

TÓPICOS ESPECIAIS – SISTEMAS EMBARCADOS *PROGRAMAÇÃO PARALELA E TEMPO REAL*

PROF. JOSENALDE OLIVEIRA

josenalde.oliveira@ufrn.br

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - UFRN

PLANO DE CURSO

OBJETIVOS

O(a) discente compreenderá os aspectos de arquitetura paralela de computadores e sistemas embarcados, de modo a projetar e implementar soluções em software que explorem o paralelismo de tarefas e gerência do sistema operacional de tempo real embarcado nestes sistemas.

CONTEÚDO

Arquitetura; Multiprocessamento simétrico e assimétrico; Threads (POSIX) e biblioteca OpenMP; controle de sincronismo (multitarefa, multinúcleos): semáforos, filas, locks, exclusão mútua, temporizadores; Definição e classificação de sistemas tempo-real/críticos; Sistemas operacionais para tempo-real (RTOS e FreeRTOS): funcionalidades básicas, executivo de tempo-real, escalonamento e gerenciamento de tarefas e mensagens de tempo-real; Projeto, modelagem e desenvolvimento embarcado de tempo real; noções de controladores digitais e automação de processos.

PLANO DE CURSO

- Referências

PACHECO, Peter S. An introduction to parallel programming. Morgan Kaufmann, 2011. (**Caps 1,2,4,5**)

LIN, Calvin.; SNYDER, Lawrence: Principles of parallel programming. Pearson, 2008. (**Caps 1, 3, 6**)

Referência oficial OpenMP



No Linux, última versão acompanha o compilador GCC/G++: `git clone git://gcc.gnu.org/git/gcc.git`

Julho 2023: 13.2; OBS: Desde GCC 9, há suporte inicial para OpenMP 5 (apenas para C/C++)

Sobre POSIX threads (pthreads): <https://pubs.opengroup.org/onlinepubs/9699919799/>

<https://hpc-tutorials.llnl.gov posix/> (**também intro e OpenMP**)

Coleção ESP32 do Embarcados: Explore o FREERTOS com o ESP32, parte 2. Disponível em:

<https://embarcados.com.br/e-books/e-book-colecao-esp32-parte-2/>

KOPETZ, Hermann. Real-time systems: design principles for distributed embedded applications, 2. ed.. New York: Springer, 2011.

LAPLANTE, Phillip A. Real-time systems design and analysis: tools for the practitioner, 4th ed. New Jersey: Wiley, 2012.

SHAW, Alan C. Sistemas e software de tempo real. Porto Alegre: Bookman, 2003.

KUO, Benjamin C. Automatic control systems, 8th ed. New York: Wiley, 2003.

PLANO DE CURSO

OBJETIVOS

O(a) discente compreenderá os aspectos de arquitetura paralela de computadores e sistemas embarcados, de modo a projetar e implementar soluções em software que explorem o paralelismo de tarefas e gerência do sistema operacional de tempo real embarcado nestes sistemas.

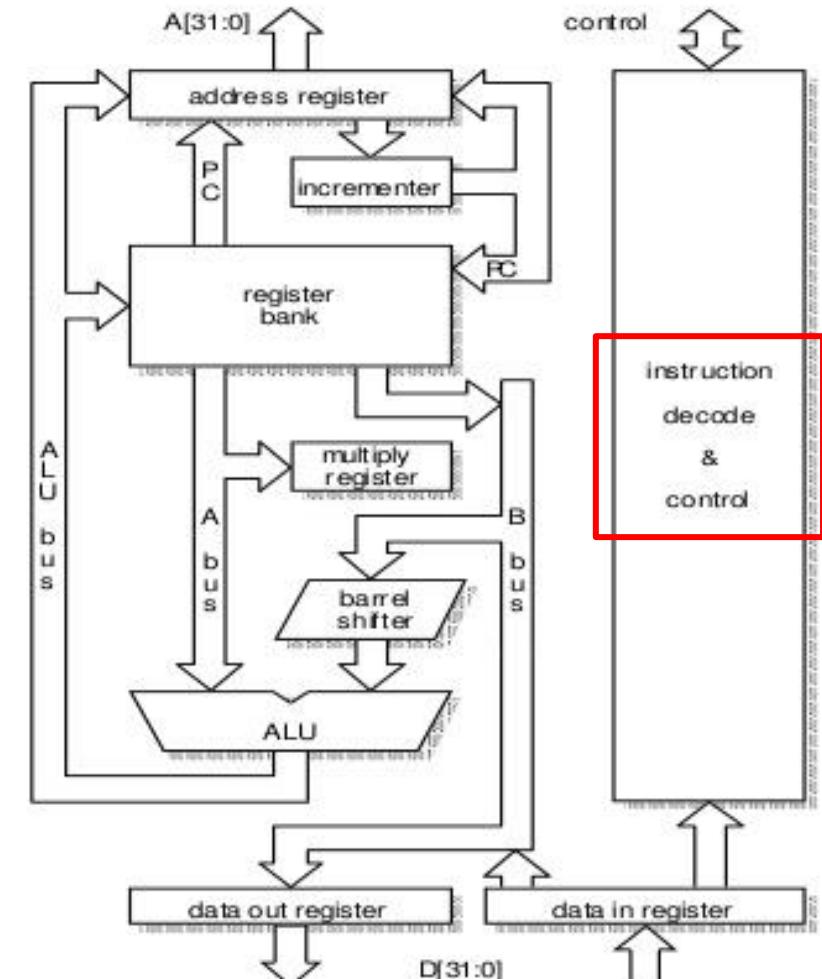
CONTEÚDO

Arquitetura; Multiprocessamento simétrico e assimétrico; Threads (POSIX) e biblioteca OpenMP; controle de sincronismo (multitarefa, multinúcleos): semáforos, filas, locks, exclusão mútua, temporizadores; Definição e classificação de sistemas tempo-real/críticos; Sistemas operacionais para tempo-real (RTOS e FreeRTOS): funcionalidades básicas, executivo de tempo-real, escalonamento e gerenciamento de tarefas e mensagens de tempo-real; Projeto, modelagem e desenvolvimento embarcado de tempo real; noções de controladores digitais e automação de processos.

Arquitetura ARM (ARM Holdings, Cambridge, Inglaterra, arm.com)

- Família de arquiteturas **RISC** para processadores
 - Set de instruções menor, instruções mais simples
 - Acesso simples à memória, menos ciclos para obter operandos
 - Set de registradores (register bank) para permitir **pipeline**, várias instruções executadas ao mesmo tempo
 - Programas maiores (assembler) ocupam mais memória
 - Licenciada para outros fabricantes, podem criar seus próprios Chips, mas usam esta arquitetura como base (Samsung, Apple, Nvidia, Qualcomm, Texas Instruments etc.)
 - **37 registradores** (30 de propósito geral)
 - Instruções ≥ 16 bits (set de instruções thumb)
 - Variantes (extensões):
 - NEON (áudio, vídeo), VFP (autotronica, gráfico, 3D, indústria)
 - DSP (sinais digitais), Jazelle (java), big.Little (múltiplos processadores) – no momento dynamIQ
 - TrustZone (proteção e segurança digital)

The ARM Architecture



Noções sobre multiprocessamento (múltipla execução simultânea)

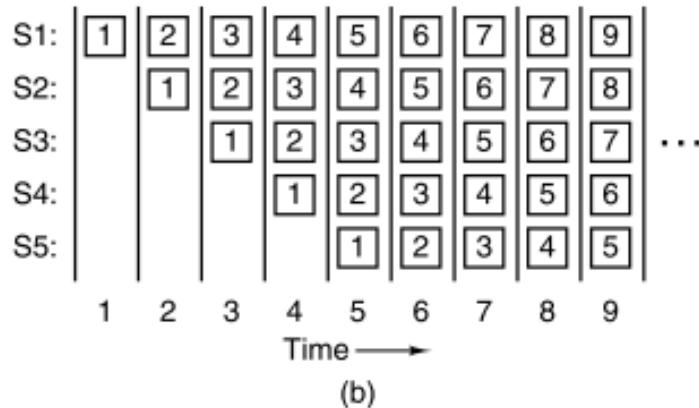
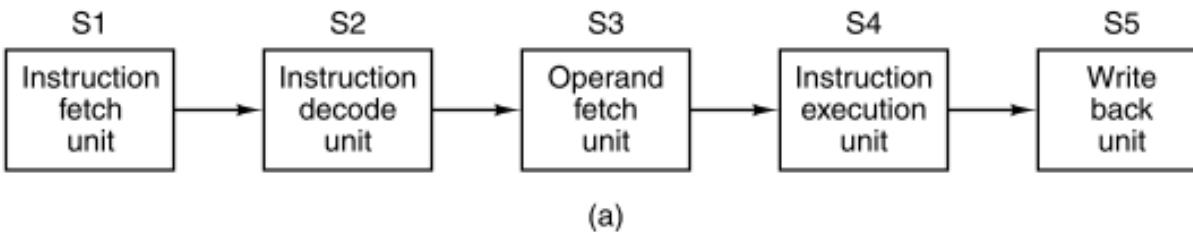


Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

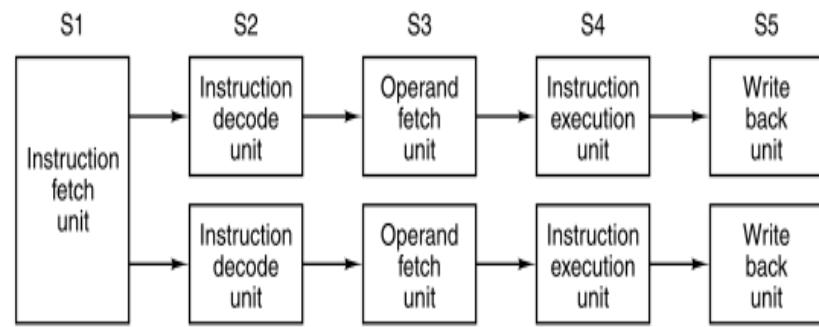
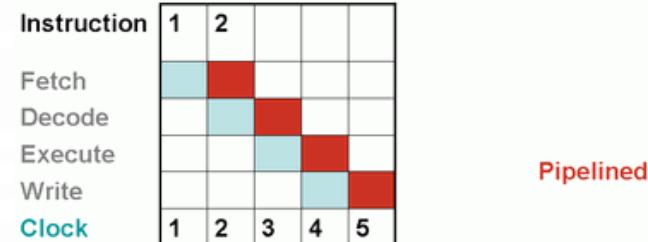
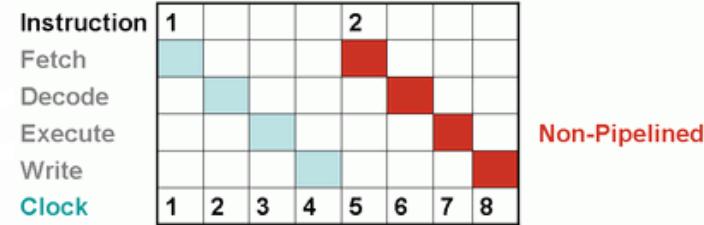
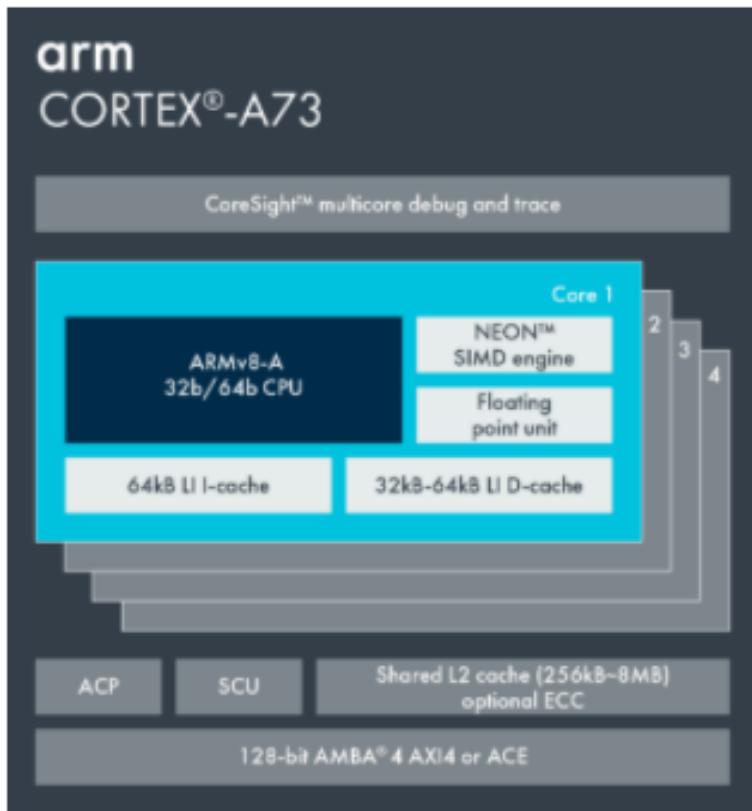


Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.



Pipeline em unidade individual de processamento
Superescalar: múltiplas unidades de processamento

Arquitetura ARM e desenvolvimento de software



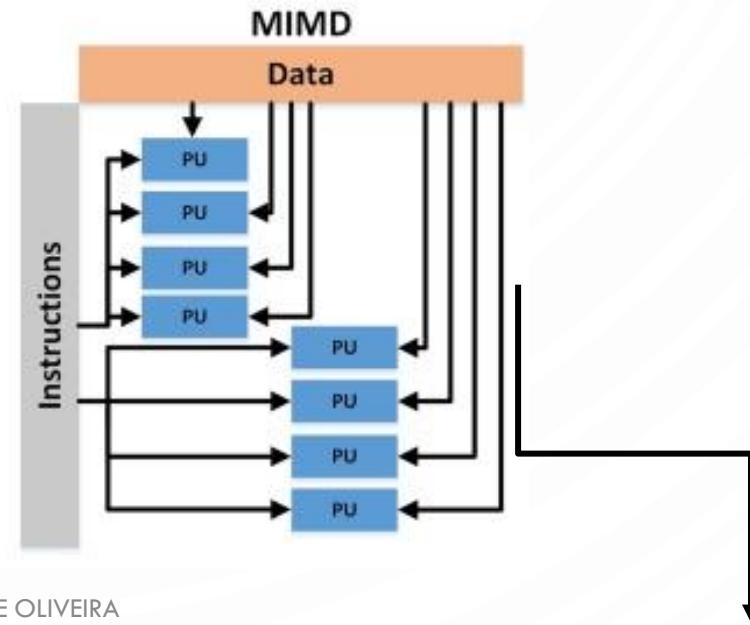
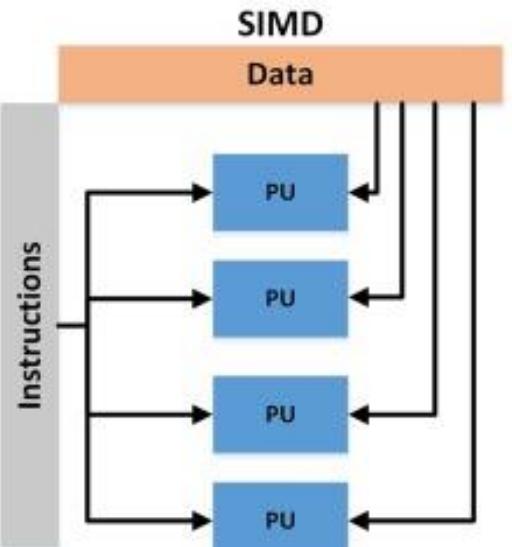
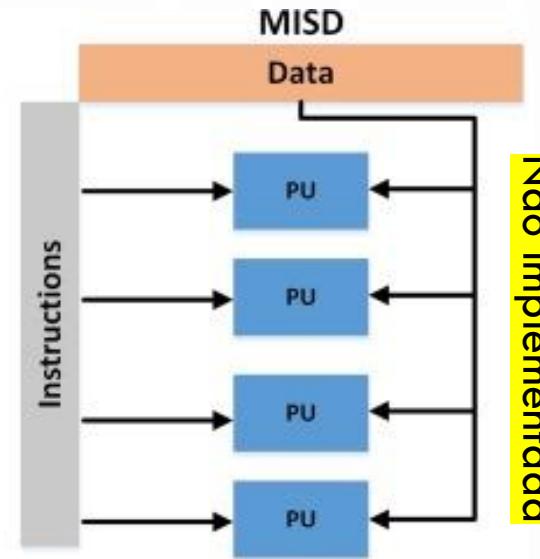
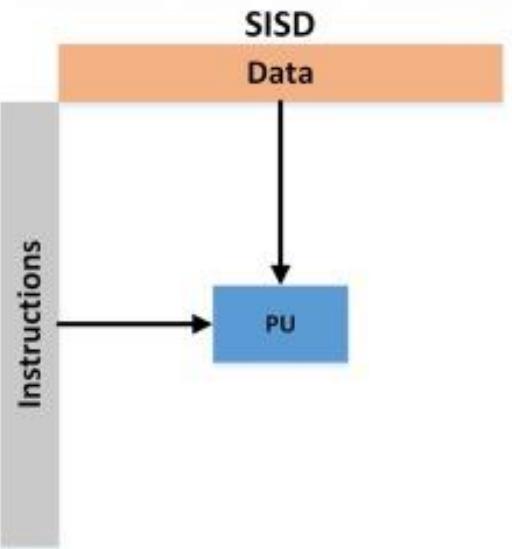
Arm Cortex-A73 CPU

Architecture	Armv8-A Desde 2011
Multicore	1-4x Symmetrical Multiprocessing (SMP) within a single processor cluster, and multiple coherent SMP processor clusters through AMBA 4 ACE technology <ul style="list-style-type: none">AArch32 for full backward compatibility with Armv7AArch64 for 64-bit support and new architectural features<u>TrustZone</u> security technology<u>Neon</u> advanced SIMDDSP & SIMD extensionsVFPv4 floating pointHardware virtualization support
ISA Support	
Debug & Trace	<u>CoreSight SoC-400</u>

<https://developer.arm.com/ip-products/processors/cortex-a/cortex-a73>

Em 2021 anunciou Armv9

Noções multiprocessamento (arquiteturas) – classificação de Flynn



```
import java.util.Arrays;
import java.util.List;

public class TestStream {
    public static void main(String[] args) {
        // stream sequencial
        List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4, 10, 20, 30, 40);
        listOfNumbers
            .stream() //stream API (Collections) desde 2014, Java 8
            .forEach(number ->
                System.out.println(number + " " + Thread.currentThread().getName())
            );
        System.out.println("Regiao Paralela: \n");
        // stream em paralelo
        listOfNumbers
            .parallelStream()
            .forEach(number ->
                System.out.println(number + " " + Thread.currentThread().getName())
            );
        // soma em paralelo e reduz para sum, adicionando 5 à soma total (0 valor inicial)
        int sum = listOfNumbers
            .parallelStream()
            .reduce(0, Integer::sum) + 5; // lambda: (subtotal, element) -> subtotal + element)
        System.out.println(sum);
    }
}
```

SISD

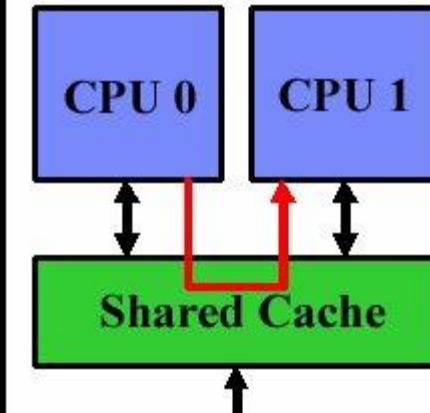
SIMD

• Noções multiprocessamento

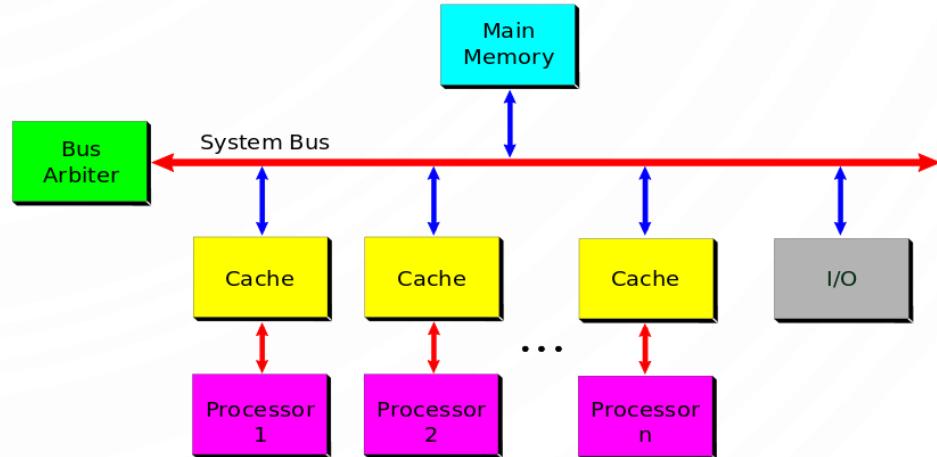
Processadores de uso geral

Solução 1: memória compartilhada: cada processador acessa programas e dados armazenados na memória compartilhada e os processadores se comunicam uns com os outros por meio dessa memória SMP – multiprocessador simétrico – múltiplos processadores compartilham uma única memória ou um pool de memória por um barramento compartilhado; o tempo de acesso é aproximadamente o mesmo para cada processador (**UMA**), embora na arquitetura variante **NUMA** o tempo de acesso possa ser diferenciado (não uniforme).

Solução 2: memória distribuída:

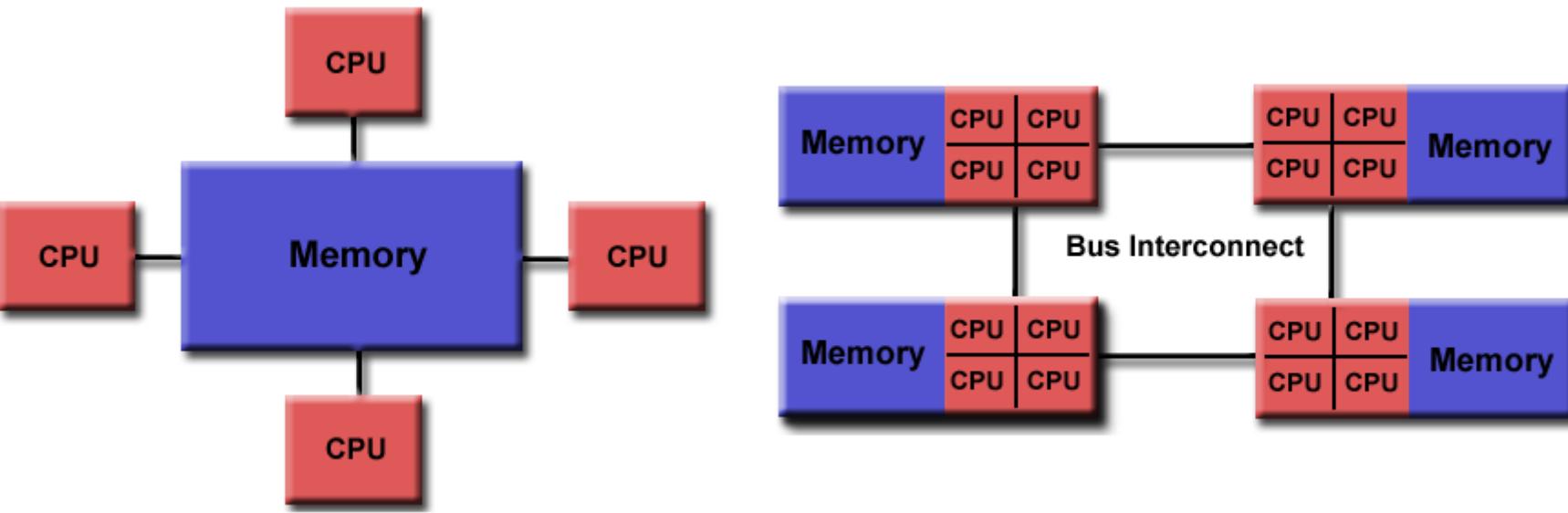


SMP - Symmetric Multiprocessor System



By Ferruccio Zulian - Milan, Italy

- Noções multiprocessamento com memória compartilhada



Arquitetura ARM e desenvolvimento de software

Cortex-A15

High-performance with infrastructure feature set

Cortex-A17

High-performance with lower power and smaller area relative to Cortex-A15

Cortex-A57

Proven high-performance

64/32-bit

Cortex-A72

2016 Premium Mobile, Infrastructure & Auto

64/32-bit

Cortex-A73

2017 Premium Mobile, Consumer

64/32-bit

High Performance

Cortex-A8

First ARMv7-A processor

Cortex-A9

Well established mid-range processor used in many markets

Cortex-A53

Balanced performance and efficiency

64/32-bit

High Efficiency

Cortex-A5

Smallest and lowest power ARMv7-A CPU, optimized for single-core

Cortex-A7

Most efficient ARMv7-A CPU, higher performance than Cortex-A5

Cortex-A32

Smallest and lowest power ARMv8-A

32-bit

Cortex-A35

Highest efficiency

64/32-bit

Ultra High Efficiency

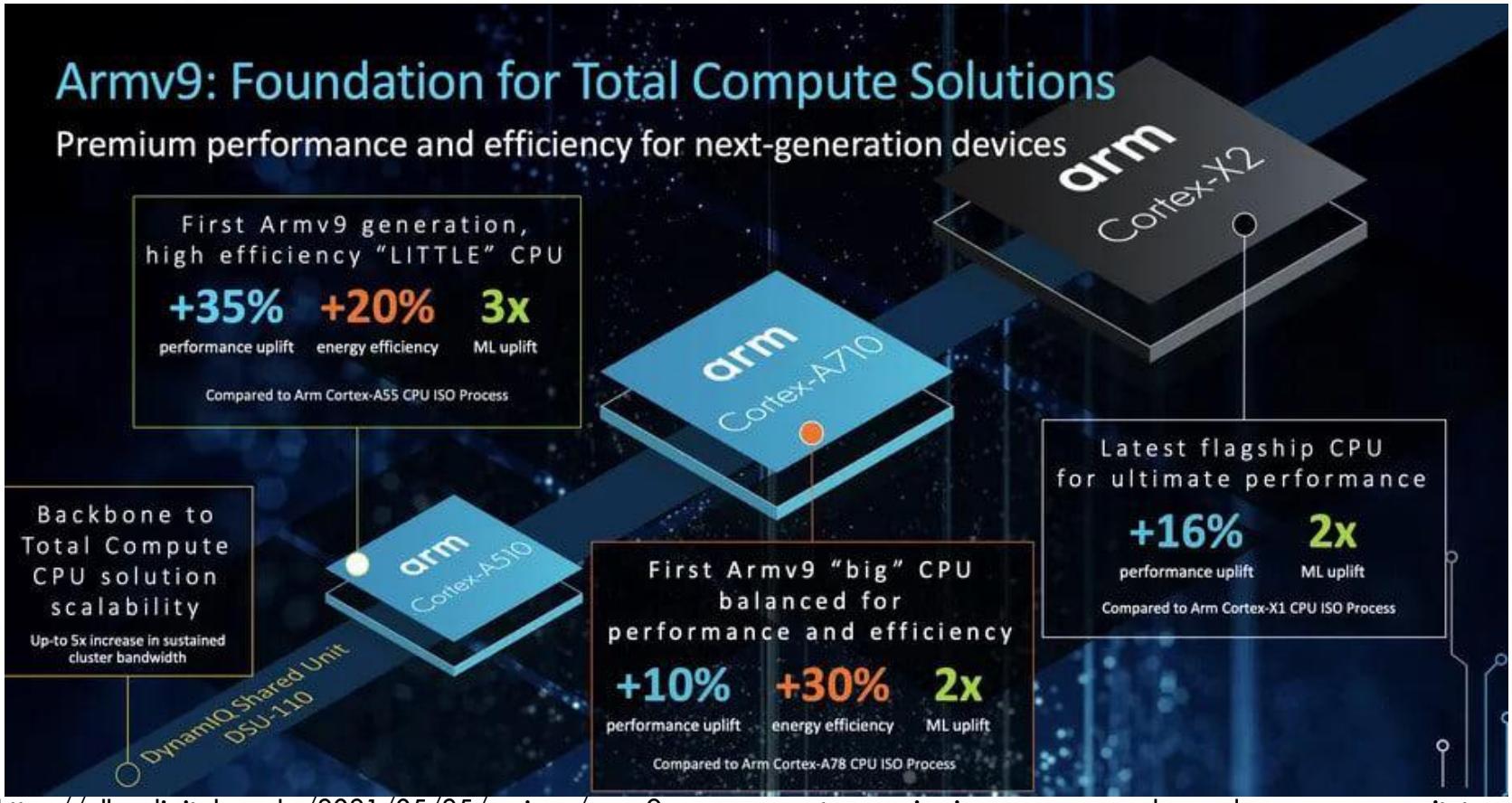
ARMv7-A

ARMv8-A

Key:  big.LITTLE compatible

A ideia do big.LITTLE

há uma combinação de um núcleo mais poderoso para tarefas que exigem “força bruta”, com núcleos menores e mais eficientes para tarefas mais leves.



Fonte: <https://olhardigital.com.br/2021/05/25/reviews/armv9-arm-apresenta-as-primeiras-cpus-e-gpus-baseadas-na-nova-arquitetura/>

Exemplo: Snapdragon 888
o conjunto é formado por:

- um Cortex-X1,
- três Cortex-A78 (big)
- quatro Cortex-A55 (LITTLE),
- GPU Adreno

GPUs com foco em ML (machine learning)



Fonte: <https://olhardigital.com.br/2021/05/25/reviews/armv9-arm-apresenta-as-primeiras-cpus-e-gpus-baseadas-na-nova-arquitetura/>

Uma rede neural é essencialmente paralela!

AI everywhere demands specialized, scalable solutions

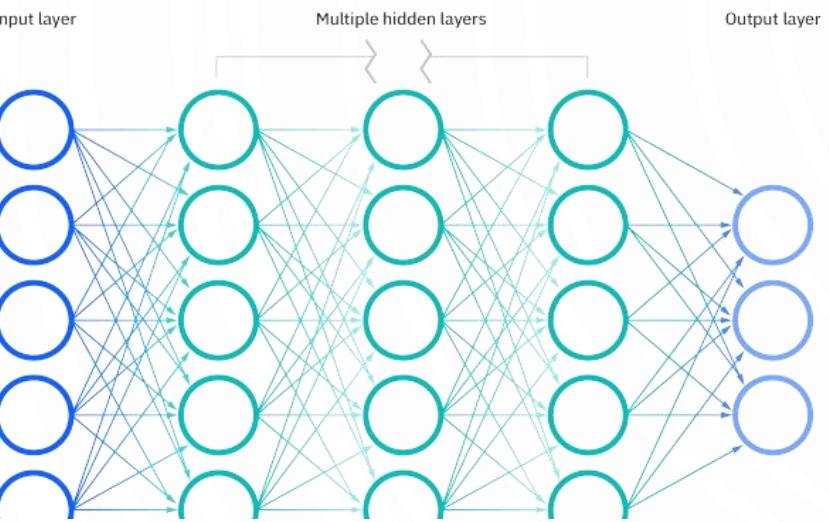
The ubiquity and range of AI workloads demands more diverse and specialized solutions. For example, it is estimated there will be more than eight billion AI-enabled voice-assisted devices in use by the mid-2020sⁱ, and 90 percent or more of on-device applications will contain AI elements along with AI-based interfaces like vision or voiceⁱⁱ.

To address this need, Arm partnered with Fujitsu to create the Scalable Vector Extension (SVE) technology, which is at the heart of Fugaku, the world's fastest supercomputer. Building on that work, Arm has developed SVE2 for Armv9 to enable enhanced machine learning (ML) and digital signal processing (DSP) capabilities across a wider range of applications.

SVE2 enhances the processing ability of 5G systems, virtual and augmented reality, and ML workloads running locally on CPUs, such as image processing and smart home applications. Over the next few years, Arm will further extend the AI capabilities of its technology with substantial enhancements in matrix multiplication within the CPU, in addition to ongoing AI innovations in its Mali™ GPUs and Ethos™ NPUs.

<https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture>

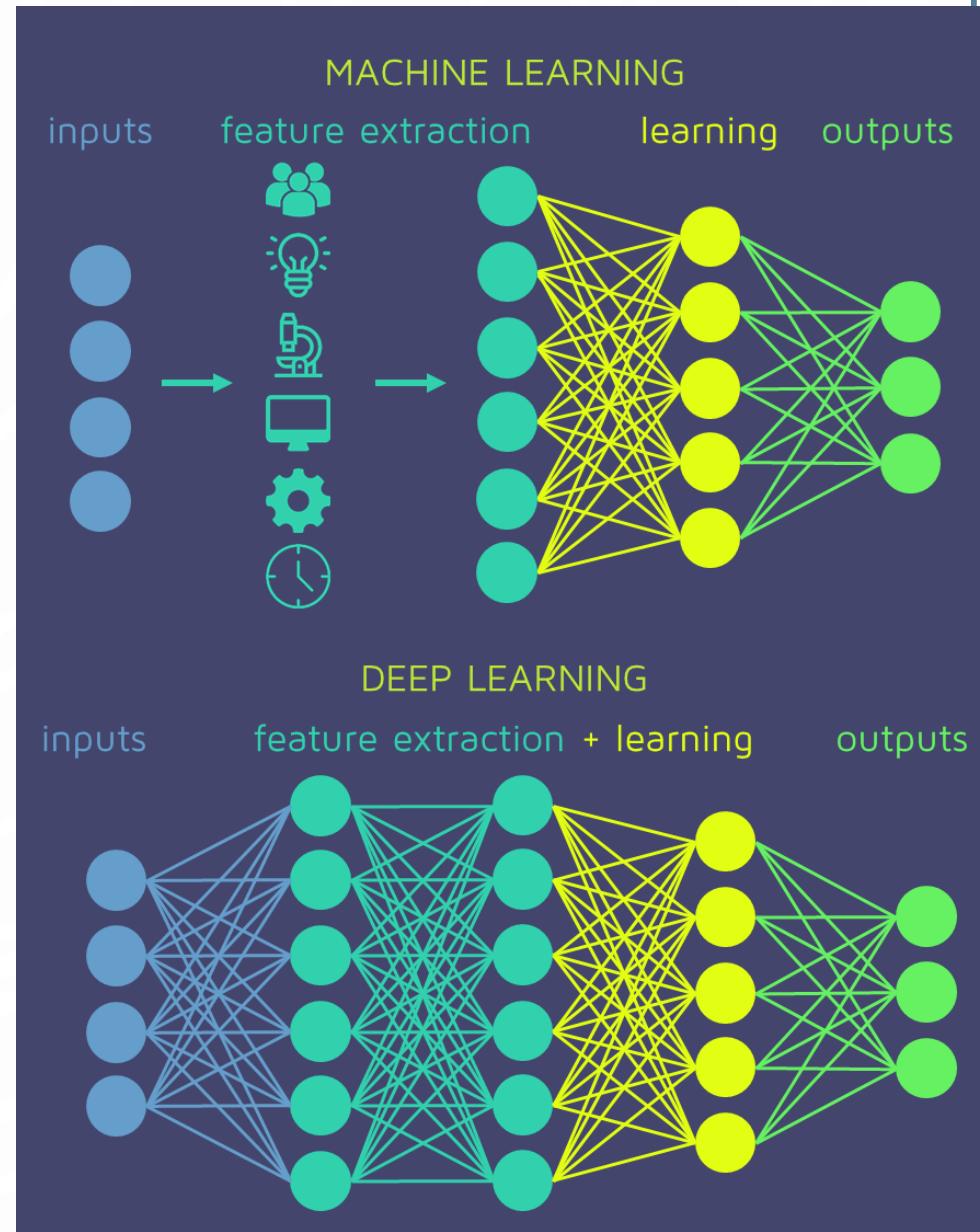
Uma rede neural é essencialmente paralela!



Para saber mais:

- Redes Neurais Convolucionais (CNN): visão
- Redes Neurais Recorrentes/Transformers: PLN
- tinyML

<https://conect2ai.dca.ufrn.br/>



Uma rede neural é essencialmente paralela!

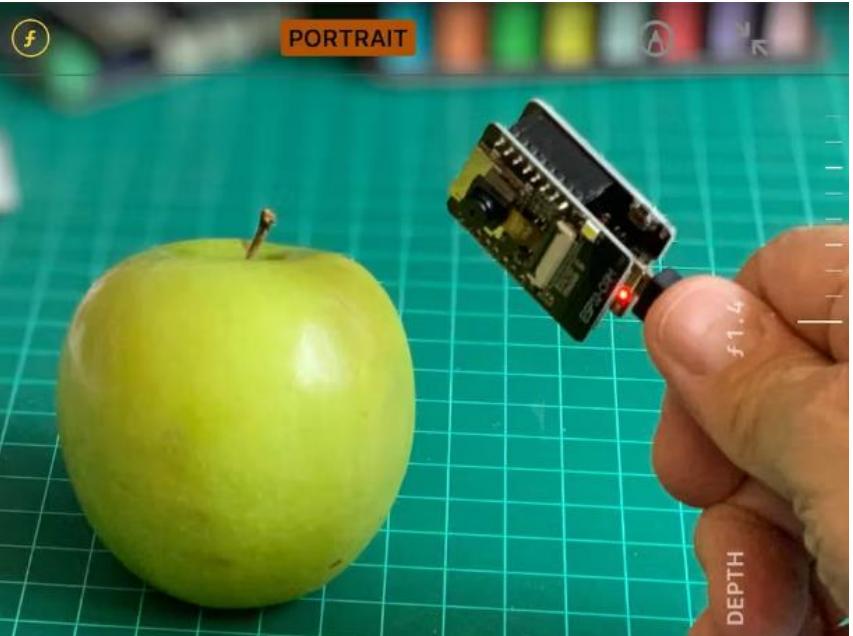
Como funciona uma Alexa Skill

Uma skill de Alexa tem tanto um modelo de interação - ou interface de usuário de voz - como uma lógica de aplicativo. Quando um cliente fala, a Alexa processa a fala no contexto do seu modelo de interação para determinar qual foi a solicitação do cliente. A Alexa então envia a solicitação à sua lógica de aplicação de skills, que processa essa informação. Numa skill, você fornece a lógica do aplicativo como um serviço de nuvem de back-end hospedado pela Alexa, AWS ou outro servidor.



<https://developer.amazon.com/pt-BR/alexza/alexza-skills-kit/start>

Mas e com processadores mais “tímidos”?



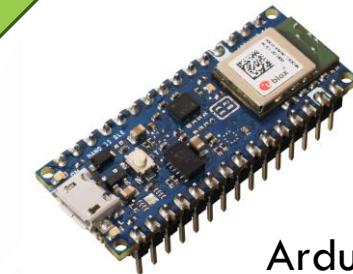
EDGE IMPULSE

TensorFlow
Lite

Treino dos modelos



Deploy



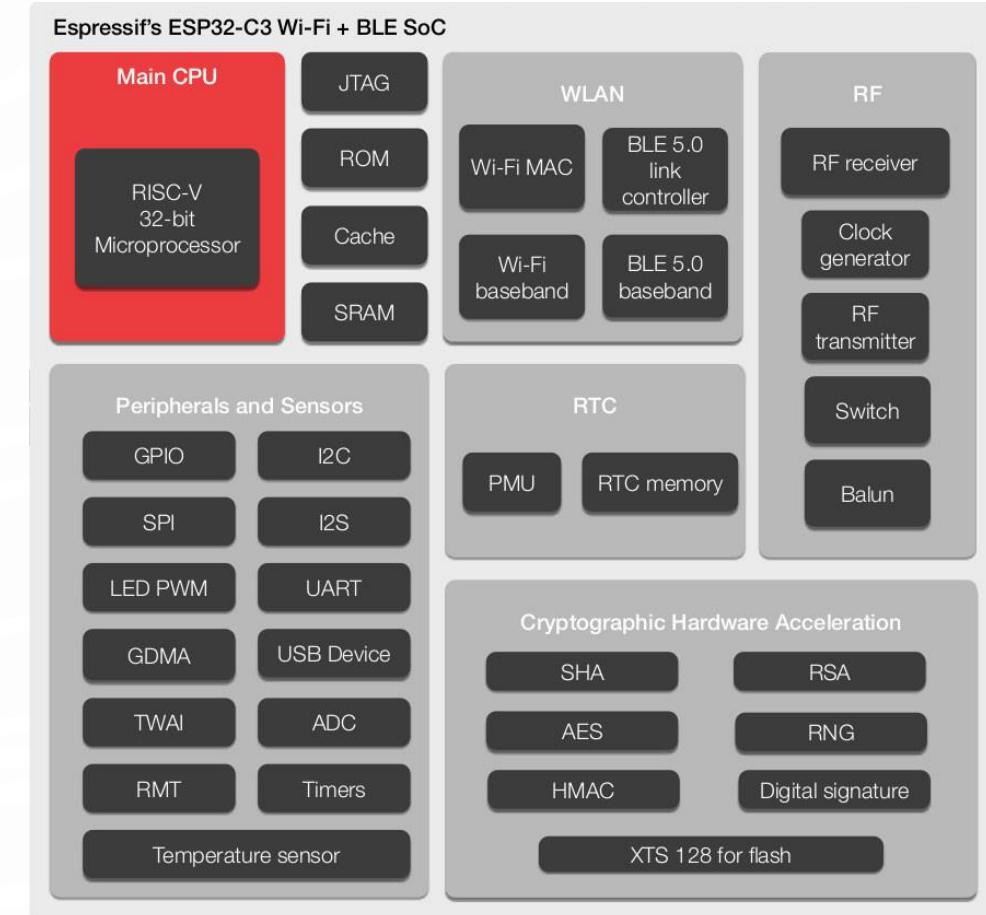
Arduino BLE Nano 33

<https://www.hackster.io/mjrobot/esp32-cam-tinyml-image-classification-fruits-vs-veggies-4ab970>

Mas e com processadores mais “tímidos”?

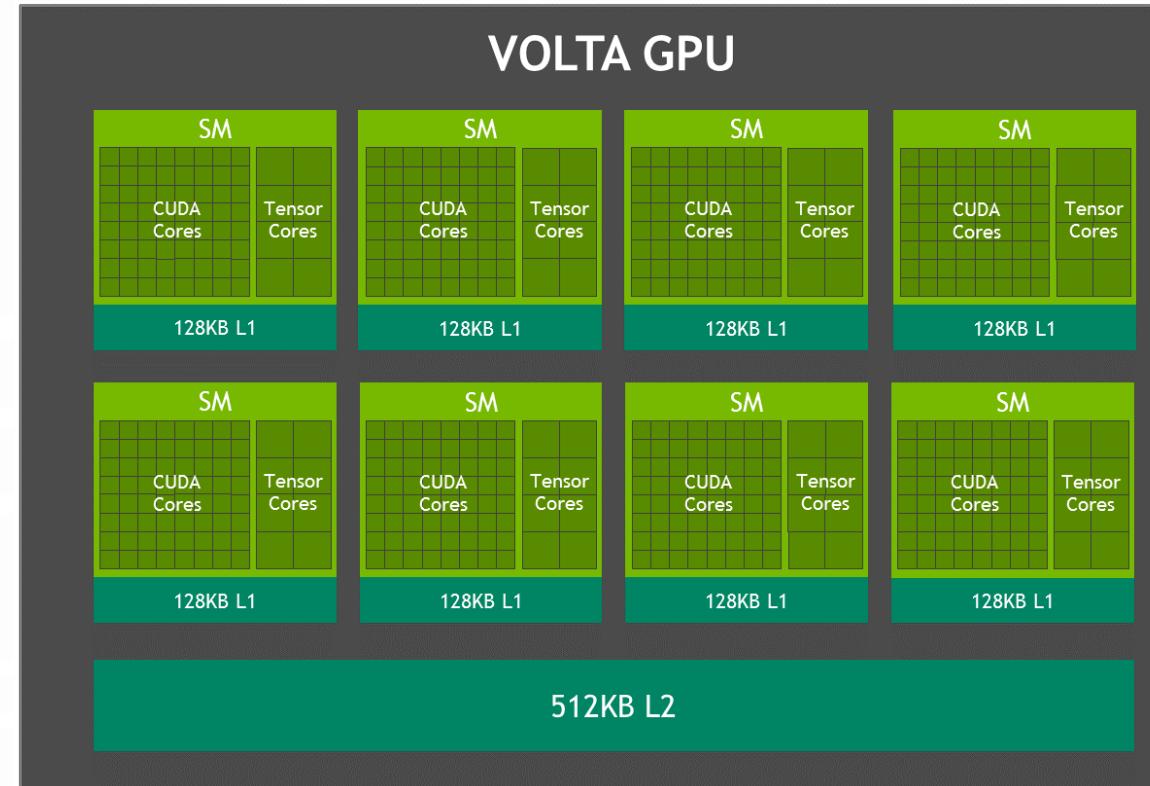
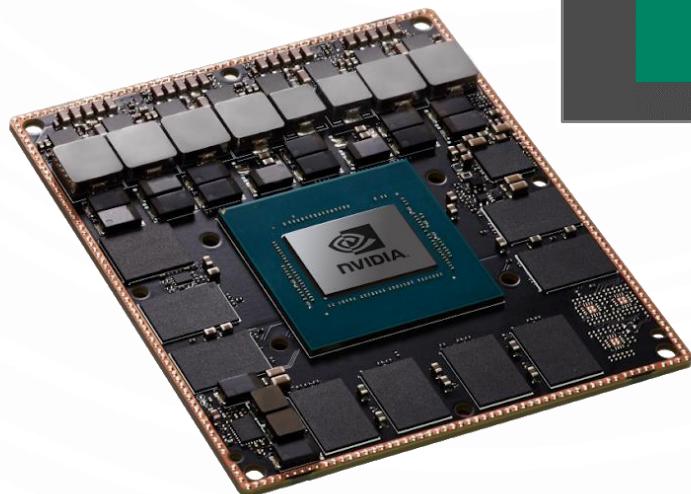
Specifications	ESP32
MCU	Xtensa Dual-Core 32-bit LX6 600 DMIPS
802.11 b/g/n Wi-Fi	Yes, HT40
Bluetooth	Bluetooth 4.2 and below
Typical Frequency	160 MHz
SRAM	512 kBytes
Flash	SPI
GPIO	36
Hardware / Software PWM	1 / 16 Channels
SPI / I2C / I2S / UART	4/2/2/2
ADC	12-bit
CAN	1
Ethernet MAC Interface	1
Touch Sensor	Yes
Temperature Sensor	Yes
Working Temperature	-40° C – 125° C

- Flash Memory: 4 MB (8/16) – módulos externos...
- SRAM: 520 KB
- Clock Speed: 240 Mhz





Plataformas com foco em IA/ML (nvidia)



Versão Orin Nano: US\$ 560

Plataformas com foco em IA/ML (nvidia)

Stencil computations on GPU

Using the `numba.cuda` module I'm able to get about a 200x increase with a modest increase in code complexity.

In [3]:

```
from numba import cuda

@cuda.jit
def smooth_gpu(x, out):
    i, j = cuda.grid(2)
    n, m = x.shape
    if 1 <= i < n - 1 and 1 <= j < m - 1:
        out[i, j] = (x[i - 1, j - 1] + x[i - 1, j] + x[i - 1, j + 1] +
                      x[i      , j - 1] + x[i      , j] + x[i      , j + 1] +
                      x[i + 1, j - 1] + x[i + 1, j] + x[i + 1, j + 1]) // 9
```

In [4]:

```
import cupy, math

x_gpu = cupy.ones((10000, 10000), dtype='int8')
out_gpu = cupy.zeros((10000, 10000), dtype='int8')

# I copied the four lines below from the Numba docs
threadsperblock = (16, 16)
blockspergrid_x = math.ceil(x_gpu.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(x_gpu.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

%timeit smooth_gpu[blockspergrid, threadsperblock](x_gpu, out_gpu)
```

2.87 ms ± 90.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)



Arquitetura ARM e System on a Chip (SoC)

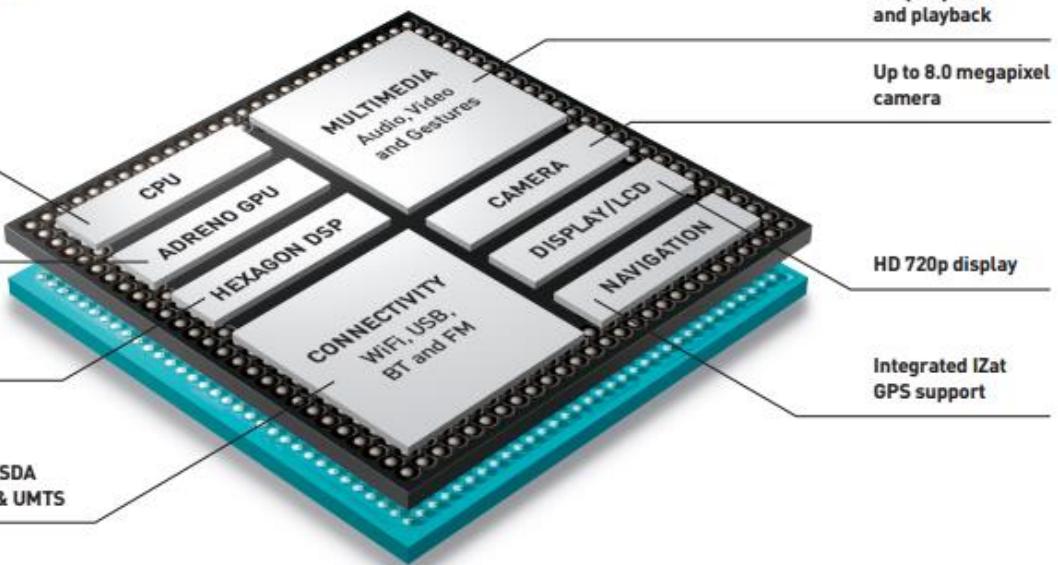
200 PROCESSOR

ARM Cortex A5 with Quad CPUs up to 1.4GHz for high performance

Adreno 203 for advanced graphics

Hexagon QDSP5 for ultra low power applications

Integrated 802.11n, BT3.0, USB 2.0 Multi-SIM DSDS, DSDA Single Platform for CDMA & UMTS



Cortex A73/A53



- Integração de componentes num único chip (subsistemas) – CPU, memória, E/S, tratamento de radiofrequência, gerenciamento de bateria, sensores
- Dispositivos menores, com menor consumo de energia
- Ideal para computação móvel, embarcada (ou embutida, **embedded systems**)
- Smartphones, tablets, câmeras digitais, e-book reader, consoles, gps, microcontroladores

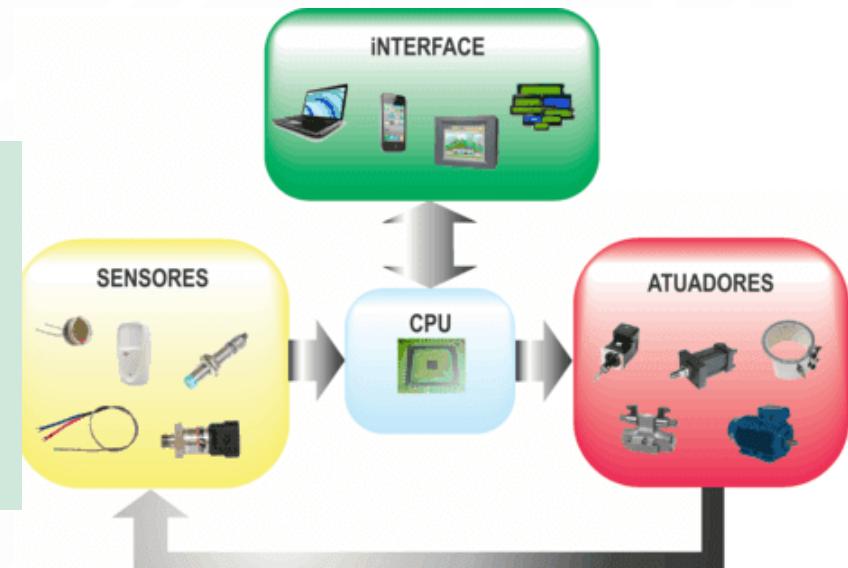


Sistemas Embarcados

- Eletrônica e software dentro de um produto, ao contrário de um computador de uso geral
- Combinação de hardware e software, com a possibilidade de integração de partes mecânicas e outras, projetada para realizar uma função dedicada; Podem fazer parte de um sistema ou produto maior, como exemplo o sistema de freios ABS de um carro
- Requisitos e restrições variáveis (segurança, confiabilidade, tempo real, flexibilidade, suporte à radiação, vibrações, umidade, tempo de resposta, precisão, etc).

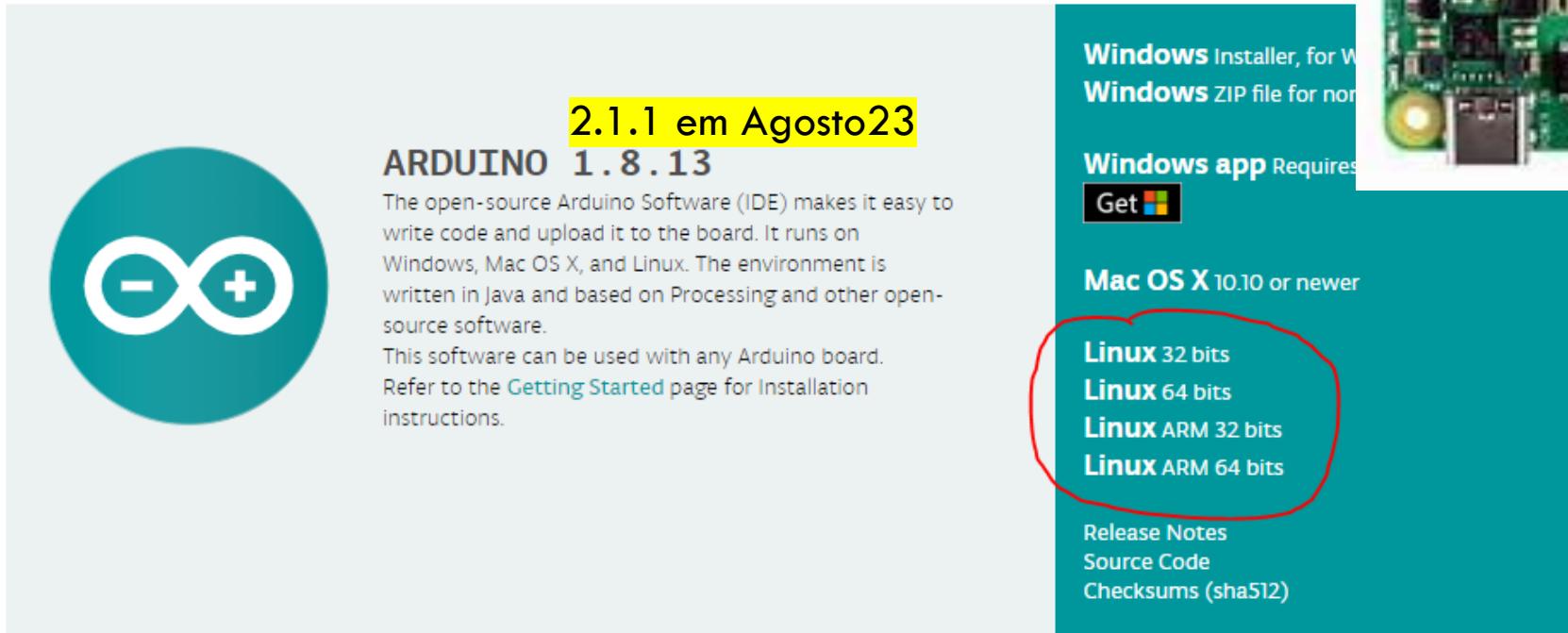
CATEGORIAS

- 1) **TEMPO REAL/CRÍTICOS:** aplicações de armazenamento, automotivas, industriais, redes etc. (Free RTOS etc.)
- 2) **PLATAFORMAS:** dispositivos com SO abertos, como Linux, Android, Chrome etc.
- 3) **SEGURANÇA:** smart cards, placas SIM, terminais de pagamento



Arquitetura ARM e desenvolvimento de software

Download the Arduino IDE



2.1.1 em Agosto23

ARDUINO 1.8.13

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software.

This software can be used with any Arduino board. Refer to the [Getting Started](#) page for installation instructions.

Windows Installer, for Windows
Windows ZIP file for non-Windows
Windows app Requires Windows 10 or newer
[Get Microsoft Store](#)

Mac OS X 10.10 or newer

Linux 32 bits
Linux 64 bits
Linux ARM 32 bits
Linux ARM 64 bits

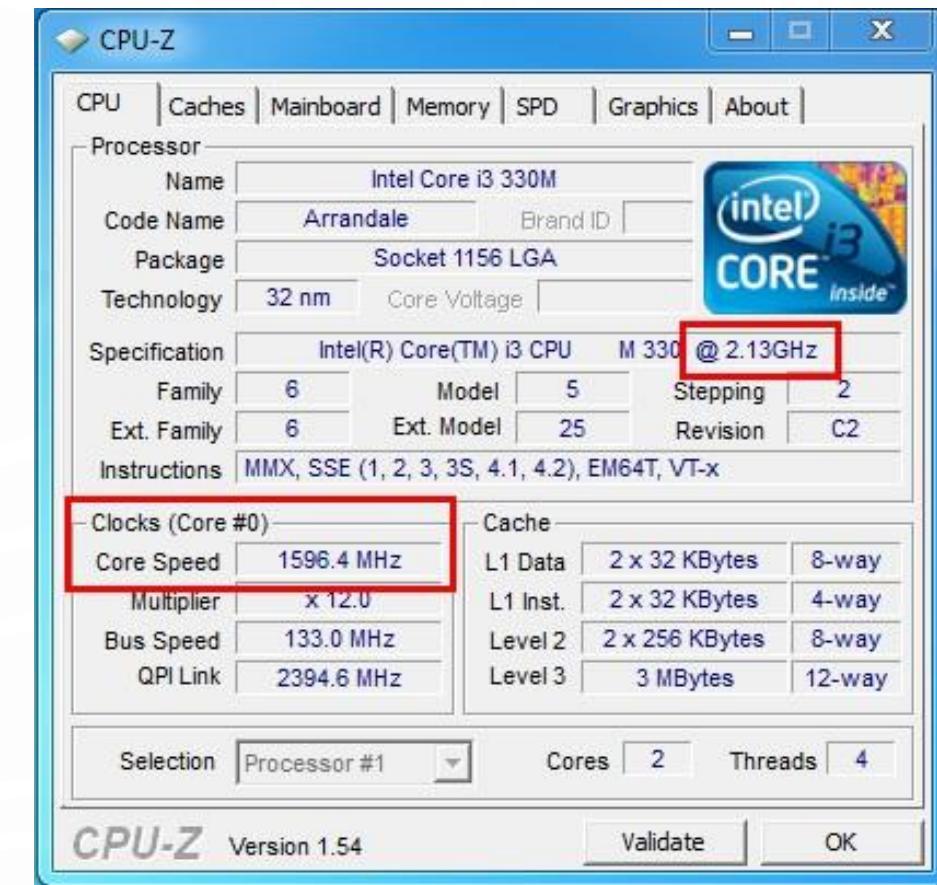
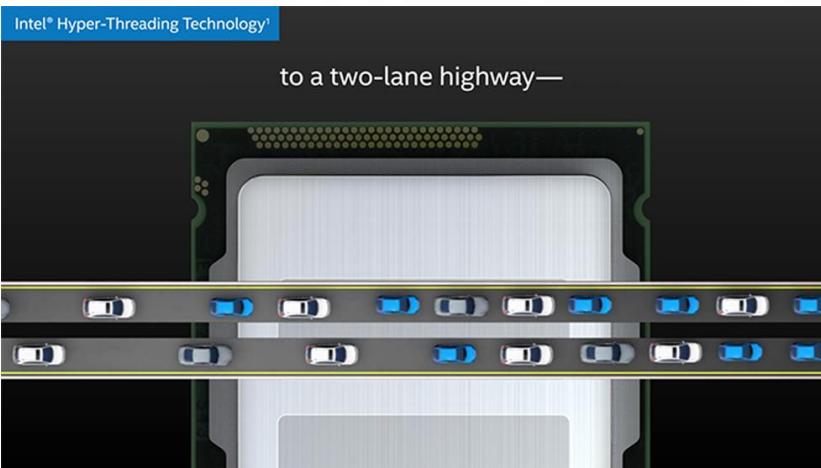
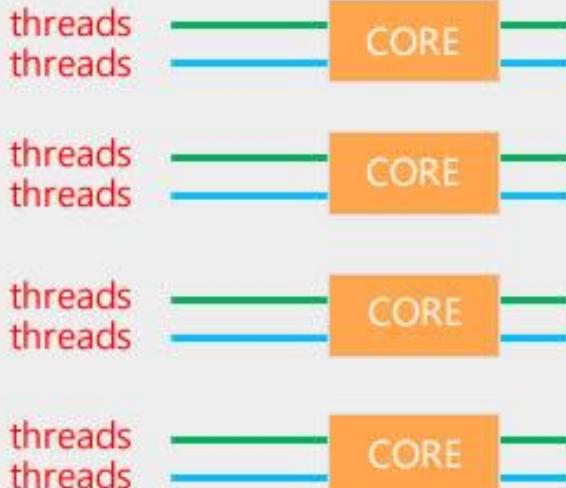
[Release Notes](#)
[Source Code](#)
[Checksums \(sha512\)](#)



Raspberry Pi 4 Model B
Broadcom 2711
Quad-core Cortex-A72
(ARMv8-A) 64-bit SoC
@ 1.5 GHz. X 4 threads

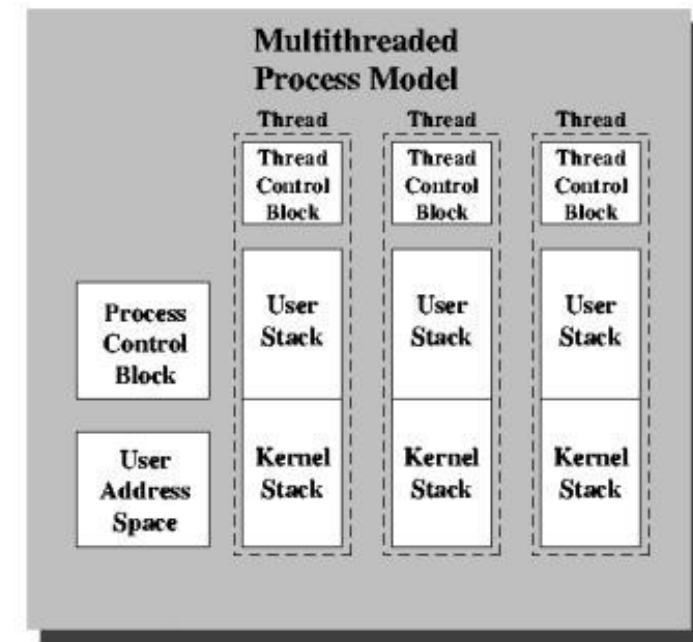
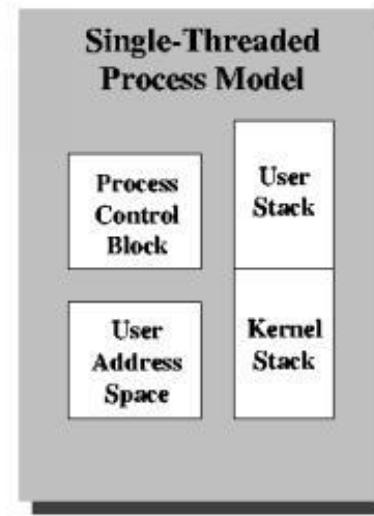
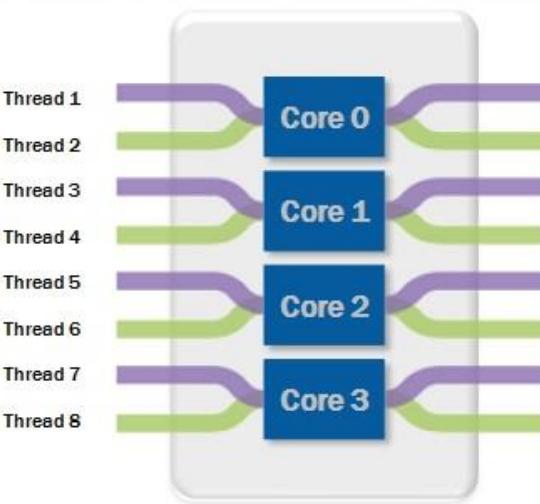
• Noções multiprocessamento (threads)

Threads X Cores



- Execução simultânea de sequências diferentes de instruções
 - Cada núcleo é uma linha de execução, mas a aplicação precisa ser programada para explorar isto (multi-thread, hyper-thread INTEL)
 - Cada núcleo processando N threads (fluxos paralelos de código)
- Programação paralela C/C++: <pthread.h>, OpenMP, MPI

• Noções multiprocessamento (threads)



Cada thread com sua própria pilha (Stack)

A thread (ou thread de execução) é uma unidade de paralelismo

Possui o necessário para executar um fluxo de instruções – pilha privada, contador de programa, trecho de código

São interessantes para cooperar em dados globais compartilhados, compartilham também sistema de arquivo

No caso de multiprocessador com cache separada, compartilhando só principal: **coerência de cache!**

• Noções multiprocessamento (threads)

Processador executando partes diferentes de um mesmo programa, ao mesmo tempo

Para o multiprocessamento ser possível

- Os circuitos de apoio da CPU (chipset) devem suportar
- O sistema operacional deve ser capaz de utilizar vários processadores simultaneamente
- O próprio processador deve poder ser usado num sistema multiprocessado
- **Os programas devem ser projetados para tirar proveito do multiprocessamento**

Tipos: **MIMD** (multiple instruction-stream, multiple data-stream): programas diferentes com dados diferentes, computadores de grande porte, SMP, cluster

SIMD: mesma sequência de instruções sobre diferentes conjuntos de dados. Executar a mesma operação com uma grande quantidade de dados diferentes. Cada elemento de processamento possui uma memória de dados associada, então cada instrução é executada num conjunto diferente de dados por processadores diferentes.

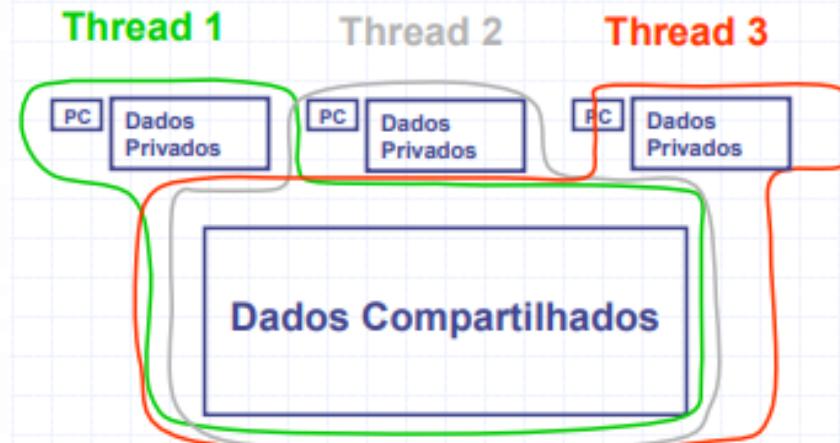
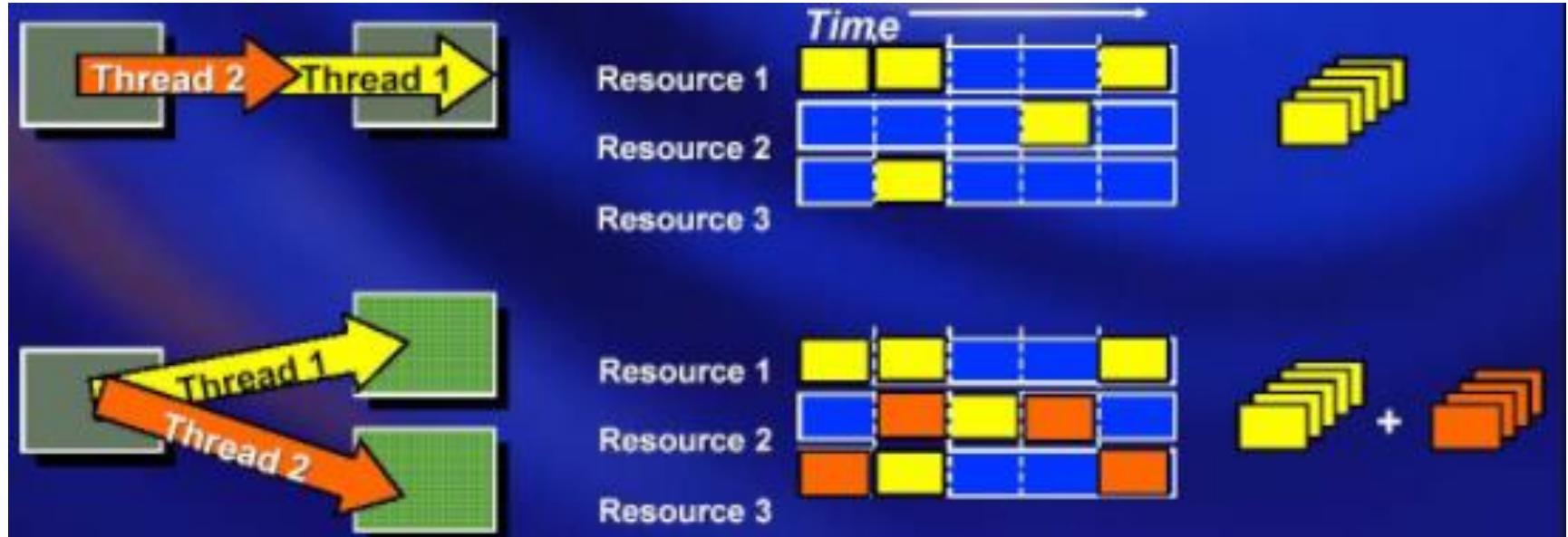
A carga precisa ser distribuída entre os processadores (load balance)

Automaticamente ou manualmente

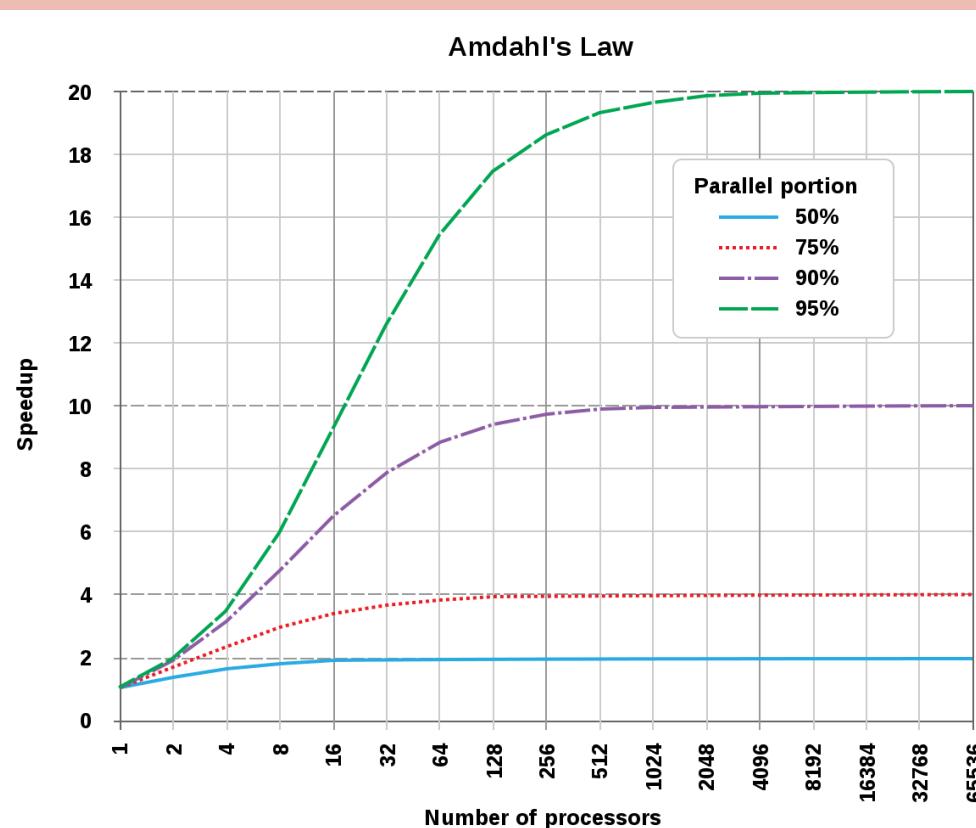
Um processador físico com várias unidades lógicas

Thread 2 precisa esperar Thread 1 finalizar, mesmo com recursos disponíveis

Já aqui, os recursos são utilizados por ambas



Uso mais eficiente da cache: é improvável que uma única thread de execução possa usar efetivamente toda a memória. Um número de threads ou processos relativamente independentes tem uma oportunidade maior de tirar vantagem da memória cache



O speedup de um programa usando múltiplos processadores em computação paralela é limitado pela fração sequencial do programa. Por exemplo, se 95% do programa pode ser paralelo, teoricamente o speedup máximo usando computação paralela seria 20x como apresentado no diagrama, não importando quantos processadores estão sendo usados.

- Desafio para o desenvolvedor: identificar partes do código ‘sequencial’ que podem ser ‘parallelizados’

$$\begin{aligned}\text{Aumento de velocidade (speedup)} &= \frac{\text{tempo para executar o programa em single core}}{\text{tempo para executar o programa em } N \text{ processadores}} \\ &= \frac{1}{(1 - f) + \frac{f}{N \text{ (cores)}}}\end{aligned}$$

Lei de Amdahl: a lei supõe um programa no qual uma fração $(1-f)$ do tempo de execução envolve código inherentemente serial e uma fração f que envolve código infinitamente parallelizável

Contudo, numa abordagem direta, não significa aumento de desempenho significativo, pois implica no aumento de sobrecarga na comunicação e distribuição de trabalho entre vários processadores e sobrecarga de coerência de cache. A curva de melhoria apresenta picos, mas depois passa a degradar. Por exemplo, se $f = 0,9$ (apenas 10% de serial), com $N = 8$, a melhoria é de 4,7. Contudo melhorias para reduzir a fração serial foram incorporadas ao longo do tempo e sistemas operacionais, bancos de dados e software para servidores usam intensamente a organização paralela

Exercício 1

- Suponha que uma tarefa faz uso intensivo de operações de ponto flutuante, com 40% do tempo consumido por operações de ponto flutuante. Com um novo design de hardware, o módulo de ponto flutuante é acelerado por um fator de K. Calcule o speedup máximo possível, considerando a Lei de Amdahl.

$$speedup = \frac{1}{(1 - f) + \frac{f}{k \text{ (melhoria)}}}$$

- Usando a Lei de Amdahl, calcule o ganho de velocidade: 67% paralelo com
 - a) 2 cores e b) 4 cores

Multicore

- Desafio para o desenvolvedor: identificar partes do código ‘sequencial’ que podem ser ‘parallelizados’
- Threads são de baixo custo computacional, mas baixo nível e gerencia mais cuidadosa – Linux/POSIX (pthreads); Windows (WinThreads)
- APIs (Application Interfaces) como OpenMP e MPI são modelos de alto nível

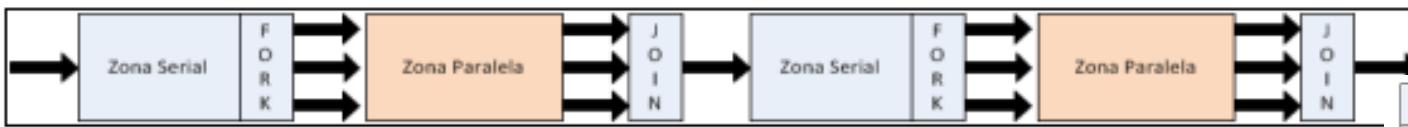


Figura 2.3. Fork-join no OpenMP

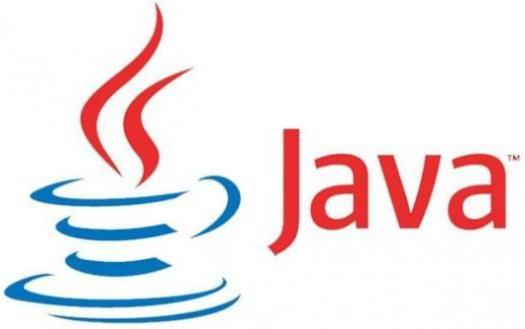
```
//soma e imprime a soma do vetor
void somaVetor(double *vetor, int tamanho)
{
    int i = 0;
    double soma=0;
    #pragma omp parallel for reduction(+:soma)
    for(i=0; i<tamanho; i++)
    {
        soma+=vetor[i];
    }
    printf("Resultado da soma %3.f\n", soma);
}
```

Figura 2.4. OpenMP: a diretiva `#pragma omp parallel for` informa ao compilador que a região deve ser parallelizada.

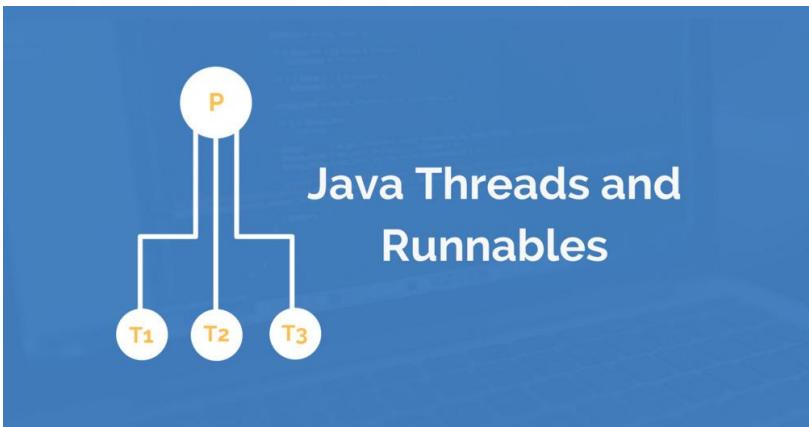
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    // Instruções MPI e a implementação do seu código
    MPI_Finalize();
    return 0;
}
```

Figura 2.6. Estrutura de um código em C/MPI.



Observação: não apenas a linguagem Java facilita aplicações *multithread*, mas a Java Virtual Machine é um processo *multithread* que provê agendamento de memória para aplicações Java. Aplicações Java podem, então, se beneficiar diretamente dos recursos *multicore*.

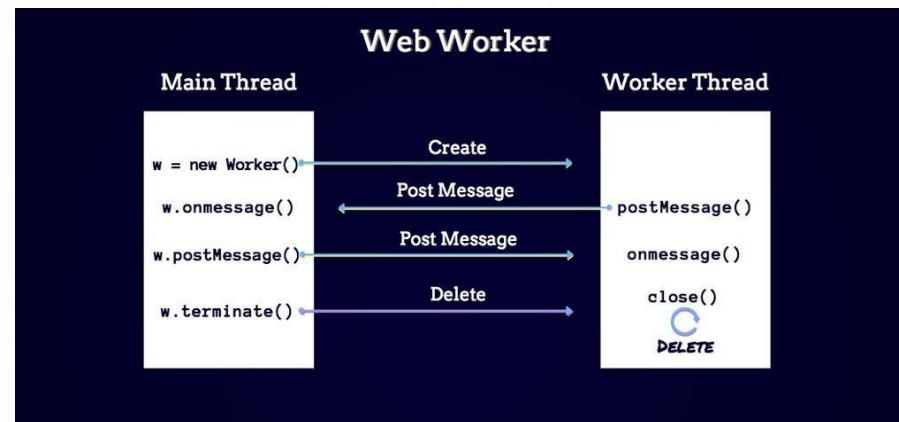


Stream API:

```

1 class KingKong {
2
3     public static synchronized void main(String[] args) {
4         Thread t = new Thread() {
5             public void run() {
6                 kong();
7             }
8         };
9
10        t.start();
11        System.out.print("King");
12    }
13
14    public static synchronized void kong() {
15        System.out.print("Kong");
16    }
17 }

```



Javascript

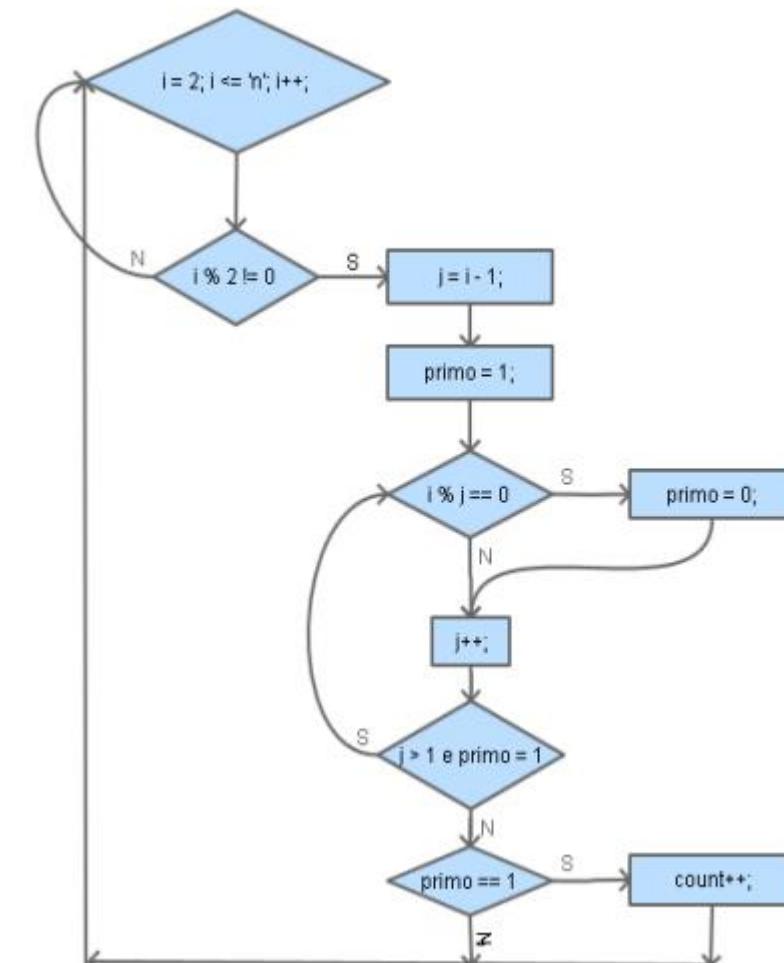
Silva, Cezar G. A.; Asenjo, Maurício N. Programação paralela – uma introdução ao paralelismo com OPEN MPI. In: 13. Congresso Nacional de Iniciação Científica, 2013. Disponível em: <<http://conic-semesp.org.br/anais/files/2013/trabalho-1000015028.pdf>>

Com as aplicações rodando em um computador equipado com o processador Intel Core i3 M350x4 em ambiente Linux, foram obtidos os seguintes resultados:

Quadro 2 – Representação dos resultados obtidos em um processador Intel Core i3.

Quantidade de Números	Sem Paralelismo	Paralelo - 2 Processos	Paralelo - 4 Processos
100	0,000028s	1,175111s	1,039873s
1.000	0,002574s	1,027346s	1,047605s
10.000	0,110456s	1,106495s	1,111911s
100.000	10,531448s	8,884402s	7,512204s
1.000.000	1050,243978s	786,315063s	579,89856s

A presença do paralelismo é evidente nos maiores números e também quando levamos o paralelismo aos limites do processador. Durante a execução da aplicação sem paralelismo foi observado que o uso de CPU não ultrapassou os 30%, porém no algoritmo paralelo dividido em quatro processos o uso de CPU atingiu a marca de 100%, ou seja, o processador estava trabalhando com carga máxima durante o paralelismo.

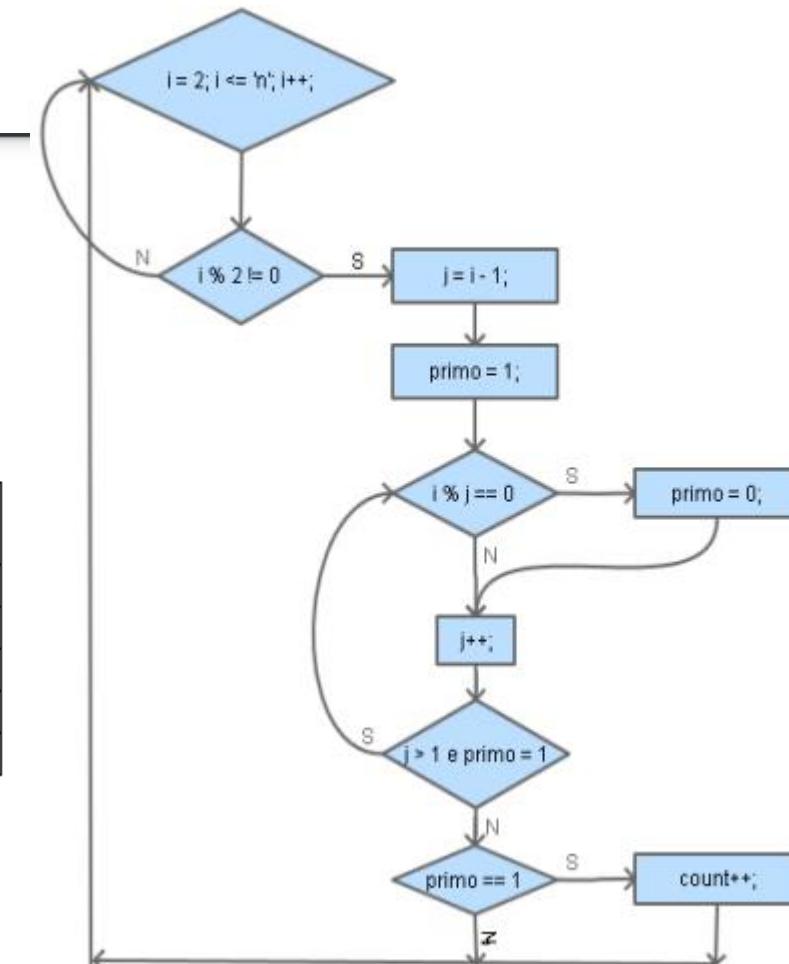


Silva, Cezar G. A.; Asenjo, Maurício N. Programação paralela – uma introdução ao paralelismo com OPEN MPI. In: 13. Congresso Nacional de Iniciação Científica, 2013. Disponível em: <<http://conic-semesp.org.br/anais/files/2013/trabalho-1000015028.pdf>>

Para ter uma melhor comparação o teste foi realizado também sobre outra configuração, um computador equipado com um processador Intel Core i7x8 em um ambiente Linux rodando em uma máquina virtual. E os resultados obtidos foram:

Quadro 3 – Representação dos resultados obtidos em um processador Intel Core i7.

Quantidade de Números	Sem Paralelismo	Paralelo - 4 Processos	Paralelo - 8 Processos
100	0,000012s	1,060125s	1,142675s
1.000	0,000467s	1,065693s	1,089045s
10.000	0,106573s	1,083212s	1,867748s
100.000	9,743917s	5,705782s	4,066512s
1.000.000	959,056274s	496,121704s	373,259491s



Encontrar o número de primos numa determinada faixa