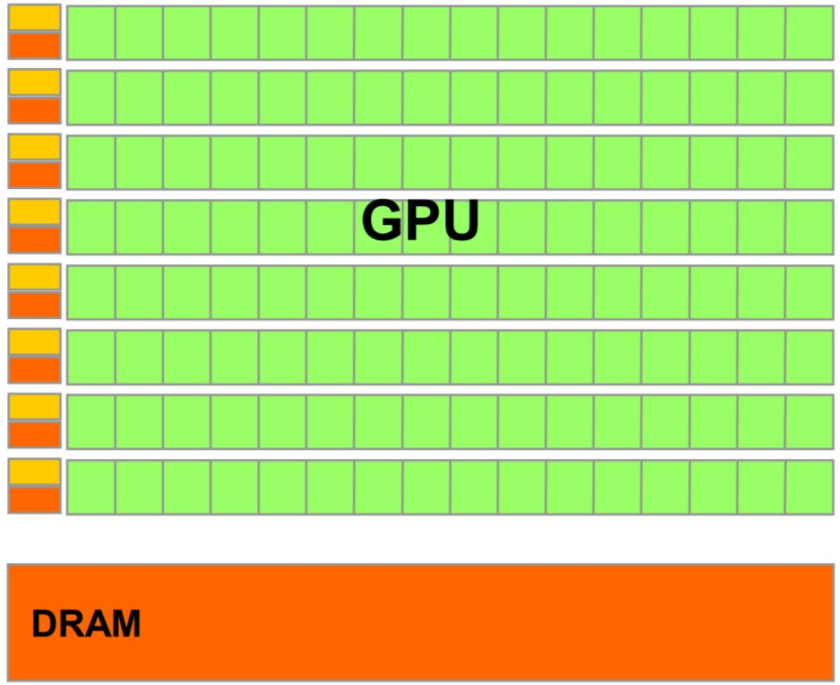


Source: Prof. Wen-mei Hwu, UIUC ECE 408: "Applied Parallel Programming"

Graphics Processing Unit (GPU)

- CPUs and GPUs have fundamentally different design philosophies
  - CPUs: Good for sequential parts where latency matters
    - CPUs can be > 10x faster than GPUs for sequential code
  - GPUs: Good for parallel parts where throughput wins
    - GPUs can be > 10x faster than CPUs for parallel code

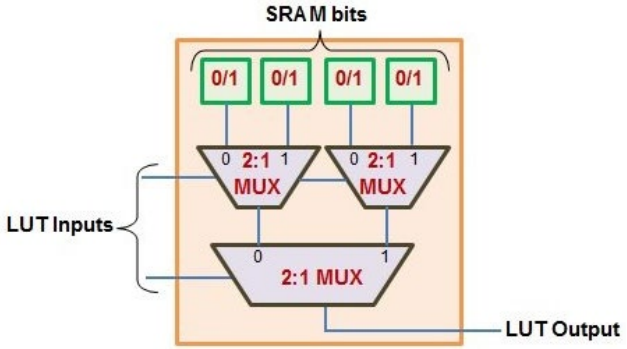


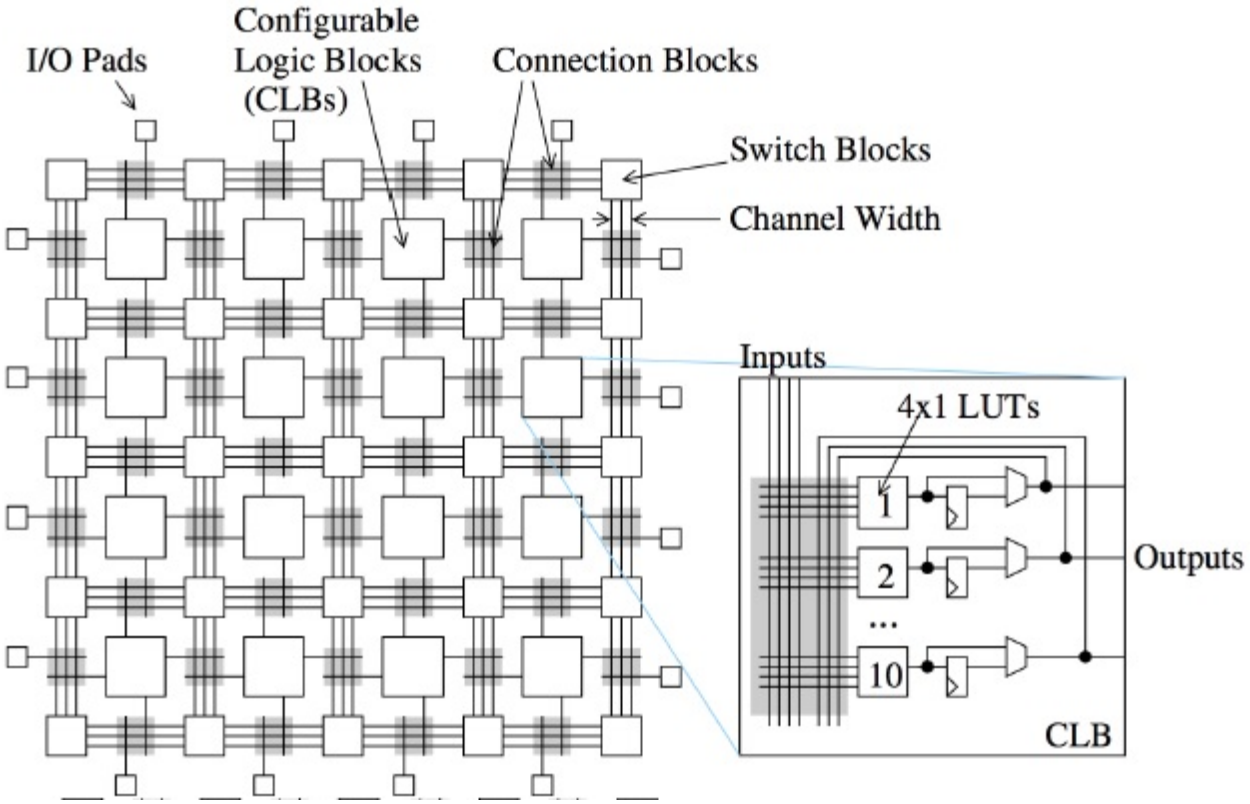


Source: Prof. Wen-mei Hwu, UIUC ECE 408: "Applied Parallel Programming"

Field-Programmable Gate Array (FPGA)

- High-density programmable gate array
- Fully programmable through bit-stream configuration file
  - Programmable Logic
  - Programmable I/O
  - Programmable Interconnects (routing)
- Look-up table (LUT) based combinational logic
  - Can be implemented with SRAM (volatile)

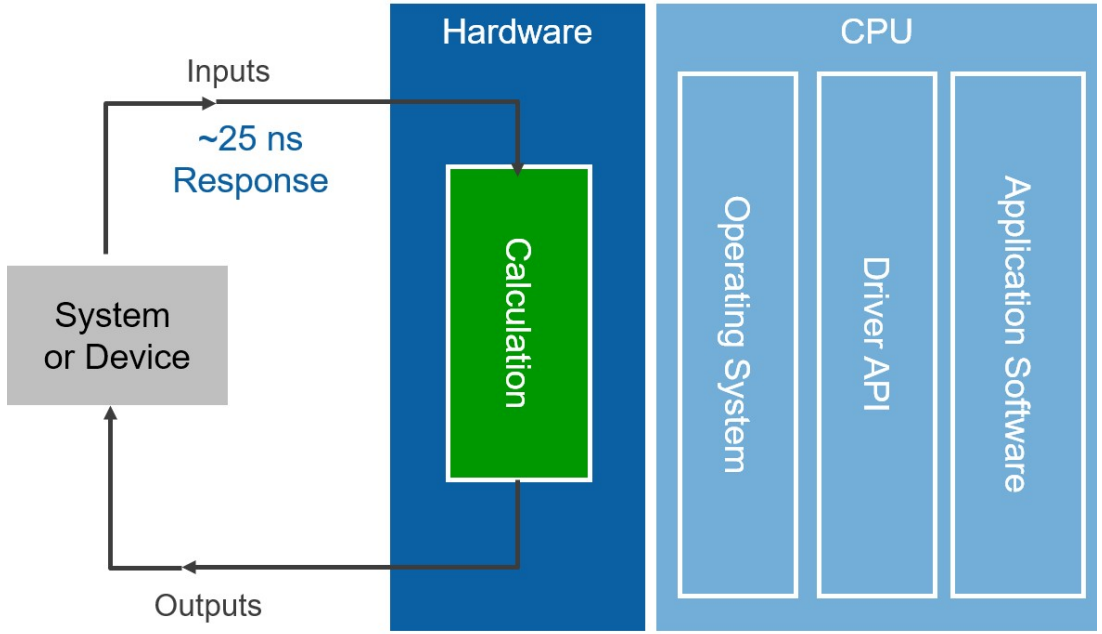




Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

Field-Programmable Gate Array (FPGA)

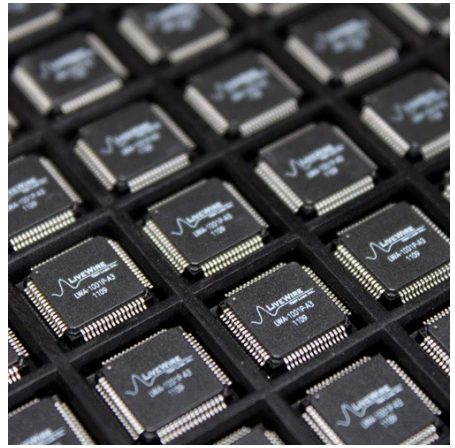
- Massive fine-grained parallelism
- Clock cycle accurate control & compute
- Very low latency, very short response time (benefits from specialization and customization)
  - Applications: real-time video processing, signal processing, high-frequency trading, etc.
- Lower clock frequency (typically < 1GHz) compared to CPU/GPU/ASIC

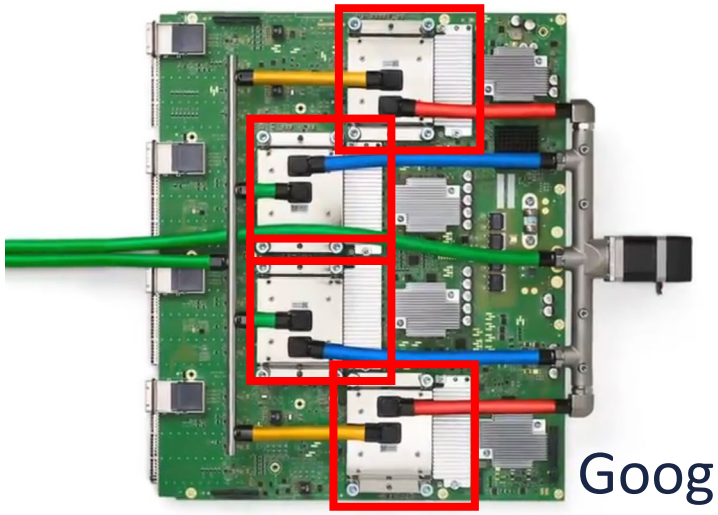


Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

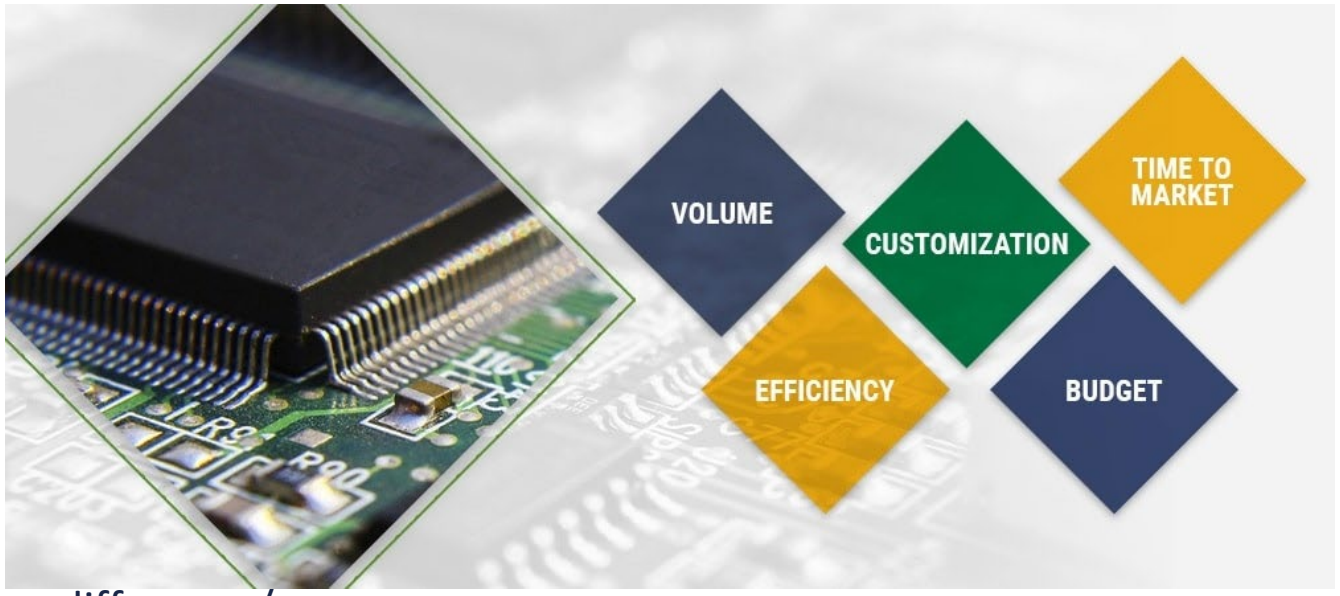
Application-Specific Integrated Circuit (ASIC)

- An IC chip customized for a particular uses
- Cannot be reprogrammed for multiple applications
- Notably more efficiently than FPGAs
- Require a higher initial cost and longer design time compared to FPGA, more cost-effective if produced in large quantities
- Use: permanent applications in electronic devices
- NOTE: ASICs may be controlled with instructions





Google TPU v4



Reference: https://www.asicnorth.com/blog/asic-vs-fpga-difference/

Languages and Compilers for Hardware Accelerators

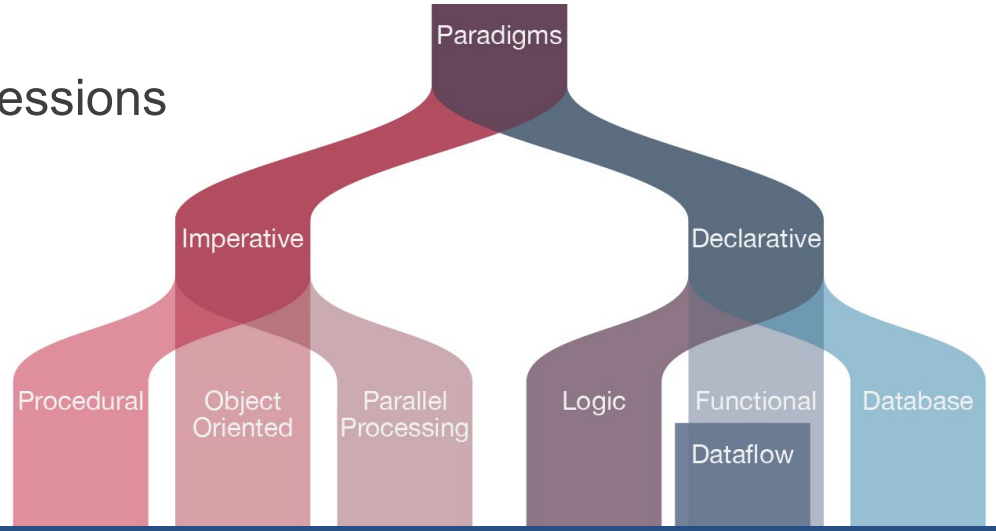
- **Programming languages**: formal language comprising a set of strings, used to implement algorithms
- Provide an abstraction for underlying hardware
- Provide a set of general operators to describe a range of applications
- Primitive elements of programming languages
  - Syntax: rules that define the correct combination of symbols

Lisp example: (from Wikipedia)

```
expression ::= atom | list
atom       ::= number | symbol
number     ::= [+-]?['0'-'9']+
symbol     ::= ['A'-'Z''a'-'z'].*
list       ::= '(' expression* ')'
```

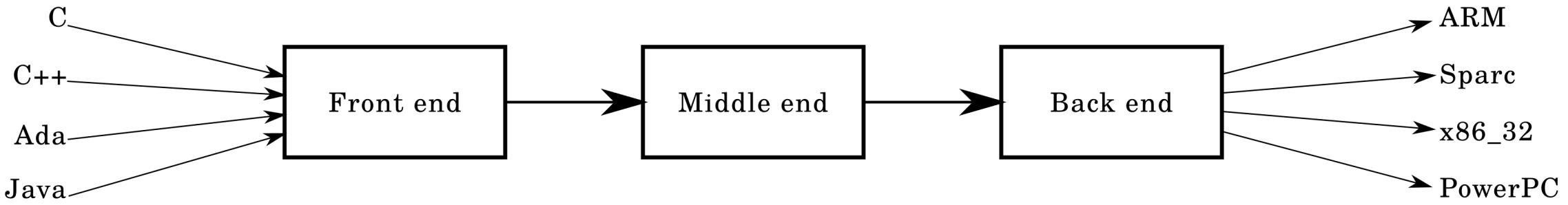
- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace);
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

  - Semantics: meaning of languages
  - Type system: how a language classify values and expressions
  - Standard library and run-time system
- Categories
  - Typed vs untyped languages, static vs dynamic typing
  - Functional programming vs imperative programming
  - ……



Languages and *Compilers* for Hardware Accelerators

- **Compilers**: a special program that processes code in a particular programming language and translates input code into the code in another or the same language.
- Categories
  - Just-In-Time (JIT) compiler, Ahead-of-Time (AOT) compiler
  - Source-to-source compiler
  - ……
- Three-stage compiler structure
  - Front end: translate source code to intermediate representation (IR), manages symbol table
  - Middle end: compiler analysis (e.g., data-flow analysis) and optimization (e.g., loop transformation)
  - Back end: architecture specific optimizations, code generation



Languages and *Compilers* for *Hardware Accelerators*

In this course, we will cover

- (I) languages and compilers that generate instructions or code that hardware accelerators can run,
  - Example: languages and compilers for an FPGA-based deep learning accelerator that takes specific instructions

Bits	63 – 60	59 – 56	55 – 32	31	30 – 24	23 – 17	16	15 – 10	9 – 4	3 – 0
input	Opcode = 0	Function	Destination Instruction ID	Fixed-Point/ Floating Point	Value Bitwidth	Fraction Bits/ Exponent Bits	1	# of Dimensions		
conv	Opcode = 1						Use the Next Word	Window Width	Window Height	Window Stride
pool	Opcode = 2							# of Neurons		
norm	Opcode = 3									
ip	Opcode = 4									
act	Opcode = 5									
fanout	Opcode = 6		# Destinations				0	Reserved		
output	Opcode = 7		0							

Instructions of DNNWeaver accelerator
Source: "DNNWeaver: From High-Level Deep Network Models to FPGA Acceleration"

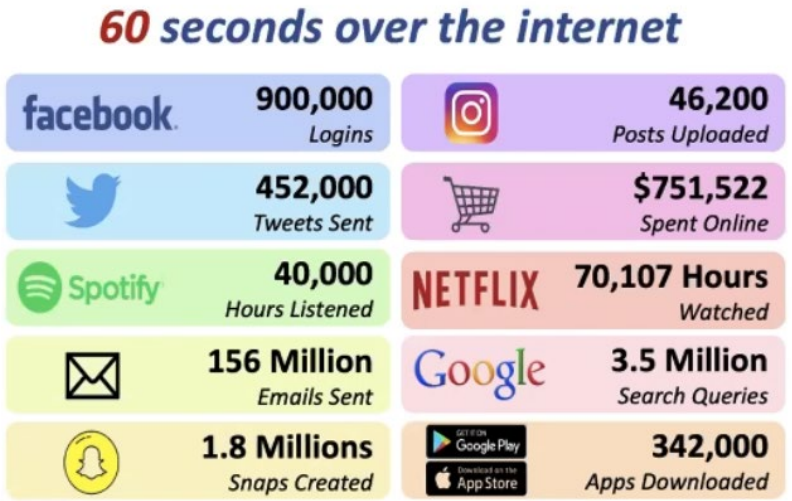
AND

- (II) languages and compilers that generate hardware accelerator designs
  - Example: languages and compilers that can generate an FPGA-based deep learning accelerator design
  - Multiple approaches: base on existing HLS flows (translate to C/C++), directly generates RTL from high-level language, OR, input language itself is a kind of HDL (hardware description language), etc.

Languages and Compilers for Hardware Accelerators

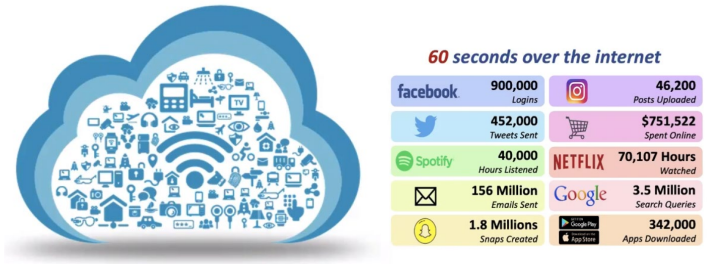
- *What?*
- ***Why?***
- *How?*

Challenges in Modern Computing

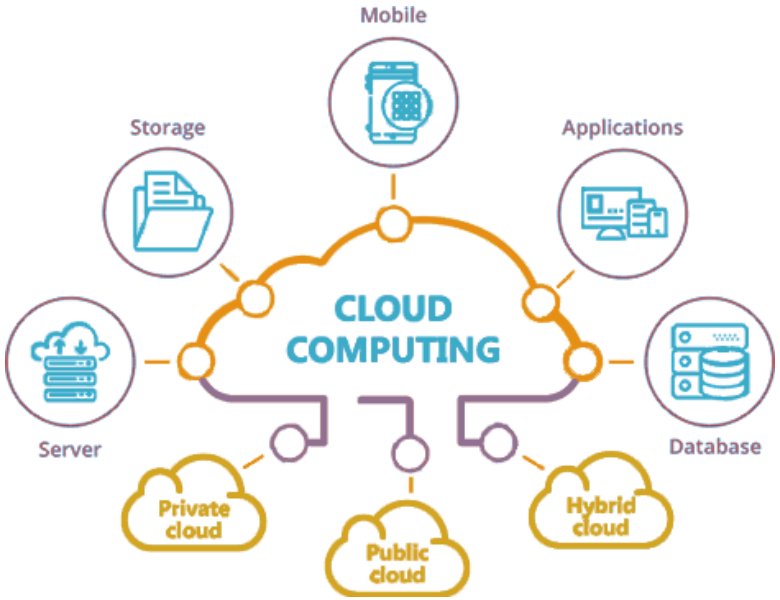


Complexity in Data

Challenges in Modern Computing



Complexity in Data

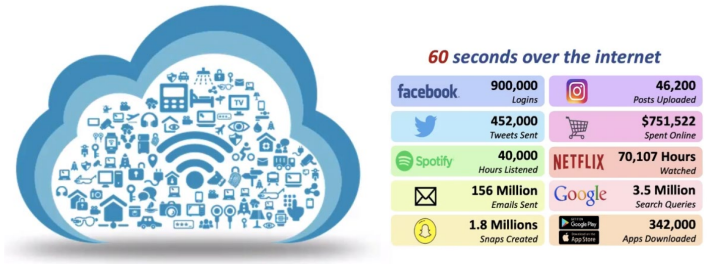


Cloud Computing

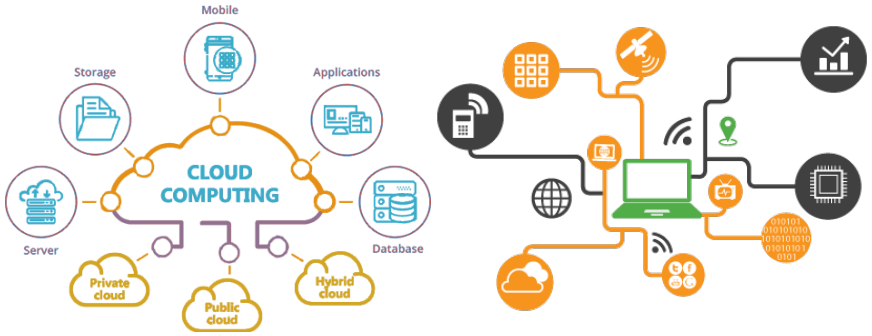
Edge Computing

Complexity in Applications

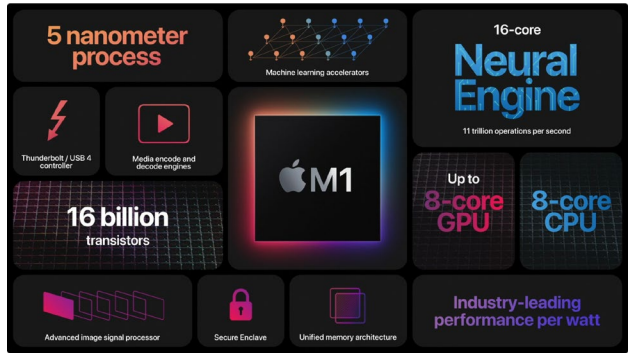
Challenges in Modern Computing



Complexity in Data

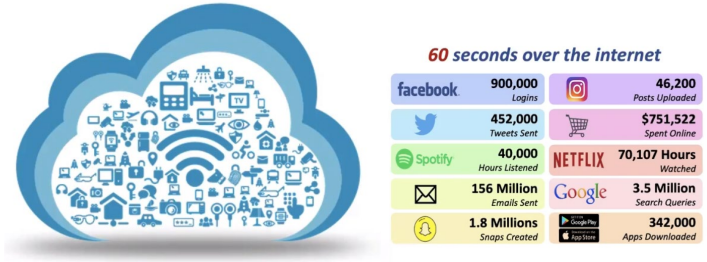


Complexity in Applications

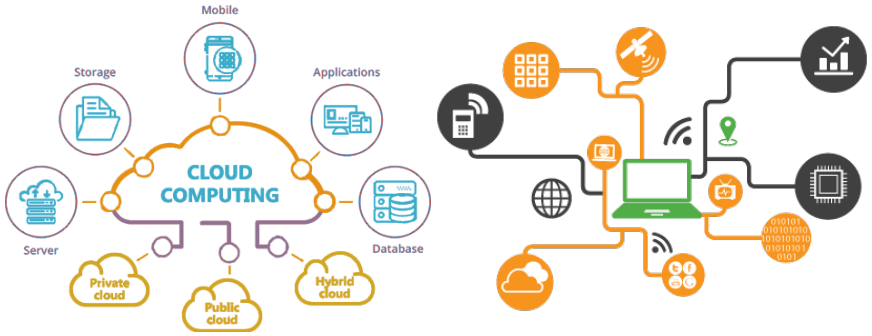


Complexity in Hardware

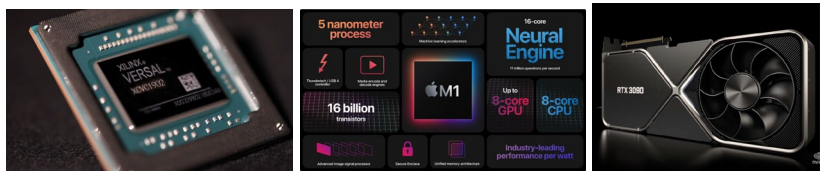
Challenges in Modern Computing



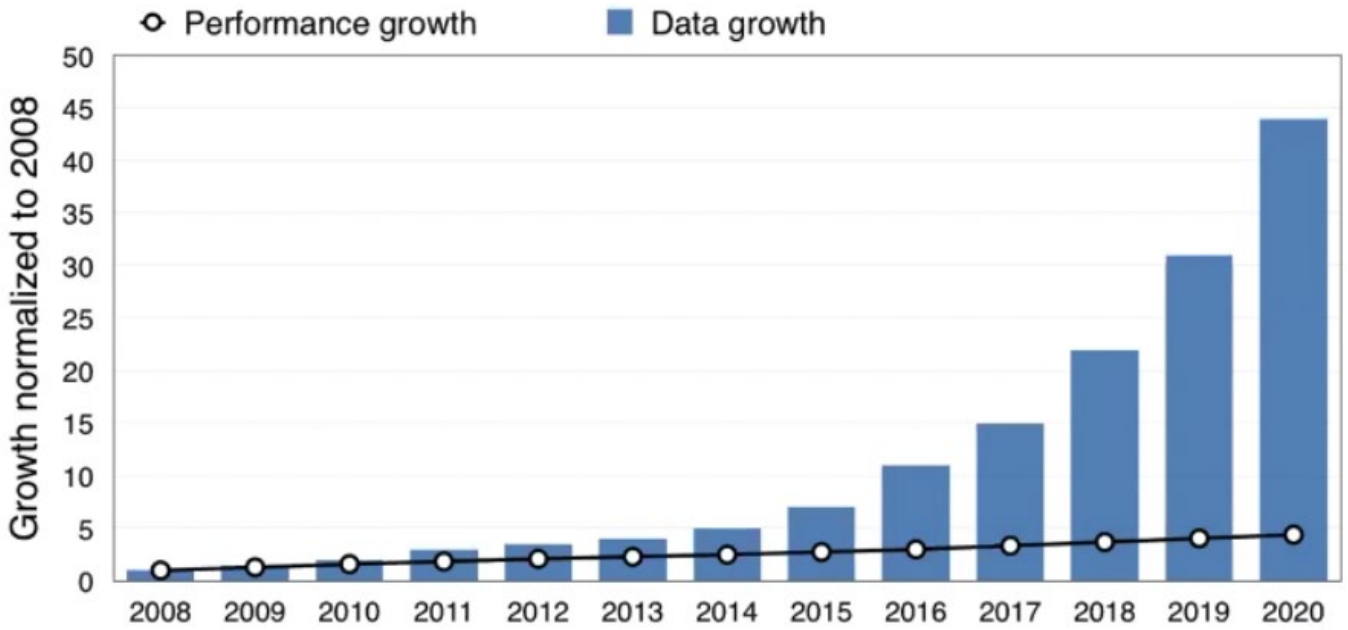
Complexity in Data



Complexity in Applications



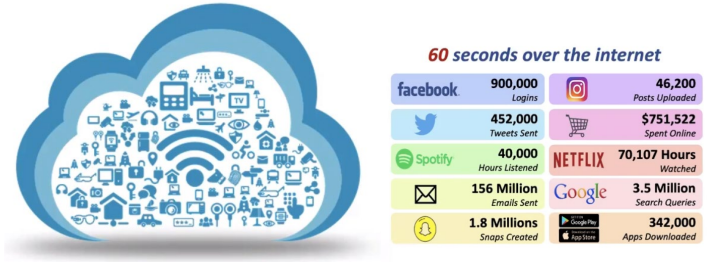
Complexity in Hardware



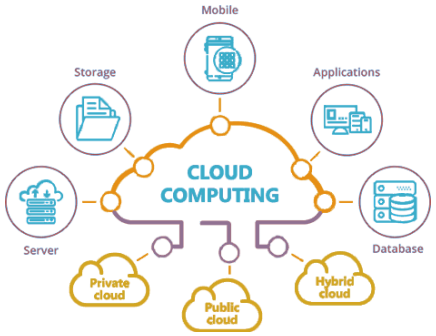
Data growth trends: IDC's Digital Universe Study, December 2012
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

Huge gap between data growth and processor performance growth!

Challenges in Modern Computing



Complexity in Data



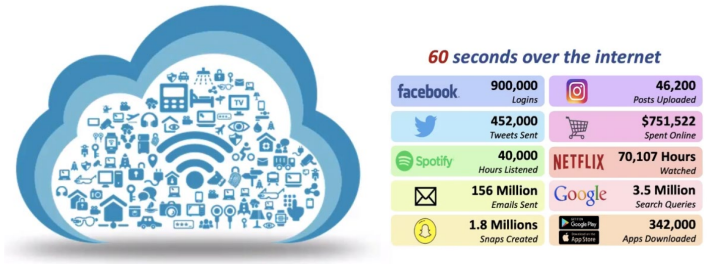
Complexity in Applications

Complexity in Hardware

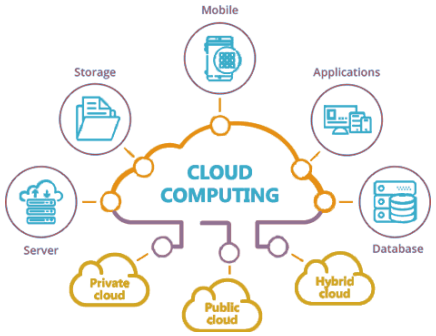
Accelerating modern applications with modern hardware is becoming more and more challenging!

Compilers automate the translation and optimization from application to hardware instructions

Challenges in Modern Computing

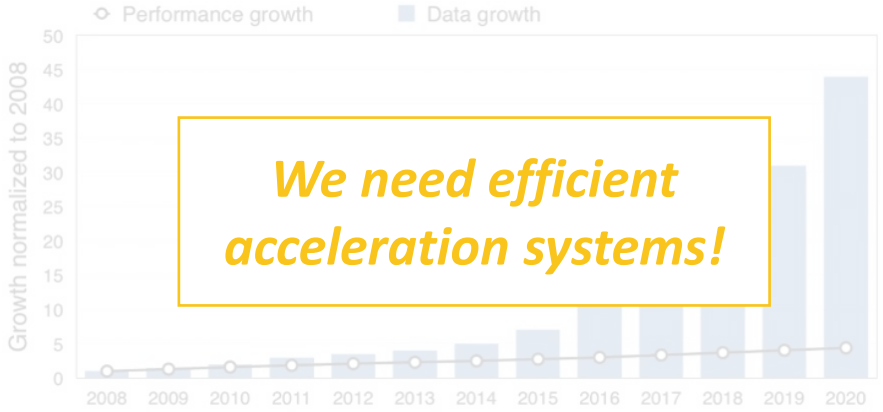


Complexity in Data



Complexity in Applications

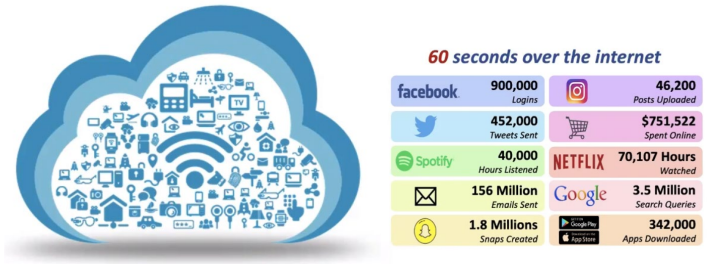
Complexity in Hardware



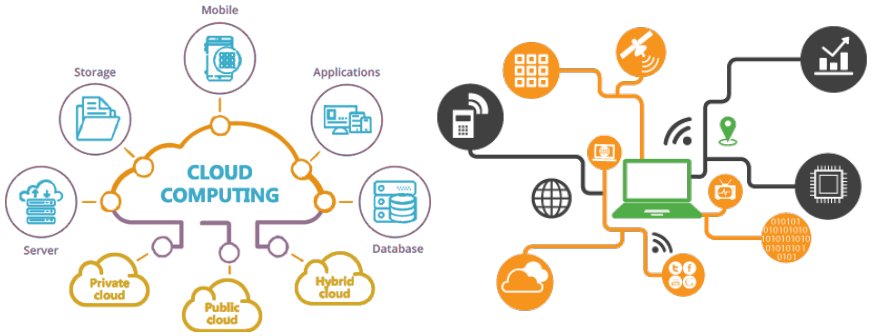
We need efficient acceleration systems!

Data growth trends: IDC's Digital Universe Study, December 2012
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

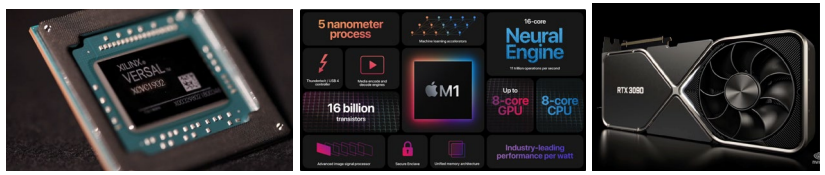
Challenges in Modern Computing



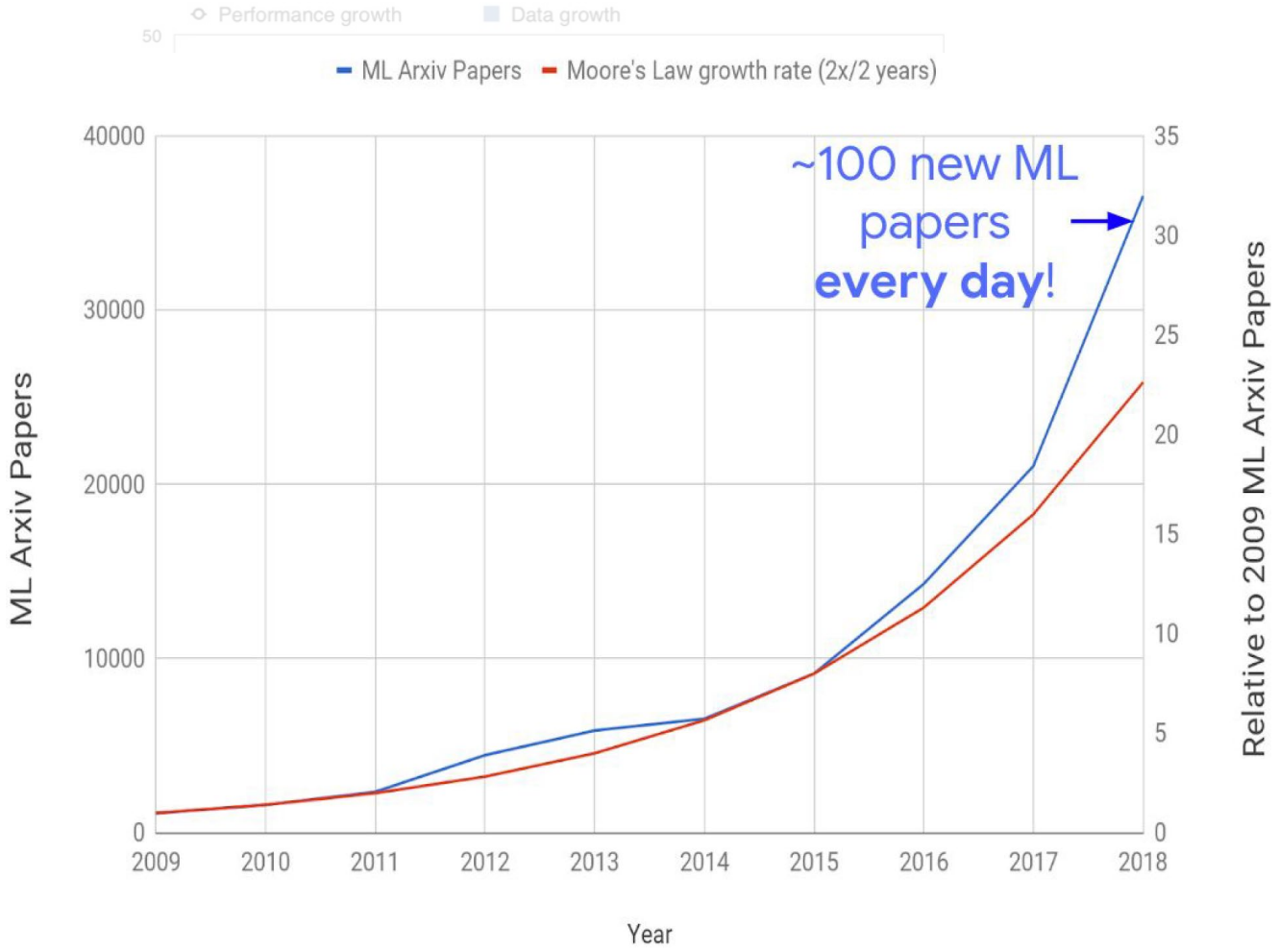
Complexity in Data



Complexity in Applications

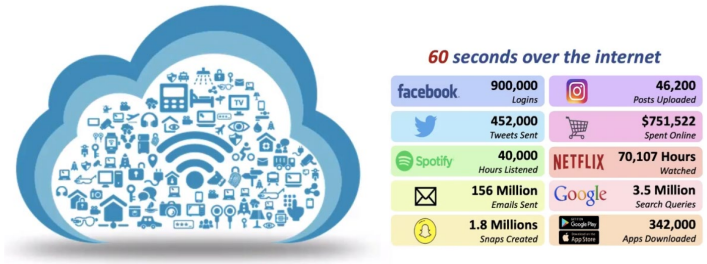


Complexity in Hardware

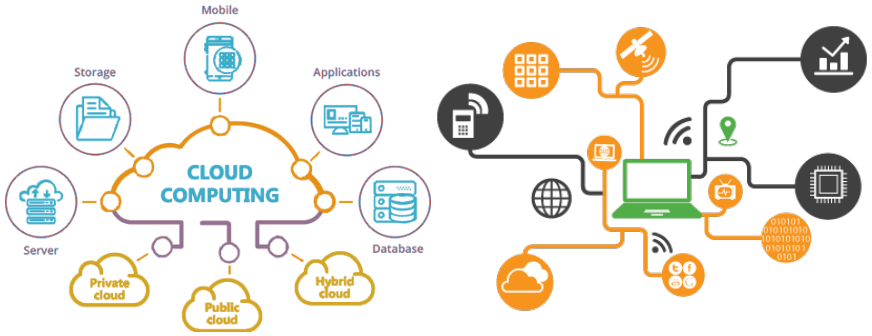


Jeff Dean. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design, arXiv 1911.05289.

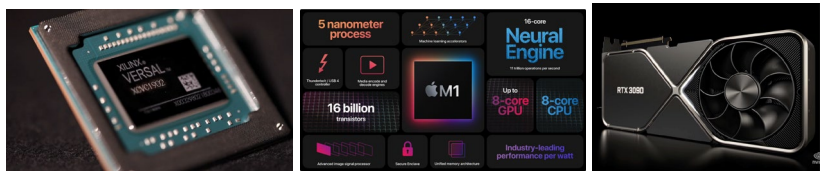
Challenges in Modern Computing



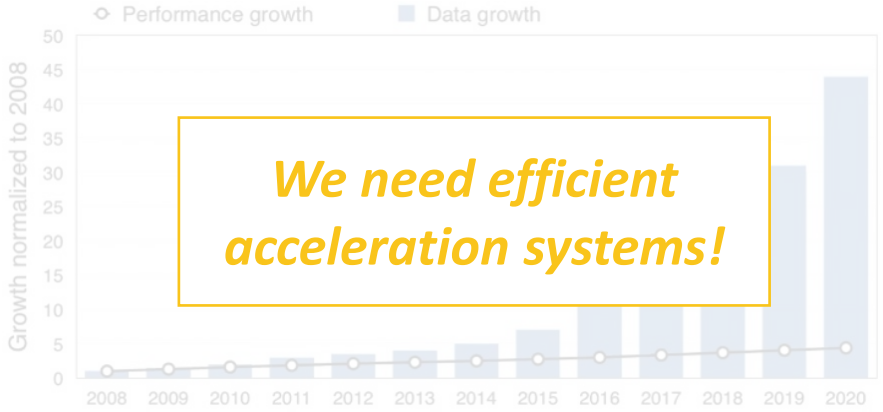
Complexity in Data



Complexity in Applications

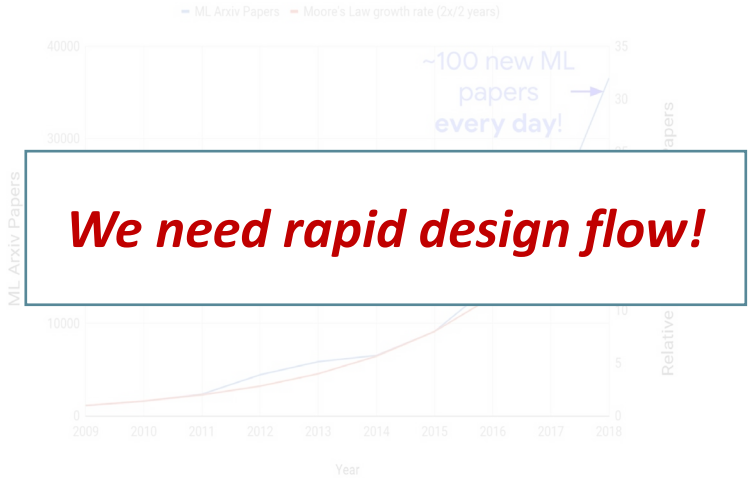


Complexity in Hardware



We need efficient acceleration systems!

Data growth trends: IDC's Digital Universe Study, December 2012
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011



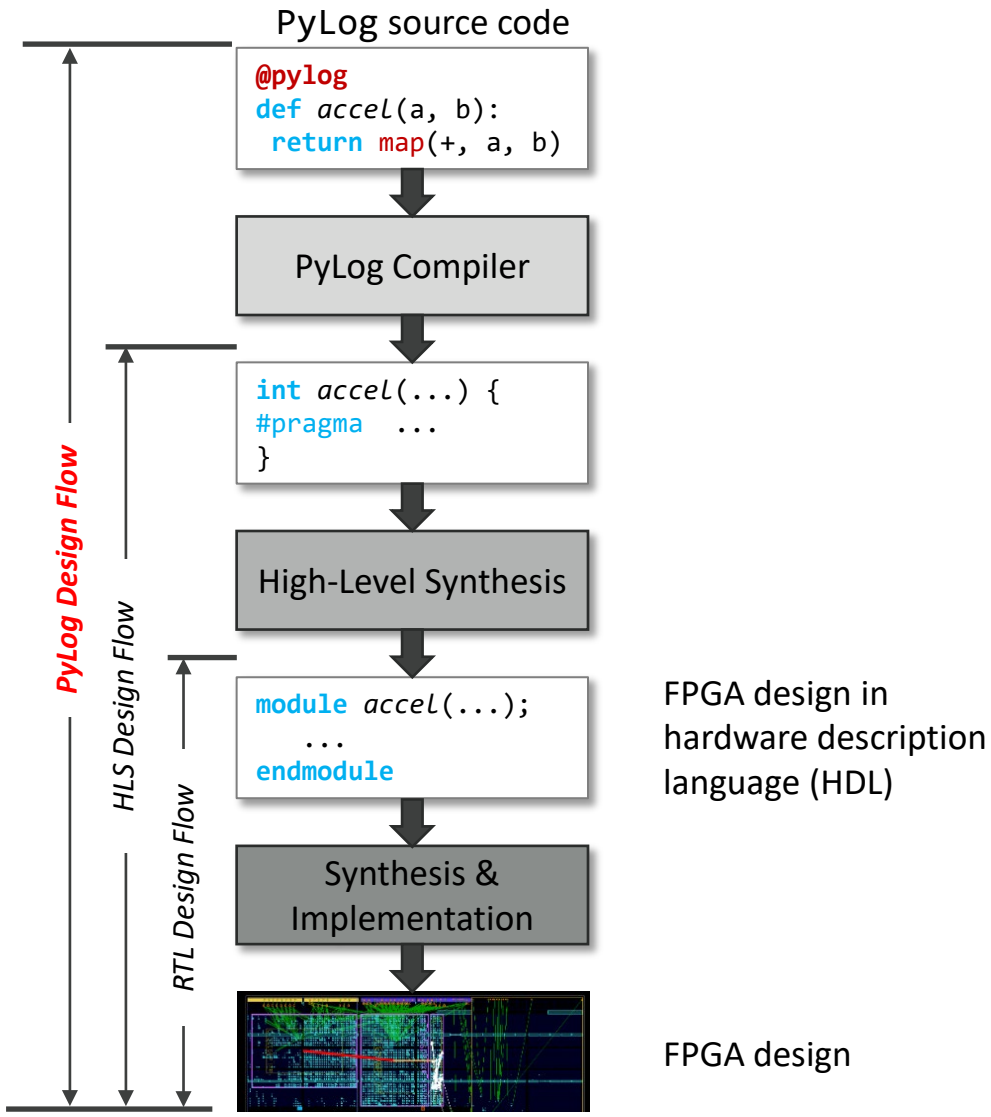
We need rapid design flow!

Jeff Dean. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design, arXiv 1911.05289.

Languages and Compilers for Hardware Accelerators

- *What?*
- *Why?*
- *How?*

Example 1: FPGA-based Accelerators, HLS, and PyLog



FPGA design in hardware description language (HDL)

FPGA design

Example 1: FPGA-based Accelerators, HLS, and PyLog

PyLog Code

```
import numpy as np
from pylog import *
@pylog                                  accelerator
def add(a, b):
  b = a + b
  b = map(lambda x, y: x+y, a, b)

if __name__ == "__main__":
  a = np.array([1, 3, ...])
  b = np.array([8, 9, ...])
  print(add(a, b))                      host
```

vecadd.py

PyLog Code Generator

- Analysis
- Type Inference
- Optimization
- Code Generation

HLS C Code

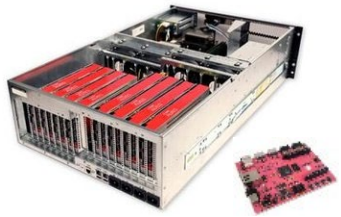
```
void accel(int32* a, int32* b,
int32* c, int32* d, int32* e) {
  for(int i = 0; i < 32; i++) {
    for (ii...)
    #pragma HLS ...
  }
  ...
}
```

PyLog System Generator

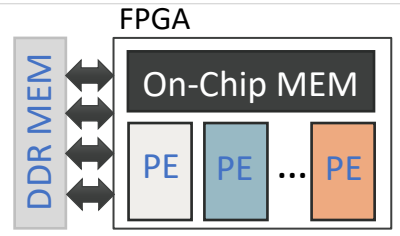
- Tcl Script
- IP Generation
- System Integration
- Design Generation

PyLog Runtime

- FPGA Programming
- Mem Allocation
- Start Accelerator
- Collect Results



Target Systems



System Design

```
>>> python3 vecadd.py
[10, 15, ...]
>>>
```

Execution Results

Synthesis Flow

Deployment Flow

Example 2: Tangram: Efficient GPU Code Generation

Goal: performance portable GPU code generation (for different GPU generations, different applications, etc.)

GPU programing strategies:

c_a

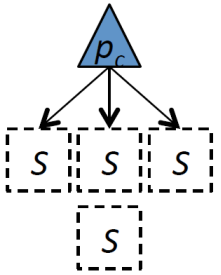
```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(a) Atomic autonomous codelet

c_b

```
__codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
  __shared int tmp[coopDim()];
  unsigned len = in.size();
  unsigned id = coopIdx();
  tmp[id] = (id < len)? in[id] : 0;
  for(unsigned s=1; s<coopDim(); s *= 2) {
    if(id >= s)
      tmp[id] += tmp[id - s];
  }
  return tmp[coopDim()-1];
}
```

(b) Atomic cooperative codelet

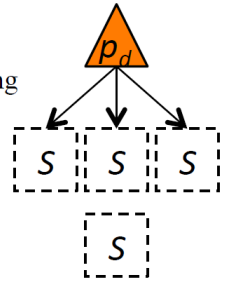



```
__codelet  __tag(asso_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,tile,len),sequence(1),sequence(tile,tile,len+1))))
}
```

(c) Compound codelet using adjacent tiling

```
__codelet __tag(stride_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,1,p),sequence(p),sequence((p-1)*tile,1,len+1))));
}
```

(d) Compound codelet using strided tiling



Example 2: Tangram: Efficient GPU Code Generation

Goal: performance portable GPU code generation (for different GPU generations, different applications, etc.)

GPU programing strategies:

Spectrum represents a unique computation with a defined set of inputs and outputs
Codelet represents a specific algorithmic implementation of a spectrum
A spectrum can have many codelets (versions), e.g., sequential, distribute, etc.

Tangram Composition Rules

Select(S_1, L_i) → Compose ($c_x \in S_1$, L_i);

Compose(c_x , L_i) → Devolve (c_x , L_i);
→ Distribute (c_m , L_i);
→ Compute ($c_{s/v}$, L_i);

Devolve(c_x, L_i) → Compose(c_x , L_{i-1});

Distribute(c_m , L_i) → Regroup([Compose(c_x , L_{i-1})$_1$, …, Compose(c_x , L_{i-1})$_P$] , L_i);

Regroup(P, L_i) → Select (S_2 , L_i);

Example 2: Tangram: Efficient GPU Code Generation

Goal: performance portable GPU code generation (for different GPU generations, different applications, etc.)

GPU programing strategies:

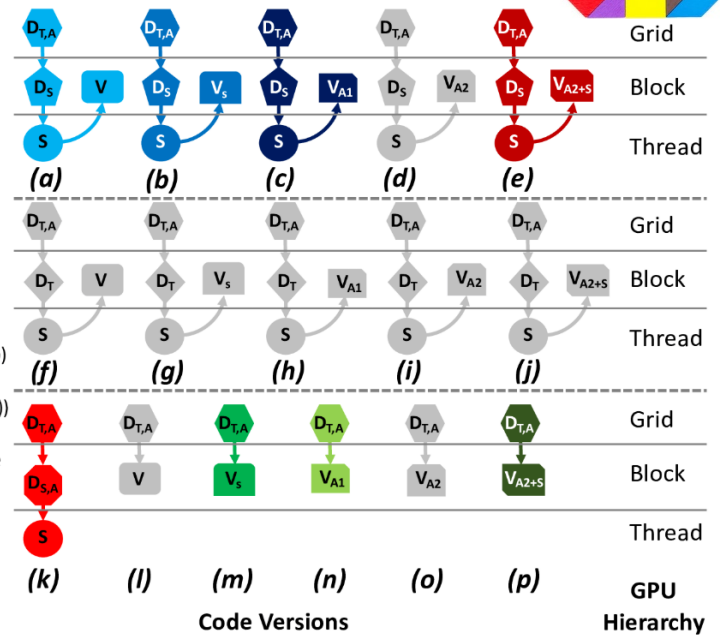
halide, LIFT, FUTHARK, CuNesl, kokkos, Tangram, Thrust, CUB, NVIDIA CUDA, OpenCL

High-level

Low-level

Spectrum represents a unique computation with a defined set of inputs and outputs
Codelet represents a specific algorithmic implementation of a spectrum
A spectrum can have many codelets (versions), e.g., sequential, distribute, etc.

- D_T Tile Distribute (Figure 1(b))
- D_S Stride Distribute (Figure 1(b))
- $D_{T,A}$ Global Atomic Tile Distribute
- $D_{S,A}$ Global Atomic Stride Distribute
- V Cooperative (Figure 1(c))
- V_s Cooperative + Shuffle
- V_{A1} Shared Memory Atomic 1 (Figure 3(a))
- V_{A2} Shared Memory Atomic 2 (Figure 3(b))
- V_{A2+S} Shared Memory Atomic 2 + Shuffle
- S Scalar (Figure 1(a))



Tangram Composition Rules — Codelets and Variants

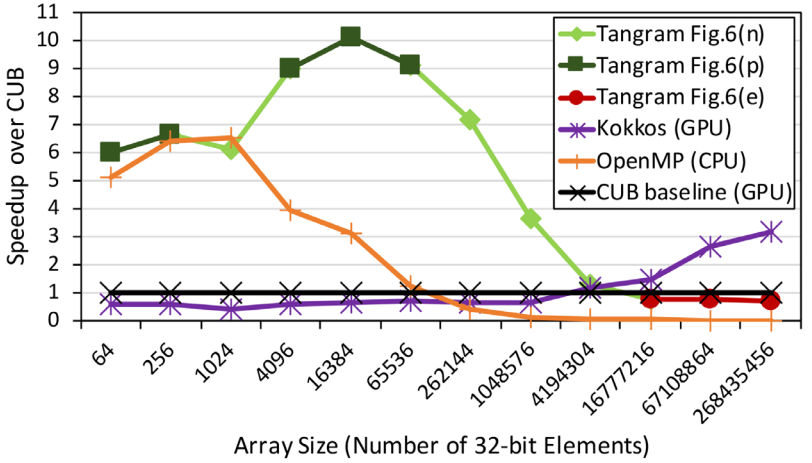
Select(S_1, L_i) → Compose ($c_x \in S_1$, L_i);

Compose(c_x , L_i) → Devolve (c_x , L_i);
→ Distribute (c_m , L_i);
→ Compute ($c_{s/v}$, L_i);

Devolve(c_x, L_i) → Compose(c_x , L_{i-1});

Distribute(c_m , L_i) → Regroup([Compose(c_x , L_{i-1})$_1$, …, Compose(c_x , L_{i-1})$_p$] , L_i);

Regroup(P, L_i) → Select (S_2 , L_i);



EECS 221:
Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu