Lecture 15:

Heterogeneous Parallelism, Hardware Specialization, DSLs

Parallel Computing

Stanford CS149, Fall 2021

Review: Transactional Memory

- Atomic construct: declaration that atomic behavior must be preserved by the system
  - Motivating idea: increase simplicity of synchronization without (significantly) sacrificing performance
- Transactional memory implementation
  - Many variants have been proposed: SW, HW, SW+HW
  - Implementations differ in:
    - Data versioning policy (eager vs. lazy)
    - Conflict detection policy (pessimistic vs. optimistic)
    - Detection granularity (object, word, cache line)
- Software TM systems (STM)
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code (e.g. StmRead, StmWrite)
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)
- Hardware Transactional Memory (HTM)
  - Versioned data is kept in caches
  - Conflict detection mechanisms augment coherence protocol

HTM Example: Transactional Coherence and Consistency

- Use TM as the coherence mechanism ➔ all transactions all the time
- Successful transaction commits update memory and all caches in the system

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

- Assumptions
  - One "commit" per execution step across all processors
  - When one transaction causes another transaction to abort and re-execute, assume that the transaction "commit" of one transaction can overlap with the "begin" of the re-executing transaction
  - Minimize the number of execution steps

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		
R D	A:0,D:0	A:1,C:2	B T3			R E	E:3	
C T1	A:0,D:0	A:1,C:2	W C, 5		C:4	W B, 6	E:3	B:6

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		
R D	A:0,D:0	A:1,C:2	B T3			R E	E:3	
C T1	A:0,D:0	A:1,C:2	W C, 5		C:4	W B, 6	E:3	B:6
			R A	A:1	C:4	W C, 7	E:3	B:6,C:7
			W E, 6	A:1	C:4,E:5	R F	E:3,F:0	B:6,C:7
				A:1	C:4,E:5	C T4	E:3,F:0	B:6,C:7

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		
R D	A:0,D:0	A:1,C:2	B T3			R E	E:3	
C T1	A:0,D:0	A:1,C:2	W C, 5		C:4	W B, 6	E:3	B:6
			R A	A:1	C:4	W C, 7	E:3	B:6,C:7
			W E, 6	A:1	C:4,E:5	R F	E:3,F:0	B:6,C:7
				A:1	C:4,E:5	C T4	E:3,F:0	B:6,C:7
			C T3	A:1	C:4,E:5			

I want to begin this lecture by reminding you…

In assignment 1 we observed that a well-optimized parallel implementation of a compute-bound application is about 40 times faster on my quad-core laptop than the output of single-threaded C code compiled with gcc -O3.

(In other words, a lot of software makes inefficient use of modern CPUs.)

Today we're going to talk about how inefficient the CPU in that laptop is, even if you are using it as efficiently as possible.

Heterogeneous processing

Observation: most "real world" applications have complex workload characteristics

They have components that can be widely parallelized.

And components that are difficult to parallelize.

They have components that are amenable to wide SIMD execution.

And components that are not. (divergent control flow)

They have components with predictable data access

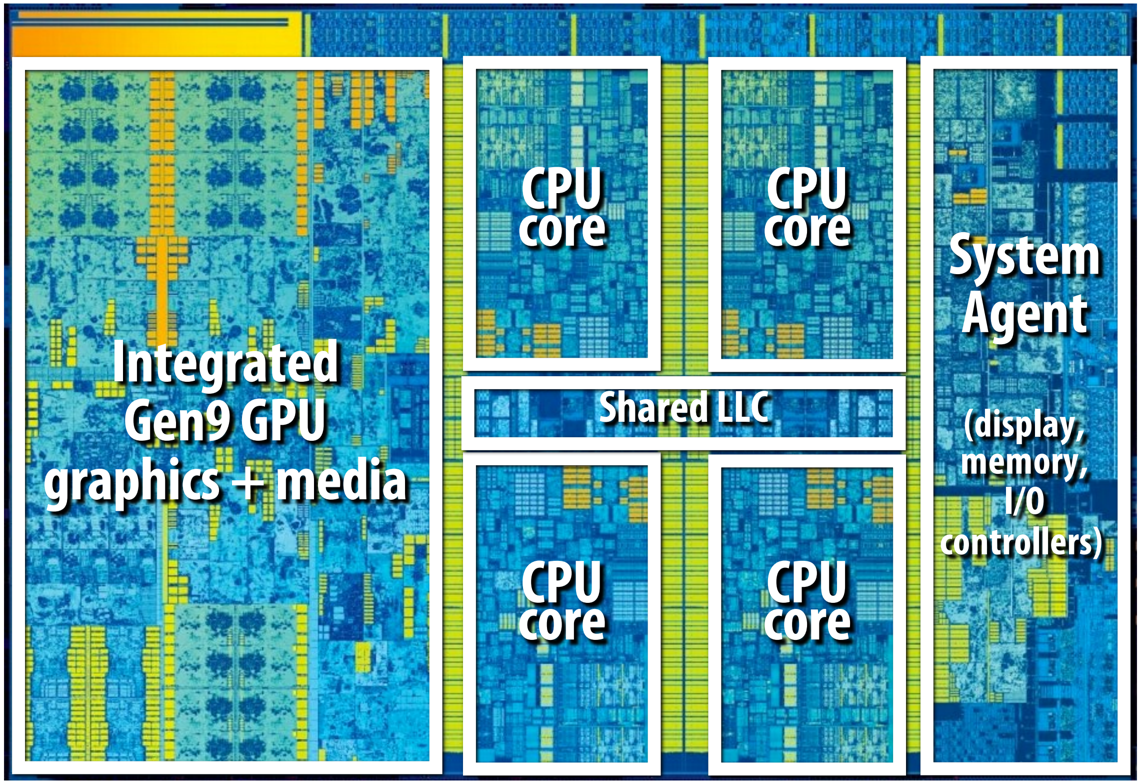
And components with unpredictable access, but those accesses might cache well.

Idea: the most efficient processor is a heterogeneous mixture of resources ("use the most efficient tool for the job")

Examples of heterogeneity

Example: Intel “Skylake" (2015)

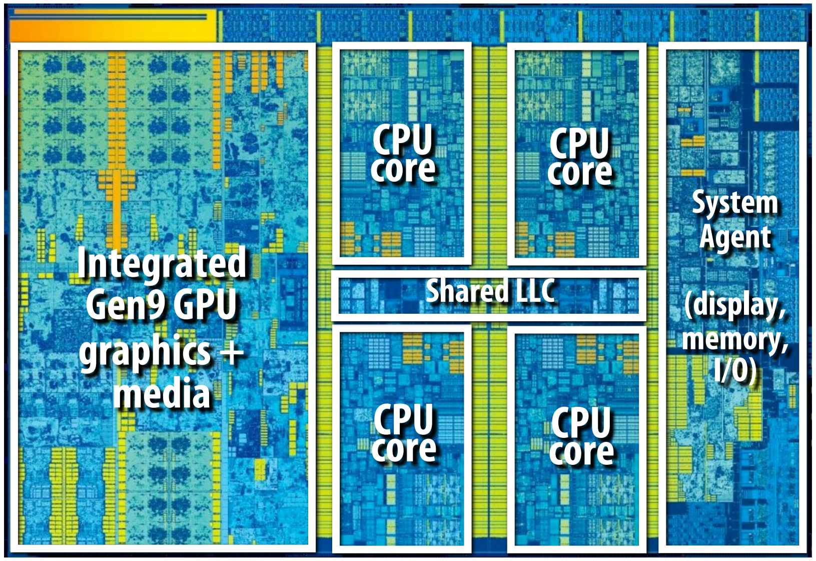
(6th Generation Core i7 architecture)


Integrated Gen9 GPU graphics + media
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent
(display, memory, I/O controllers)

4 CPU cores + graphics cores + media accelerators

Example: Intel “Skylake" (2015)

(6th Generation Core i7 architecture)

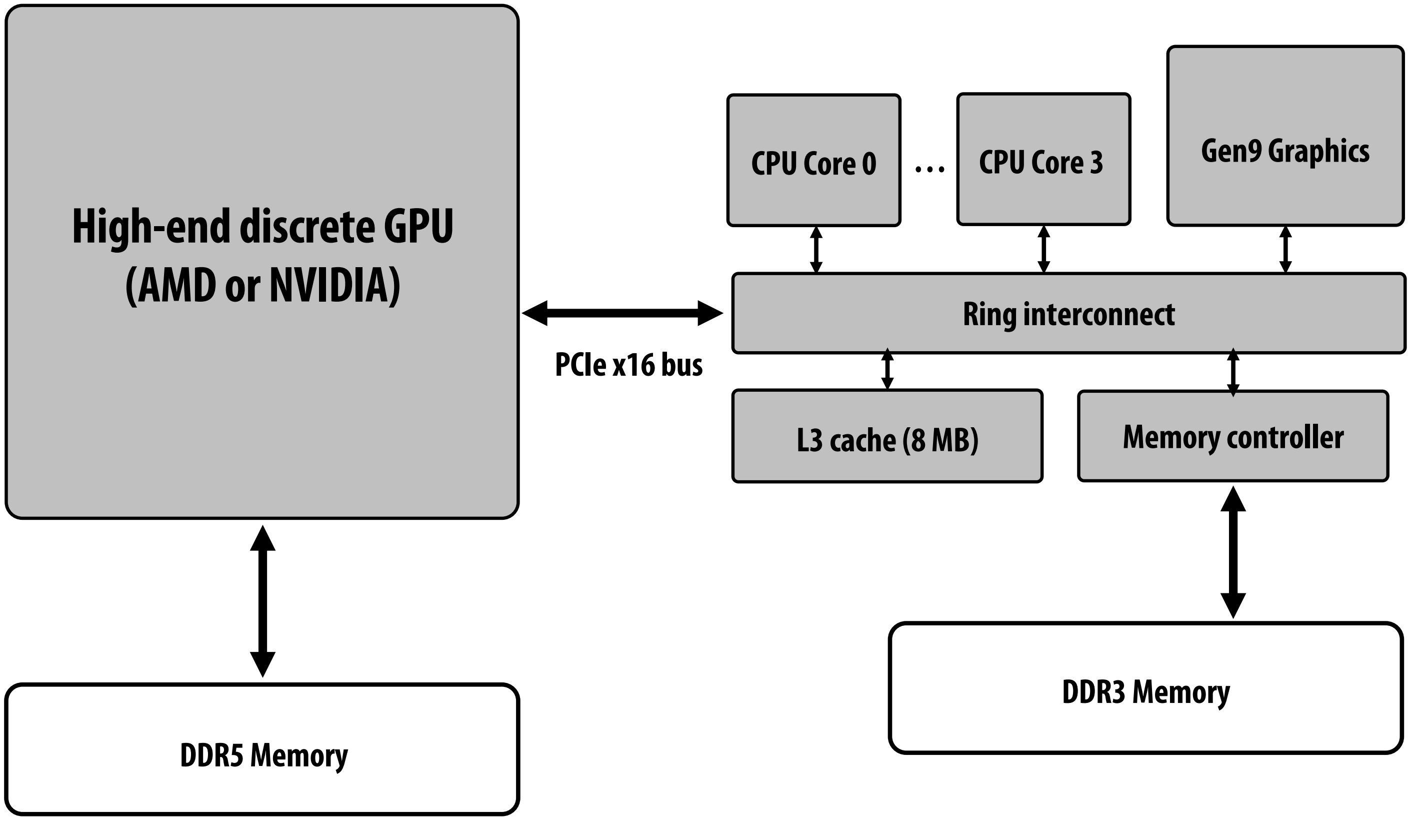


- CPU cores and graphics cores share same memory system
- Also share LLC (L3 cache)
  - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU
- Graphics cores are cache coherent with CPU cores

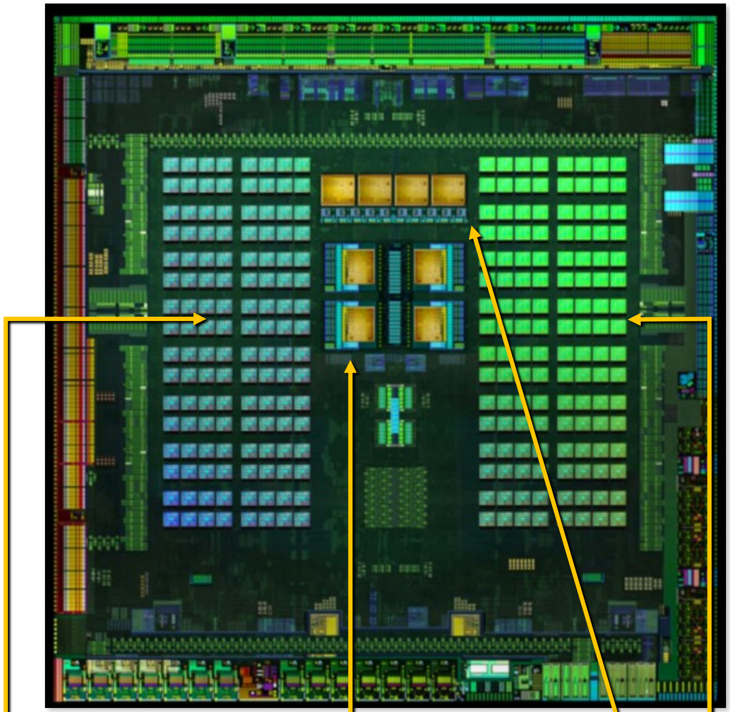
More heterogeneity: add discrete GPU

Keep discrete (power hungry) GPU unless needed for graphics-intensive applications

Use integrated, low power graphics for basic graphics/window manager/UI



Mobile heterogeneous processors

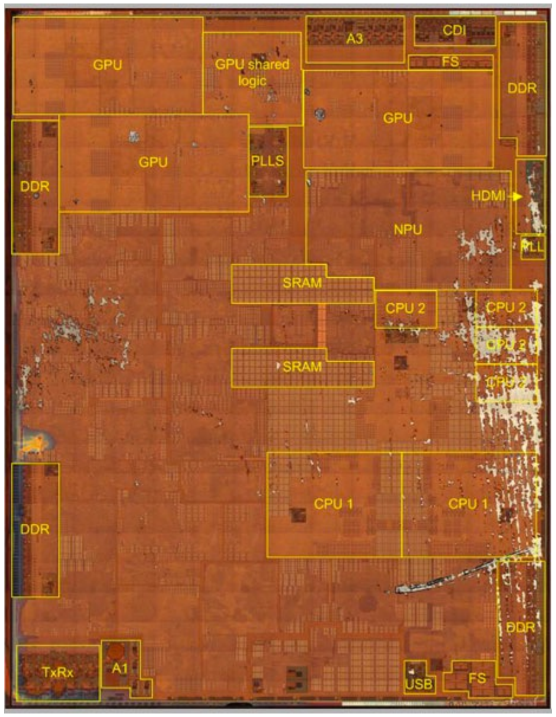


NVIDIA Tegra X1

Four ARM Cortex A57 CPU cores for applications

Four low performance (low power) ARM A53 CPU cores

One Maxwell SMM (256 “CUDA” cores)



Apple A11 Bionic *

Two “high performance” 64 bit ARM CPU cores

Four “low performance” ARM CPU cores

Three “core” Apple-designed GPU

Image processor

Neural Engine for DNN acceleration

Motion processor

A11 image credit: TechInsights Inc.’

* Disclaimer: estimates by TechInsights, not an official Apple reference.

GPU-accelerated supercomputing

Summit (at Oak Ridge National Lab)
(world’s #1 in Fall 2018)
9,216 IBM Power9 22-core CPUs
27,648 NVIDIA V100 GPUs
10 Petabytes DRAM

Energy-constrained computing

Performance and Power

Performance

Energy efficiency

$$Power = \frac{Ops}{second} \times \frac{Joules}{Op}$$

FIXED

↑

↓

What is the magnitude of improvement from specialization?

Specialization (fixed function) ⇒ better energy efficiency

Pursuing highly efficient processing…
(specializing hardware beyond just parallel CPUs and GPUs)

Efficiency benefits of compute specialization

- Rules of thumb: compared to high-quality C code on CPU...
- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf / watt
  - Assuming code maps well to wide data-parallel execution and is compute bound

- Fixed-function ASIC ("application-specific integrated circuit")
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound and is not floating-point math

[Source: Chung et al. 2010 , Dally 08]

Why is a “general-purpose processor” so inefficient?

Wait… this entire class we’ve been talking about making efficient use out of multi-core CPUs and GPUs…
and now you’re telling me these platforms are “inefficient”?

Consider the complexity of executing an instruction on a modern processor…

Read instruction ⟞ Address translation, communicate with icache, access icache, etc.

Decode instruction ⟞ Translate op to uops, access uop cache, etc.

Check for dependencies/pipeline hazards

Identify available execution resource

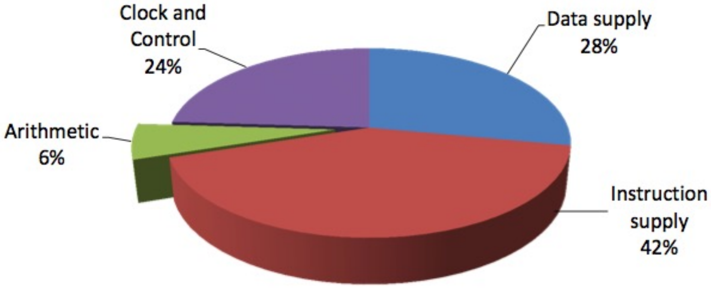
Use decoded operands to control register file SRAM (retrieve data)

Move data from register file to selected execution resource

Perform arithmetic operation

Move data from execution resource to register file

Use decoded operands to control write to register file SRAM



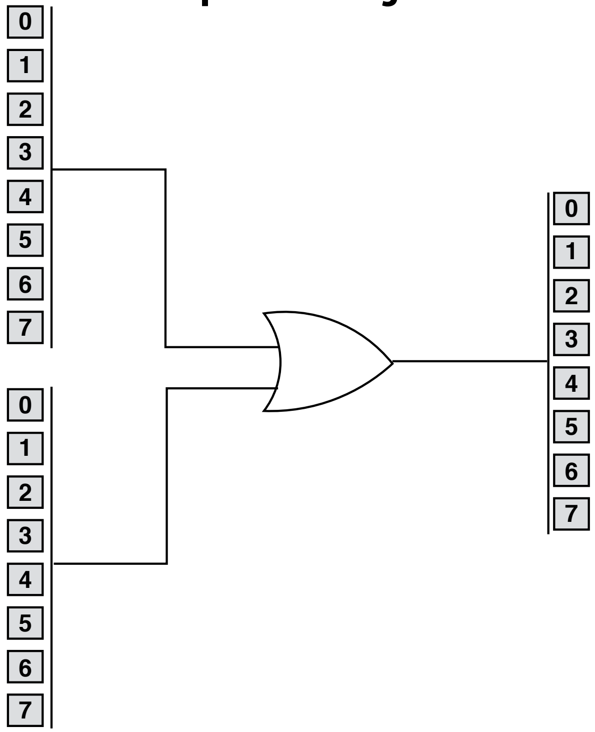
Efficient Embedded Computing [Dally et al. 08]

[Figure credit Eric Chung]

Review question:
How does SIMD execution reduce overhead of certain types of computations?
What properties must these computations have?

Contrast that complexity to the circuit required to actually perform the operation

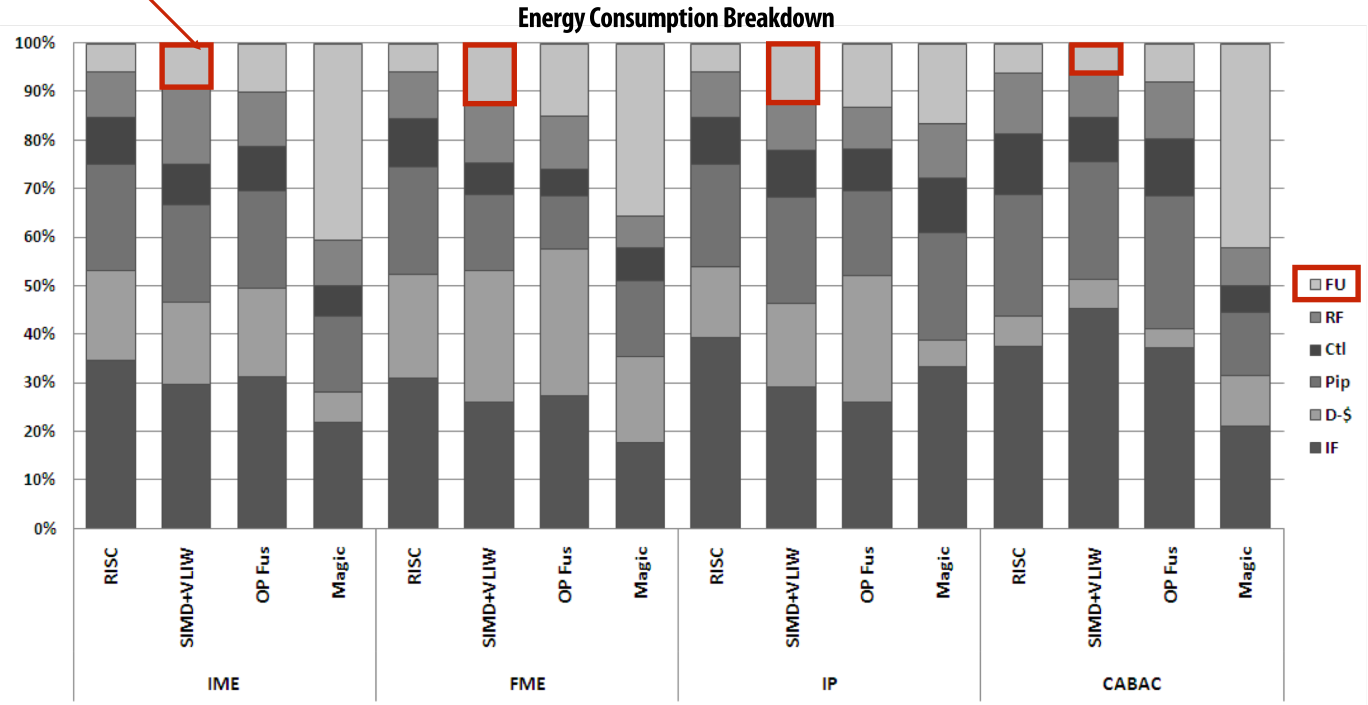
Example: 8-bit logical OR



H.264 video encoding: fraction of energy consumed by functional units is small (even when using SIMD)

Even after encoding implemented with SIMD instruction

[Hameed et al. ISCA 2010]



integer motion estimation

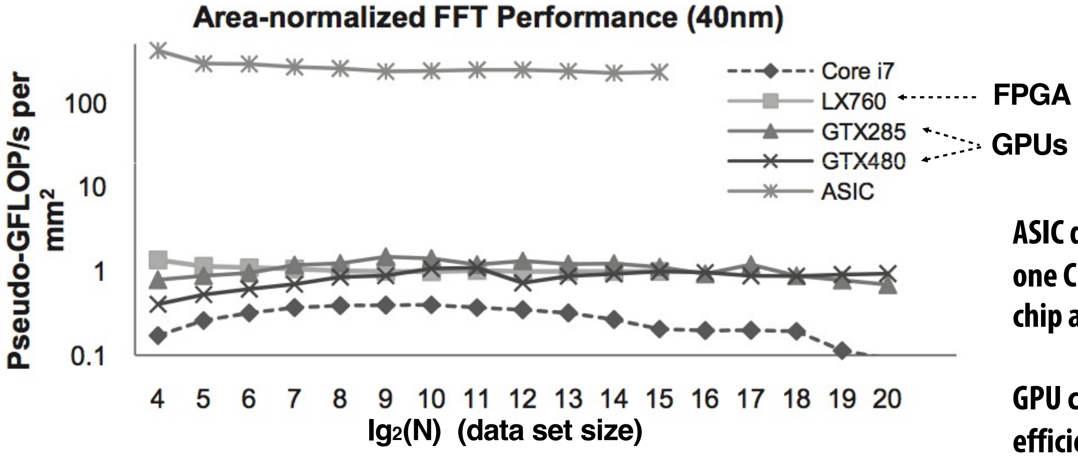
fractional (subpixel) motion estimation

intra-frame prediction, DTC, quantization

arithmetic encoding

FU = functional units	Pip = pipeline registers (interstage)
RF = register fetch	D-$ = data cache
Ctrl = misc pipeline control	IF = instruction fetch + instruction cache

Fast Fourier transform (FFT): throughput and energy benefits of specialization

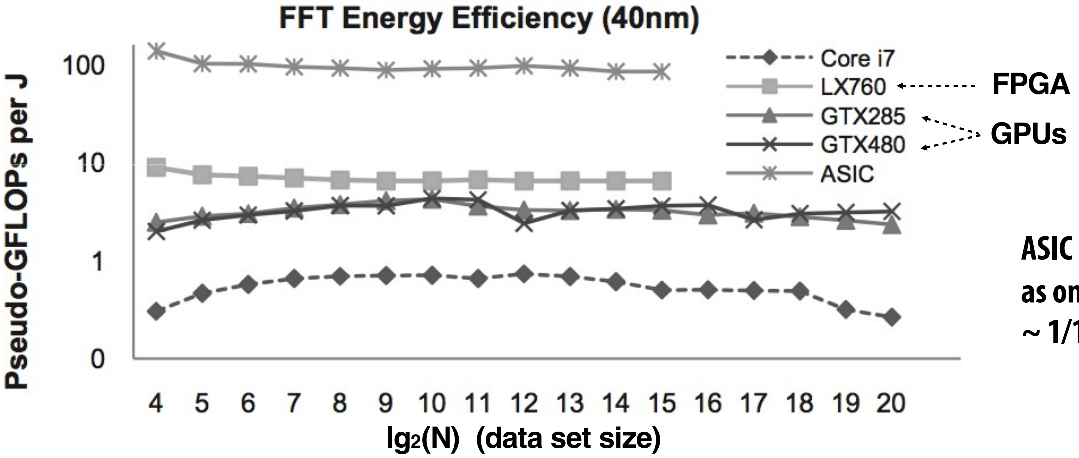


FPGA

GPUs

ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.



FPGA

GPUs

ASIC delivers same performance as one CPU core using only ~ 1/100th the power

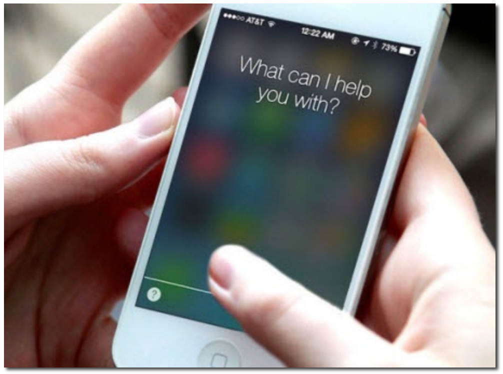
[Chung et al. MICRO 2010]

Mobile: benefits of increasing efficiency

- Run faster for a fixed period of time
  - Run at higher clock, use more cores (reduce latency of critical task)
  - Do more at once

- Run at a fixed level of performance for longer
  - e.g., video playback, health apps
  - Achieve "always-on" functionality that was previously impossible

Google Glass: ~40 min recording per charge (nowhere near "always on")



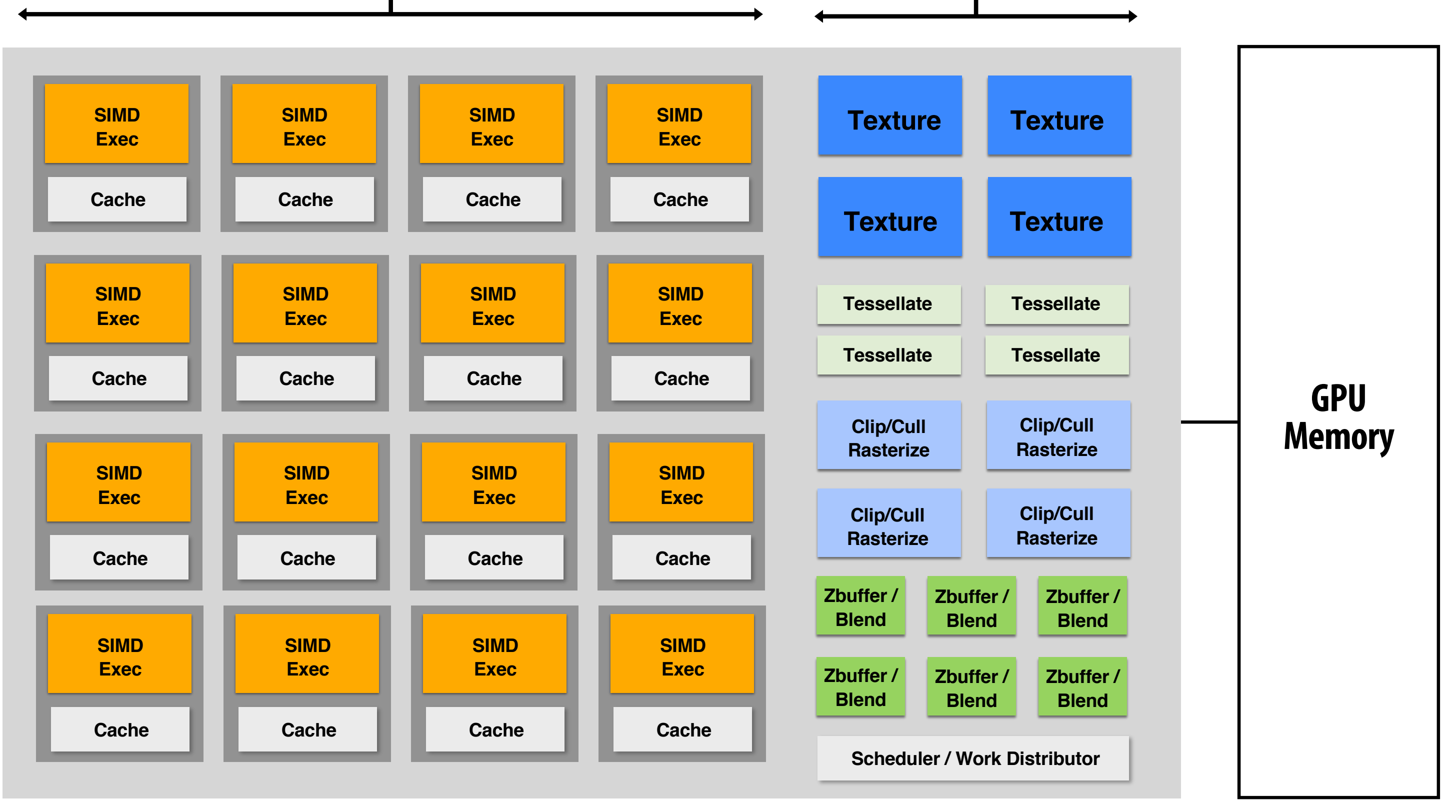
iPhone:
Siri activated by button press or holding phone up to ear

Amazon Echo / Google Home
Always listening

GPU's are themselves heterogeneous multi-core processors

Compute resources your CUDA programs used in Assignment 2

Graphics-specific, fixed-function compute resources



SIMD Exec
Cache
(× 16)

Texture
Texture
Texture
Texture

Tessellate
Tessellate
Tessellate
Tessellate

Clip/Cull Rasterize
Clip/Cull Rasterize
Clip/Cull Rasterize
Clip/Cull Rasterize

Zbuffer / Blend
Zbuffer / Blend
Zbuffer / Blend
Zbuffer / Blend
Zbuffer / Blend
Zbuffer / Blend

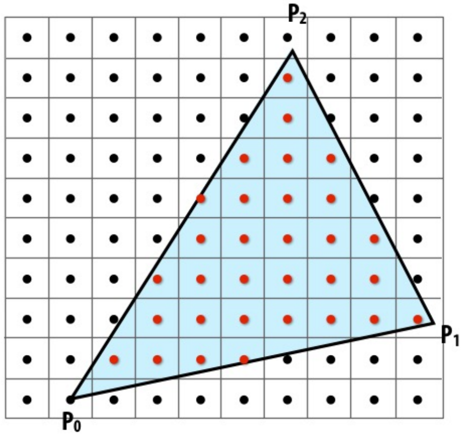
Scheduler / Work Distributor

GPU Memory

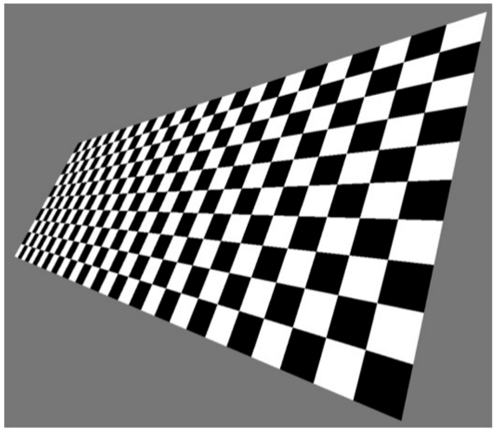
GPU

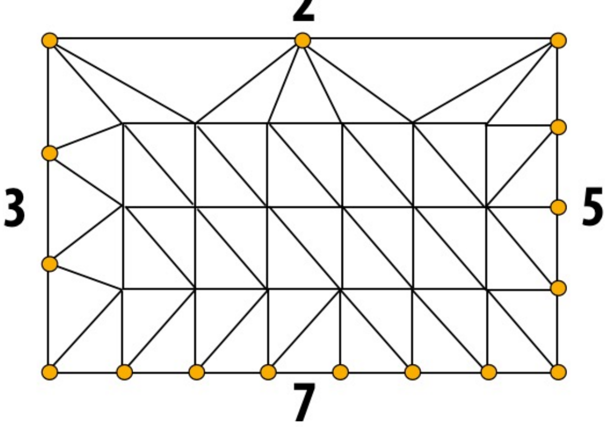
Example graphics tasks performed in fixed-function HW

Rasterization:
Determining what pixels a triangle overlaps



Texture mapping:
Warping/filtering images to apply detail to surfaces





Geometric tessellation:
computing fine-scale geometry
from coarse geometry

Digital signal processors (DSPs)

Programmable processors, but simpler instruction stream control paths

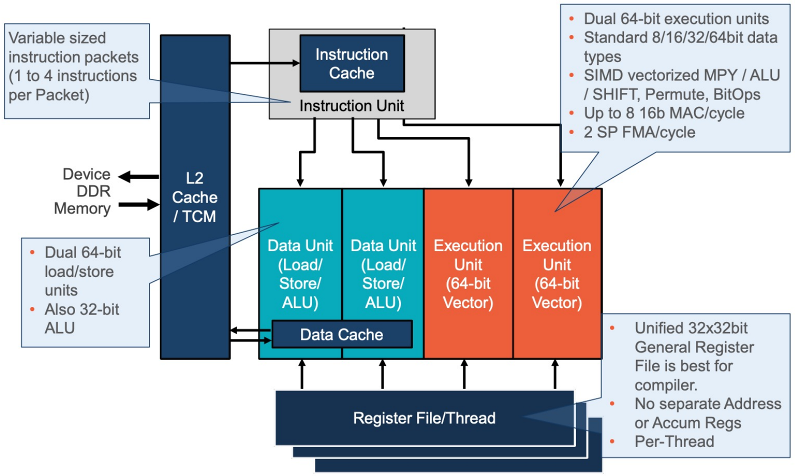
Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction (amortize cost of control)

Example: Qualcomm Hexagon DSP

Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors

VLIW: "very-long instruction word"
Single instruction specifies multiple different operations to do at once (contrast to SIMD)

Below: innermost loop of FFT
Hexagon DSP performs 29 "RISC" ops per cycle


Instruction Cache
Instruction Unit
L2 Cache / TCM
Device DDR Memory
Data Unit (Load/ Store/ ALU)
Data Unit (Load/ Store/ ALU)
Execution Unit (64-bit Vector)
Execution Unit (64-bit Vector)
Data Cache
Register File/Thread

Variable sized instruction packets (1 to 4 instructions per Packet)

- Dual 64-bit execution units
- Standard 8/16/32/64bit data types
- SIMD vectorized MPY / ALU / SHIFT, Permute, BitOps
- Up to 8 16b MAC/cycle
- 2 SP FMA/cycle

- Dual 64-bit load/store units
- Also 32-bit ALU

- Unified 32x32bit General Register File is best for compiler.
- No separate Address or Accum Regs
- Per-Thread

64-bit Load and

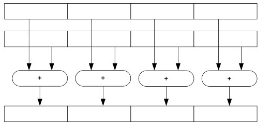
64-bit Store with post-update addressing

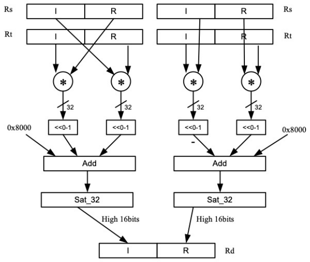
```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

Complex multiply with round and saturation

Zero-overhead loops
- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



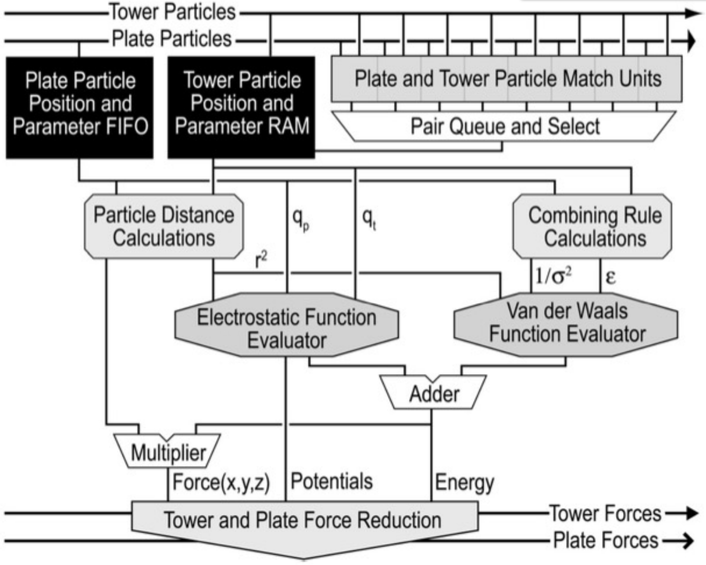


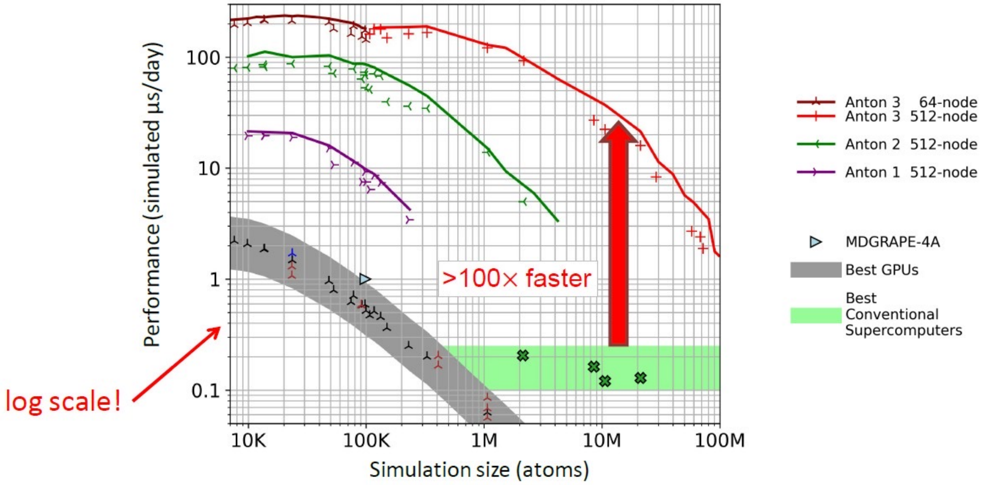
Hexagon DSP is in Google Pixel phone

Anton supercomputer for molecular dynamics

[Developed by DE Shaw Research]

- Simulates time evolution of proteins
- ASIC for computing particle-particle interactions (512 of them in machine)
- Throughput-oriented subsystem for efficient fast-fourier transforms
- Custom, low-latency communication network designed for communication patterns of N-body simulations

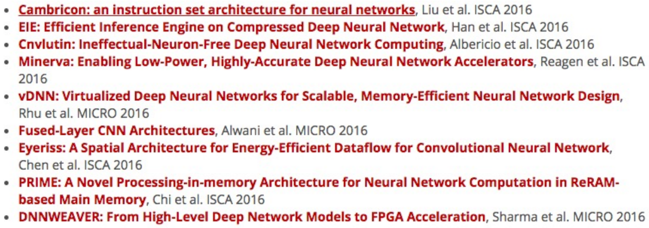

Tower Particles
Plate Particles
Plate Particle Position and Parameter FIFO
Tower Particle Position and Parameter RAM
Plate and Tower Particle Match Units
Pair Queue and Select
Particle Distance Calculations
q_p
q_t
Combining Rule Calculations
r^2
$1/\sigma^2$
ε
Electrostatic Function Evaluator
Van der Waals Function Evaluator
Adder
Multiplier
Force(x,y,z)
Potentials
Energy
Tower and Plate Force Reduction
Tower Forces
Plate Forces



log scale!

Specialized processors for evaluating deep networks

Countless recent papers at top computer architecture research conferences on the topic of ASICs or accelerators for deep learning or evaluating deep networks…



- Cambricon: an instruction set architecture for neural networks, Liu et al. ISCA 2016
- EIE: Efficient Inference Engine on Compressed Deep Neural Network, Han et al. ISCA 2016
- Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing, Albericio et al. ISCA 2016
- Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators, Reagen et al. ISCA 2016
- vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, Rhu et al. MICRO 2016
- Fused-Layer CNN Architectures, Alwani et al. MICRO 2016
- Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Network, Chen et al. ISCA 2016
- PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory, Chi et al. ISCA 2016
- DNNWEAVER: From High-Level Deep Network Models to FPGA Acceleration, Sharma et al. MICRO 2016

Example: Google's Tensor Processing Unit (TPU)
Accelerates deep learning operations

Intel Lake Crest ML accelerator
(formerly Nervana)

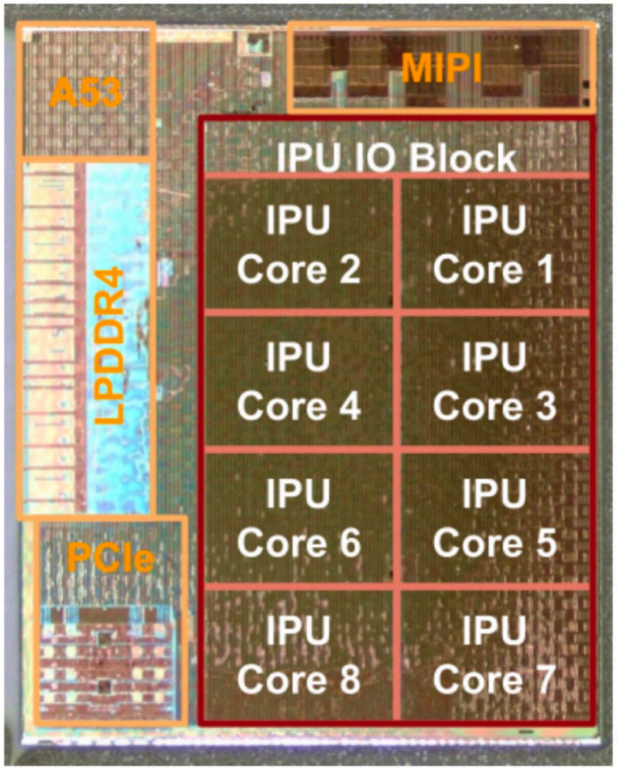
Example: Google's Pixel Visual Core

Programmable "image processing unit" (IPU)

- Each core = 16x16 grid of 16 bit multiply-add ALUs

- ~10-20x more efficient than GPU at image processing tasks

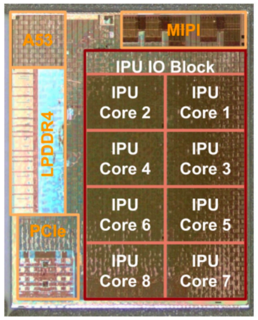
  (Google's claims at HotChips '18)

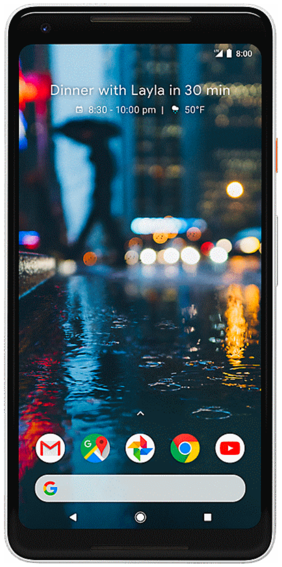


Let’s crack open a modern smartphone

Google Pixel 2 Phone:
Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core

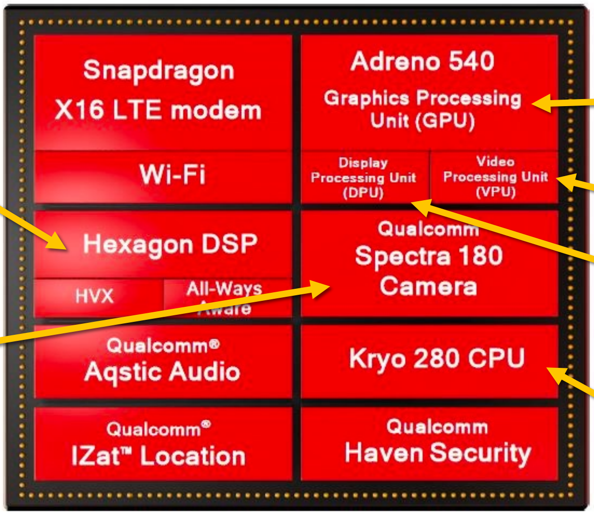
Visual Pixel Core
Programmable image processor and DNN accelerator





“Hexagon” Programmable DSP
data-parallel multi-media processing

Image Signal Processor
ASIC for processing camera sensor pixels



Multi-core GPU
(3D graphics, OpenCL data-parallel compute)

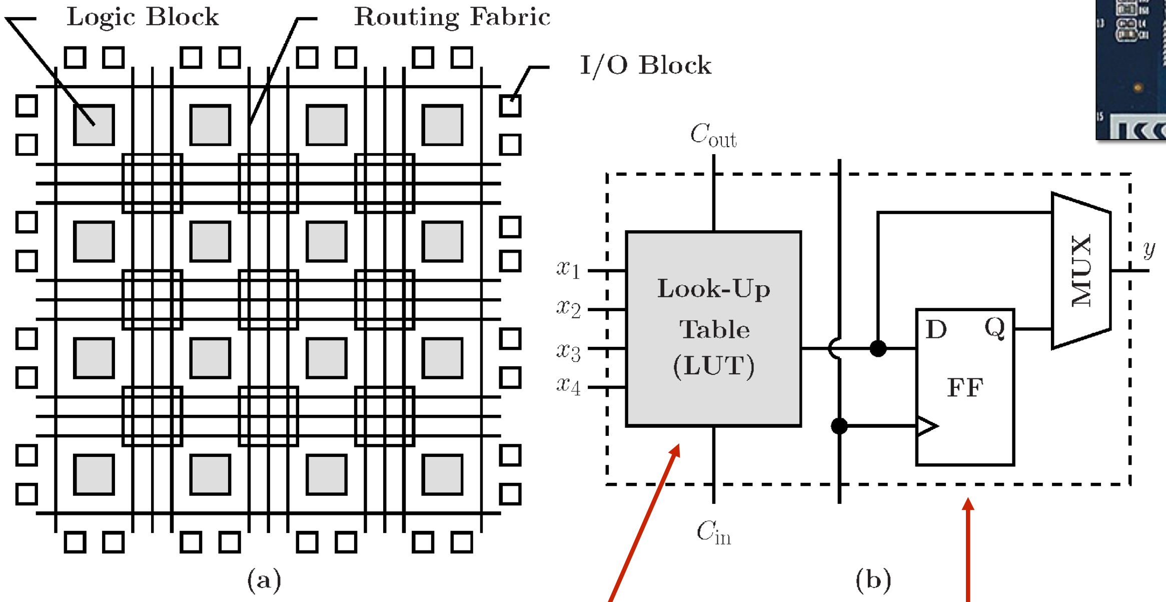
Video encode/decode ASIC

Display engine
(compresses pixels for transfer to high-res screen)

Multi-core ARM CPU
4 “big cores” + 4 “little cores”

FPGAs (Field Programmable Gate Arrays)

- Middle ground between an ASIC and a processor
- FPGA chip provides array of logic blocks, connected by interconnect
- Programmer-defined logic implemented directly by FGPA



(a)

(b)

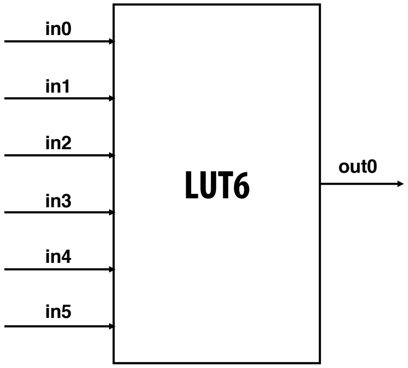
Programmable lookup table (LUT)

Flip flop (a register)

Image credit: Bai et al. 2014

Specifying combinatorial logic as a LUT

- Example: 6-input, 1 output LUT in Xilinx Virtex-7 FPGAs
  - Think of a LUT6 as a 64 element table



Example:
6-input AND

In	Out
0	0
1	0
2	0
3	0
⋮	⋮
63	1

40-input AND constructed by chaining outputs of eight LUT6's (delay = 3)

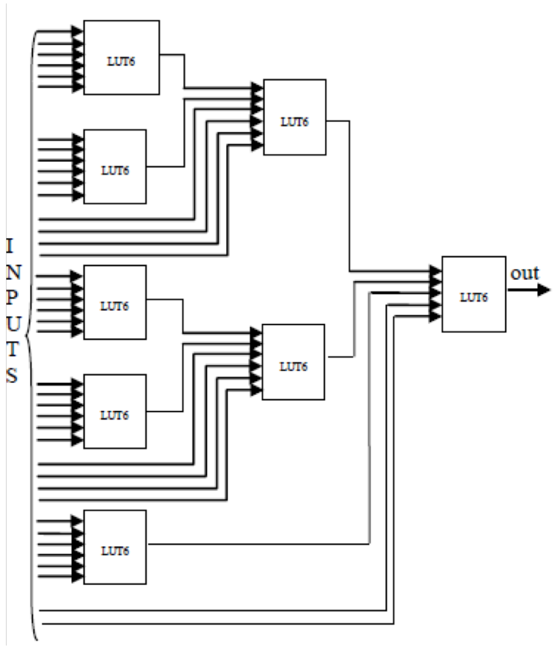
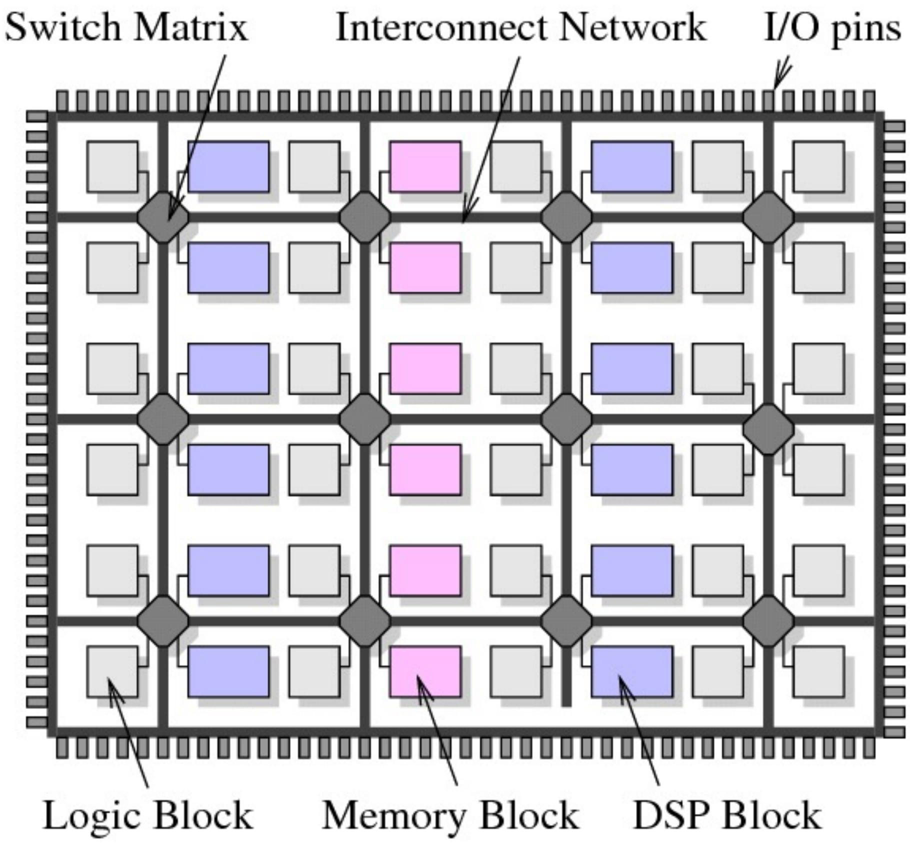


Image credit: [Zia 2013]

Modern FPGAs



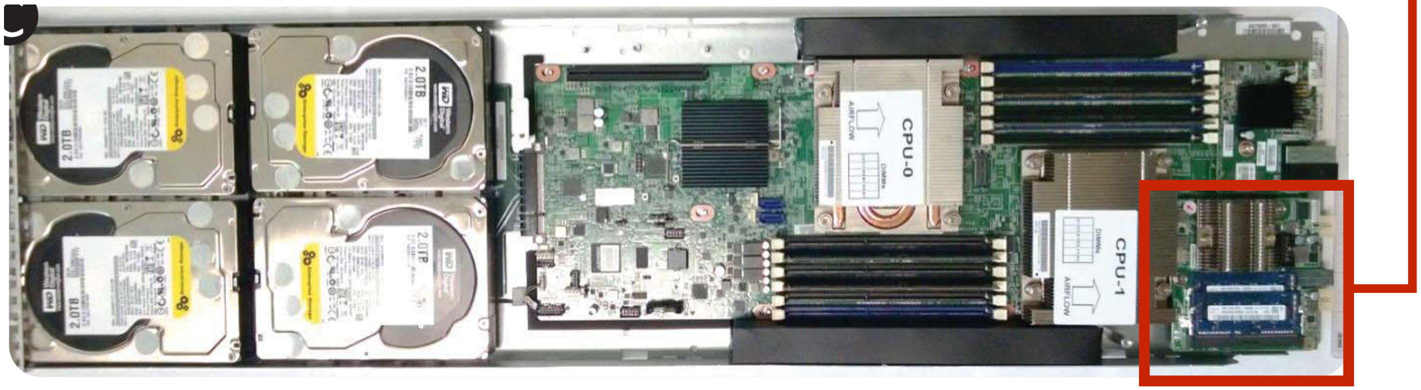
- **A lot of area devoted to hard gates**
  - **Memory blocks (SRAM)**
  - **DSP blocks (multiplier)**

Project Catapult

[Putnam et al. ISCA 2014]

- Microsoft Research investigation of use of FPGAs to accelerate datacenter workloads
- Demonstrated offload of part of Bing search's document ranking logic

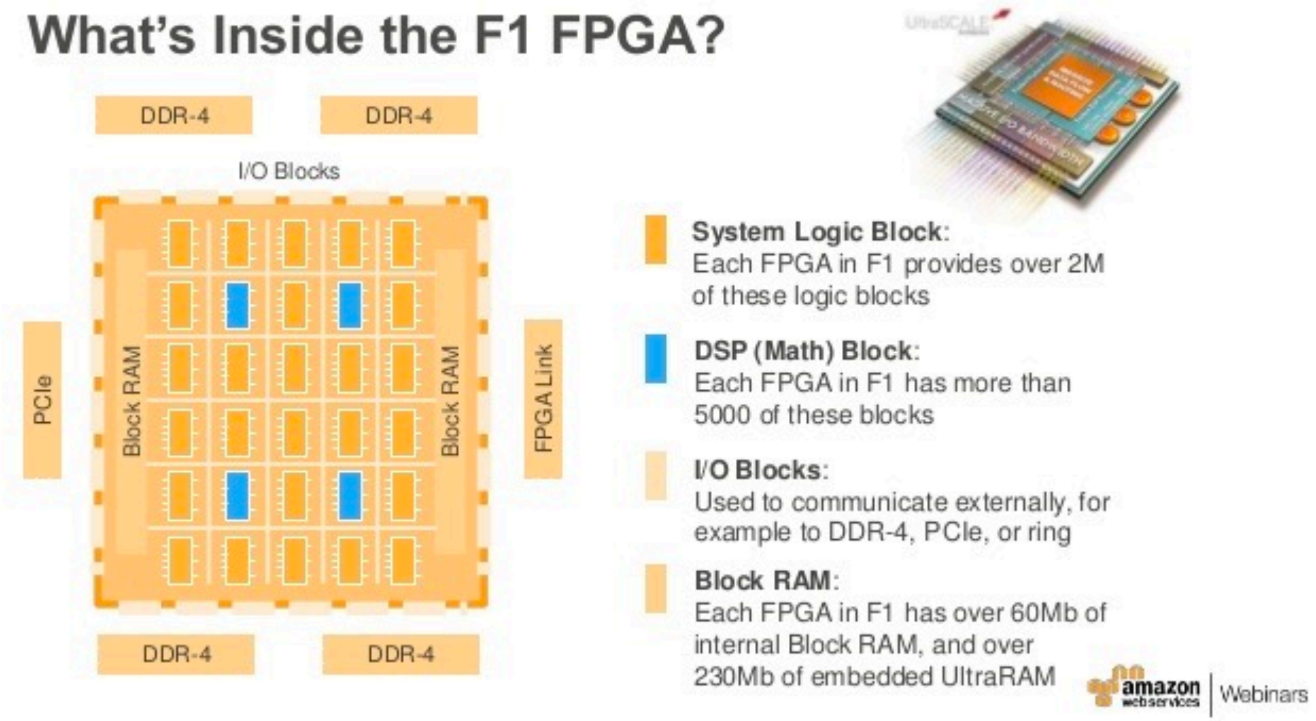
FPGA board



1U server (Dual socket CPU + FPGA connected via PCIe bus)

Amazon F1

- **FPGA's are now available on Amazon cloud services**



What's Inside the F1 FPGA?

🟧 **System Logic Block**:
Each FPGA in F1 provides over 2M of these logic blocks

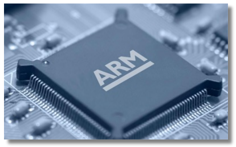
🟦 **DSP (Math) Block**:
Each FPGA in F1 has more than 5000 of these blocks

⬜ **I/O Blocks**:
Used to communicate externally, for example to DDR-4, PCIe, or ring

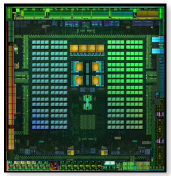
🟧 **Block RAM**:
Each FPGA in F1 has over 60Mb of internal Block RAM, and over 230Mb of embedded UltraRAM

Summary: choosing the right tool for the job

Energy-optimized CPU



Throughput-oriented processor (GPU)



Programmable DSP

HEXAGON™

FPGA/ reconfigurable logic

ASIC

Video encode/decode,
Audio playback,
Camera RAW processing,
neural nets (future?)

~10X more efficient

~100X???
(jury still out)

~100-1000X
more efficient

Easiest to program

Difficult to program (making it easier is active area of research)

Not programmable + costs 10-100's millions of dollars to design / verify / create

Credit: Pat Hanrahan for this slide design

Challenges of heterogeneous designs:

(it’s not easy to realize the potential of specialized, heterogeneous processing)

Challenges of heterogeneity

- Heterogeneous system: preferred processor for each task
- Challenge to software developer: how to map application onto a heterogeneous collection of resources?
  - Challenge: "Pick the right tool for the job": design algorithms that decompose into components that each map well to different processing components of the machine
  - The scheduling problem is more complex on a heterogeneous system
- Challenge for hardware designer: what is the right mixture of resources?
  - Too few throughput oriented resources (lower peak throughput for parallel workloads)
  - Too few sequential processing resources (limited by sequential part of workload)
  - How much chip area should be dedicated to a specific function, like video?

Reducing energy consumption idea 1: use specialized processing

(use the right processor for the job)

Reducing energy consumption idea 2: move less data

Data movement has high energy cost

- Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory
  - Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption
- "Ballpark" numbers [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ ← Suggests that recomputing values, rather than storing and reloading them, is a better answer when optimizing code for energy efficiency!
- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

Three trends in energy-optimized computing

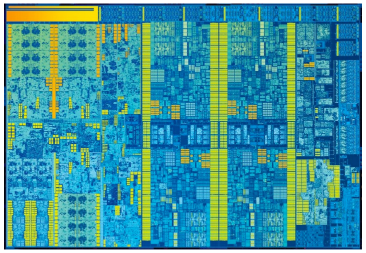
- **Compute less!**
  - Computing costs energy: parallel algorithms that do more work than sequential counterparts may not be desirable even if they run faster
- **Specialize compute units:**
  - Heterogeneous processors: CPU-like cores + throughput-optimized cores (GPU-like cores)
  - Fixed-function units: audio processing, "movement sensor processing" video decode/encode, image processing/computer vision?
  - Specialized instructions: expanding set of AVX vector instructions, new instructions for accelerating AES encryption (AES-NI)
  - Programmable soft logic: FPGAs
- **Reduce bandwidth requirements**
  - Exploit locality (restructure algorithms to reuse on-chip data as much as possible)
  - Aggressive use of compression: perform extra computation to compress application data before transferring to memory (likely to see fixed-function HW to reduce overhead of general data compression/decompression)

Summary: heterogeneous processing for efficiency

- Heterogeneous parallel processing: use a mixture of computing resources that fit mixture of needs of target applications
  - Latency-optimized sequential cores, throughput-optimized parallel cores, domain-specialized fixed-function processors
  - Examples exist throughout modern computing: mobile processors, servers, supercomputers
- Traditional rule of thumb in “good system design” is to design simple, general-purpose components
  - This is not the case in emerging systems (optimized for perf/watt)
  - Today: want collection of components that meet perf requirement AND minimize energy use
- Challenge of using these resources effectively is pushed up to the programmer
  - Current CS research challenge: how to write efficient, portable programs for emerging heterogeneous architectures?

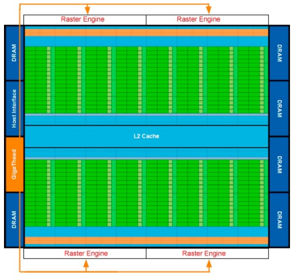
Heterogeneous Parallel Programming Today

Pthreads
OpenMP
IPSC



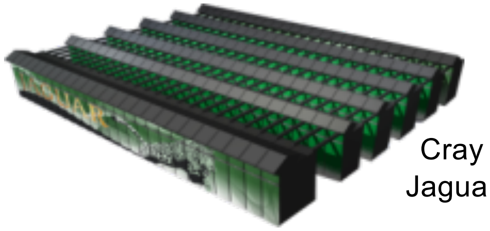
Intel
Skylake

CUDA
OpenCL



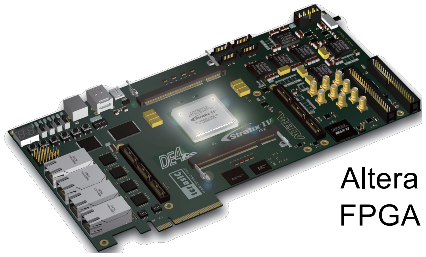
Nvidia
Fermi

MPI
PGAS
Spark



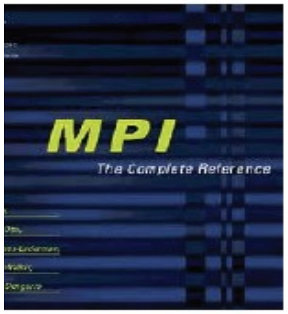
Cray
Jaguar

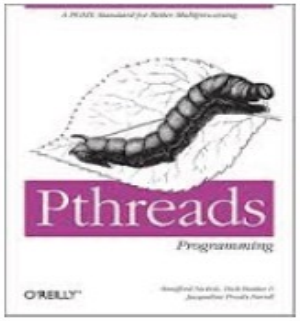
Verilog
VHDL

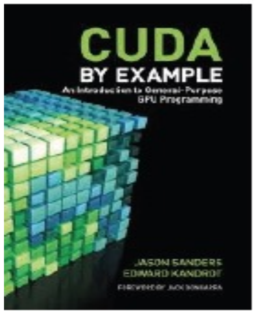


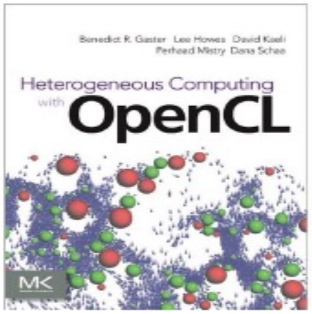
Altera
FPGA

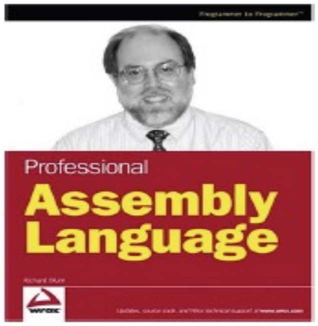
EXPERT PROGRAMMERS ⇒ LOW PRODUCTIVITY

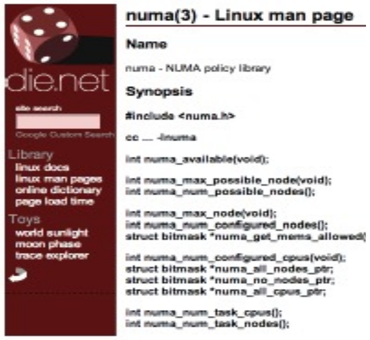







Benedict R. Gaster Lee Howes David Kaeli
Perhaad Mistry Dana Schaa
Heterogeneous Computing with OpenCL
MK





numa(3) - Linux man page

Name

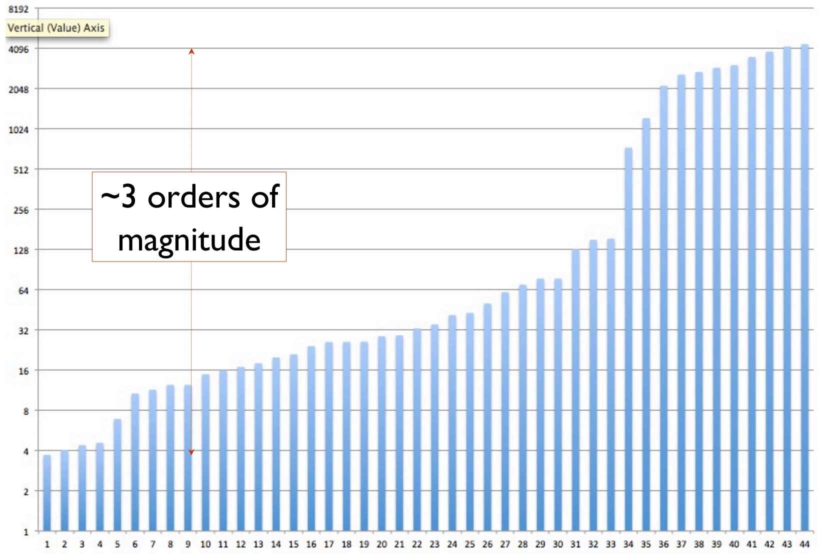
numa - NUMA policy library

Synopsis

```
#include <numa.h>

cc ... -lnuma

int numa_available(void);

int numa_max_possible_node(void);
int numa_num_possible_nodes();

int numa_max_node(void);
int numa_num_configured_nodes();
struct bitmask *numa_get_mems_allowed(

int numa_num_configured_cpus(void);
struct bitmask *numa_all_nodes_ptr;
struct bitmask *numa_no_nodes_ptr;
struct bitmask *numa_all_cpus_ptr;

int numa_num_task_cpus();
int numa_num_task_nodes();
```

Expert Programming is Difficult

Image Filter in OpenMP

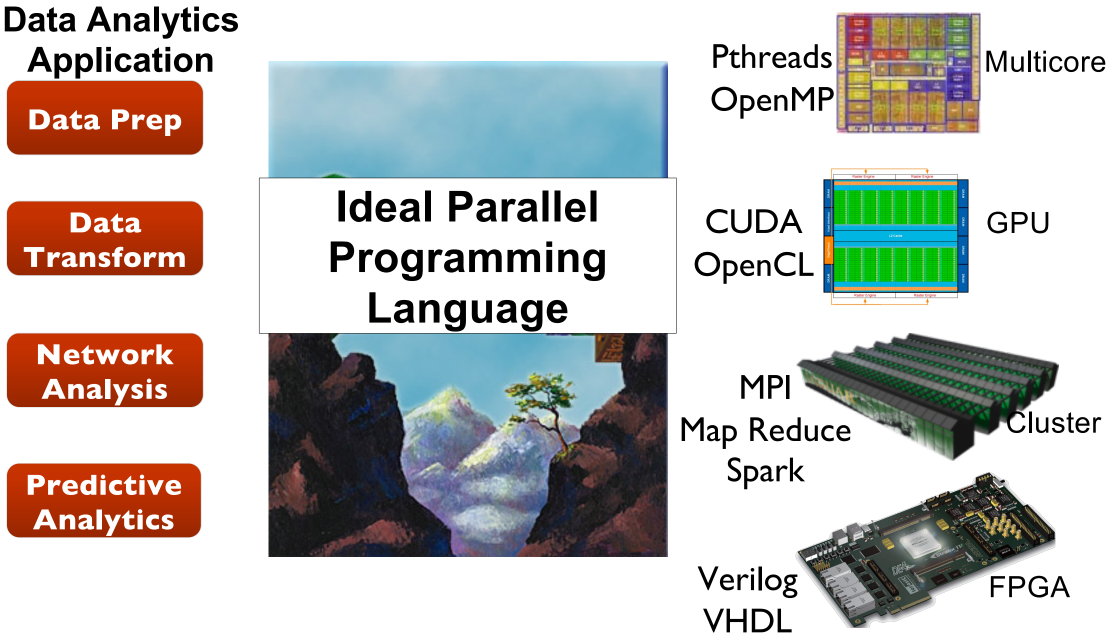


~3 orders of magnitude

Optimizations:

- **Precomputing twiddle**
- **Not computing what not part of the filter**
- **Transposing the matrix**
- **Using SSE**

Big-Data Analytics Programming Challenge



Data Analytics Application

Data Prep

Data Transform

Network Analysis

Predictive Analytics

Ideal Parallel Programming Language

Pthreads
OpenMP

Multicore

CUDA
OpenCL

GPU

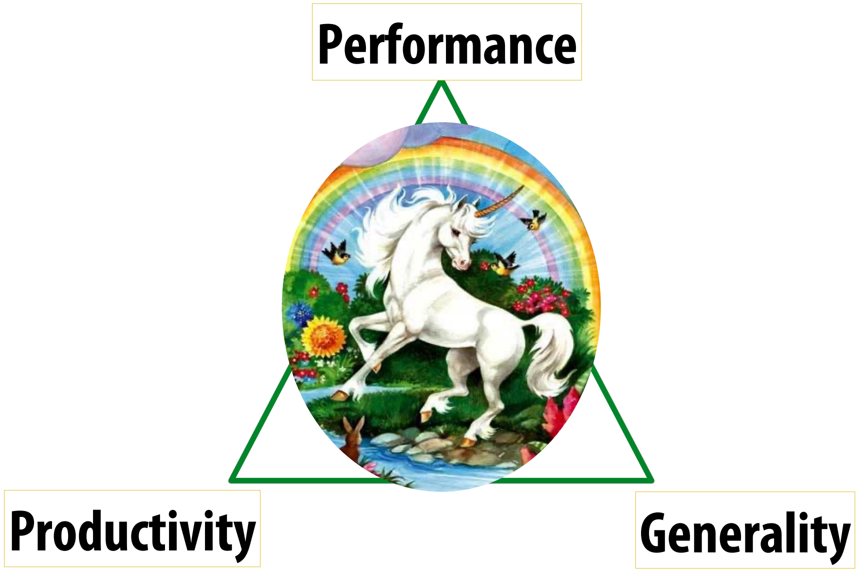
MPI
Map Reduce
Spark

Cluster

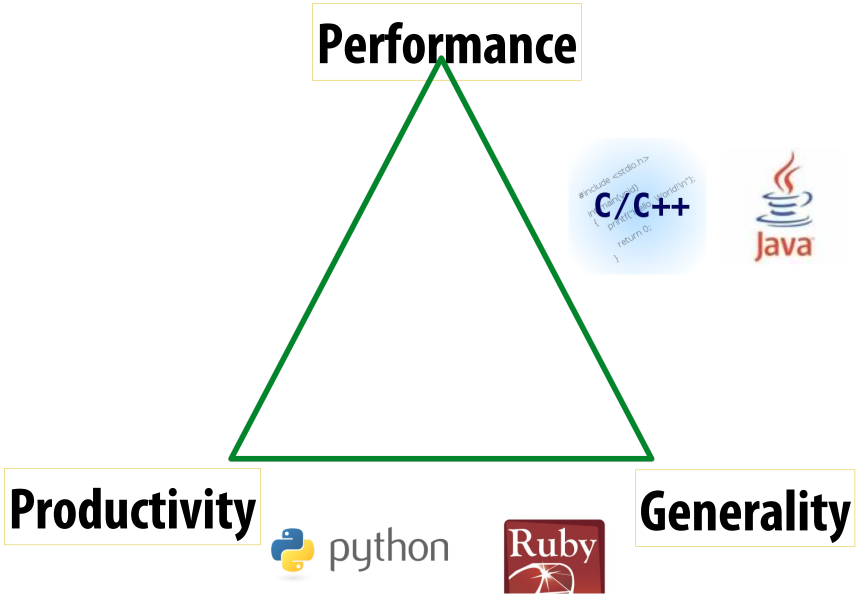
Verilog
VHDL

FPGA

The Ideal Parallel Programming Language



Successful Languages (not exhaustive ;-))

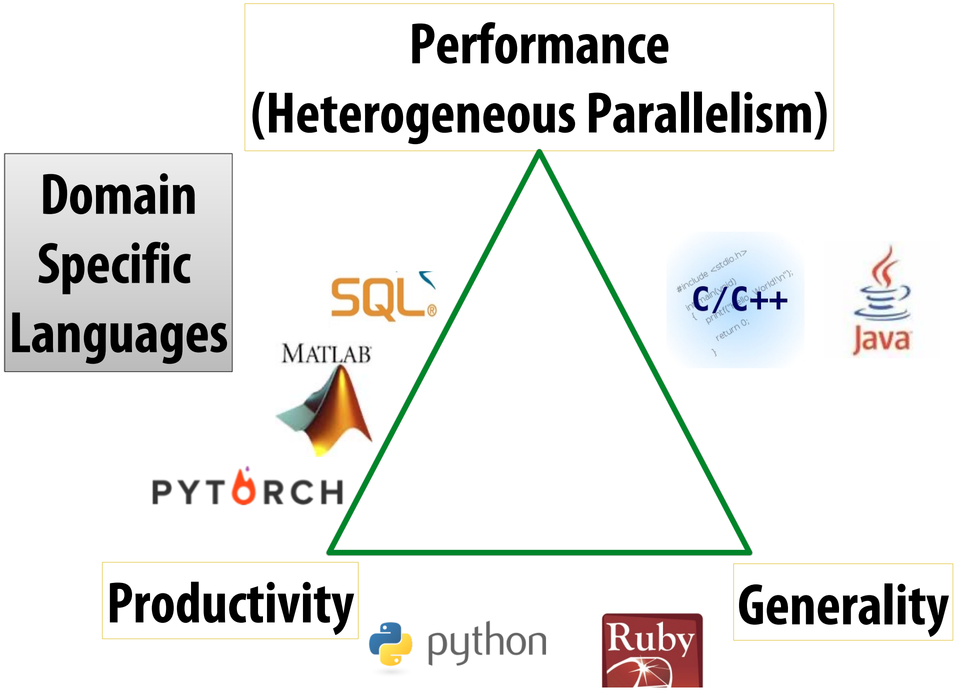


Performance

Productivity

Generality

Way Forward ⇒ Domain Specific Languages



Performance
(Heterogeneous Parallelism)

Domain Specific Languages

Productivity

Generality

DSL Hypothesis

It is possible to write one program
and
run it efficiently on heterogeneous
parallel systems

Domain Specific Languages

- Domain Specific Languages (DSLs)
  - Programming language with restricted expressiveness for a particular domain
  - High-level, usually declarative, and deterministic

Domain-specific programming systems

- Main idea: raise level of abstraction for expressing programs
    - Goal: write one program, and run it efficiently on different machines
- Introduce high-level programming primitives specific to an application domain
    - Productive: intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in targeted domain
    - Performant: system uses domain knowledge to provide efficient, optimized implementation(s)
        - Given a machine: system knows what algorithms to use, parallelization strategies to employ for this domain
        - Optimization goes beyond efficient mapping of software to hardware! The hardware platform itself can be optimized to the abstractions as well
- Cost: loss of generality/completeness

Building DSLs

- External DSL
  - An external DSL is implemented as a standalone language
    - Matlab, SQL
  - + Flexible syntax and semantics, simplicity
  - – YAPL, interpretations slow, no tool chain (IDE, debugger, prof, …

- Embedded (Internal) DSL
  - An internal DSL is embedded within another language. Ideally, the host language has features that make it easy to build DSLs
  - OptiML(Scala), Halide(C++), PyTorch(Python), TensorFlow(Python)
  - +Familiar syntax, access to general purpose features, tool chain
  - –verbose syntax, hard to limit features, hard to debug DSLs, slow interpreters

Delite: A Framework for High Performance DSLs

- Overall Approach: Generative Programming for “Abstraction without regret”
  - Embed compilers in Scala libraries: Scala does syntax and type checking
  - Use metaprogramming with LMS (type-directed staging) to build an intermediate representation (IR) of the user program
  - Optimize IR and map to multiple targets

- Goal: Make embedded DSL compilers easier to develop than stand alone DSLs
  - As easy as developing a library

OptiML: Overview

- Provides a familiar (MATLAB-like) language and API for writing ML applications
  - Ex. `val c = a * b` (a, b are Matrix[Double])

- Implicitly parallel data structures
  - Base types
    - Vector[T], Matrix[T], Graph[V,E], Stream[T]
  - Subtypes
    - TrainingSet, IndexVector, Image, …

- Implicitly parallel control structures
  - sum{…}, (0::end) {…}, gradient { … }, untilconverged { … }
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures

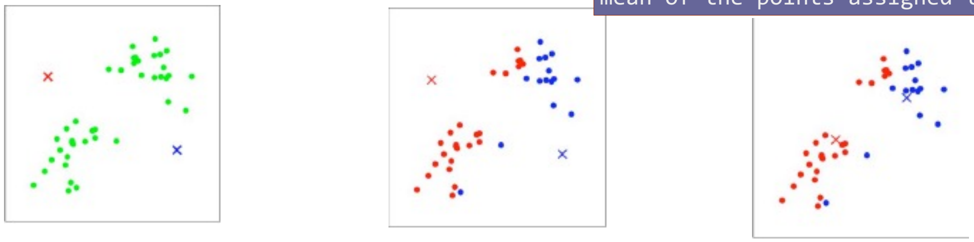
K-means Clustering in OptiML

```
untilconverged(kMeans, tol){kMeans =>
  val clusters = samples.groupRowsBy { sample =>
      kMeans.mapRows(mean => dist(sample, mean)).minIndex
  }
  val newKmeans = clusters.map(e => e.sum         gth)
  newKmeans
}
```

assign each sample to the closest mean

calculate distances to current means

move each cluster centroid to the mean of the points assigned to it



- No explicit map-reduce, no key-value pairs
- No distributed data structures (e.g. RDDs)
- No annotations for hardware design
- Efficient multicore and GPU execution
- Efficient cluster implementation
- Efficient FPGA hardware

Importance of DSLs is Growing

- Linear Algebra
  - Matlab
- Data processing
  - SQL
- Machine Learning
  - PyTorch
  - TensorFlow
- Image processing
  - Halide

- Key question
  - How to design and implement high-performance DSLs that support heterogeneous computing

A DSL example:

Halide: a domain-specific language for image processing

Jonathan Ragan-Kelley, Andrew Adams et al.
[SIGGRAPH 2012, PLDI 13]