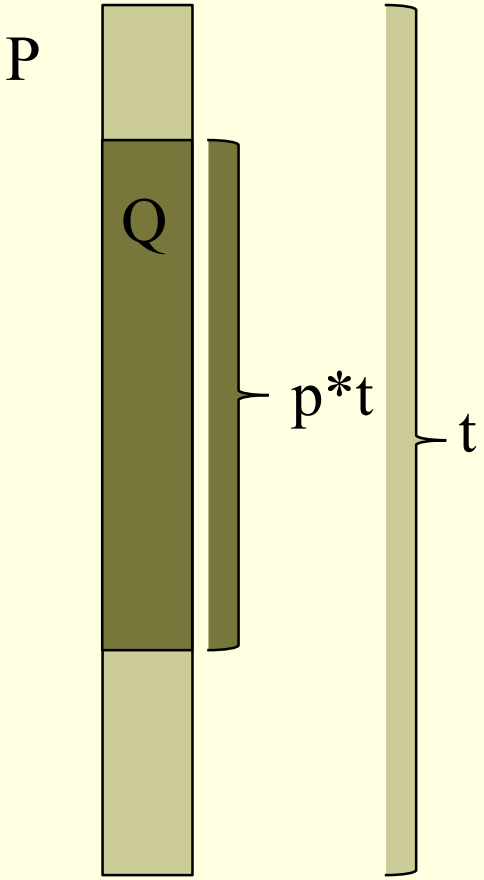
Sadržaj:

- Alati metode za poboljšavanje performansi
  - Profileri
  - Biblioteke
  - Programski jezici
- VTune Amplifier
- Integrated Performance Primitives IPP
- Domaće zadaće 3 i 4

Osnove profiling-a

Vrijeme izvođenja programa P je t
Dio programa Q u omjeru p se može ubrzati N puta
Koliko je ubrzanje cijelog programa U?



Amdahlov zakon:

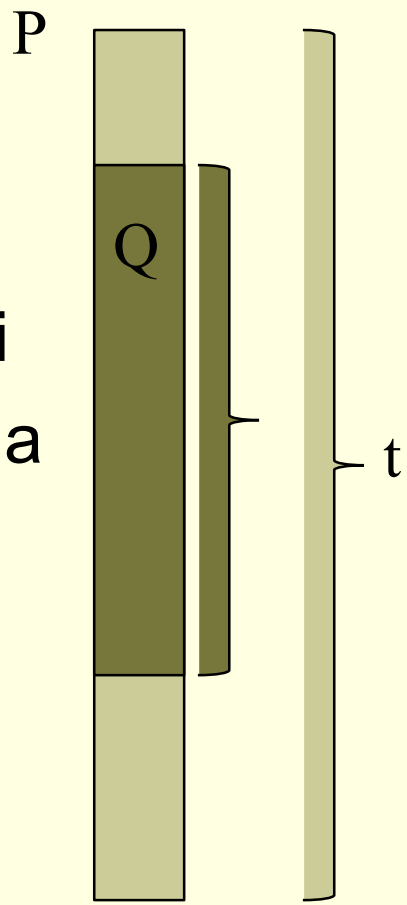
$$U = \frac{t}{t'} = \frac{t}{t\left((1-p) + \frac{p}{N}\right)} = \frac{1}{\left(1 - p + \frac{p}{N}\right)}$$

Ako $N \longrightarrow \infty$

$$U \approx \frac{1}{1-p}$$

Osnove profiling-a

- Zašto je Amdahl-ov zakon bitan
  - Jer definira realnu situaciju u kojoj se dio programa ne može ubrzati, a postoje i dijelovi koji se mogu ubrzati
  - Definira maksimalnu granicu ubrzanja
  - Pomaže u definiranju strategije ubrzavanja programa



Osnove profiling-a

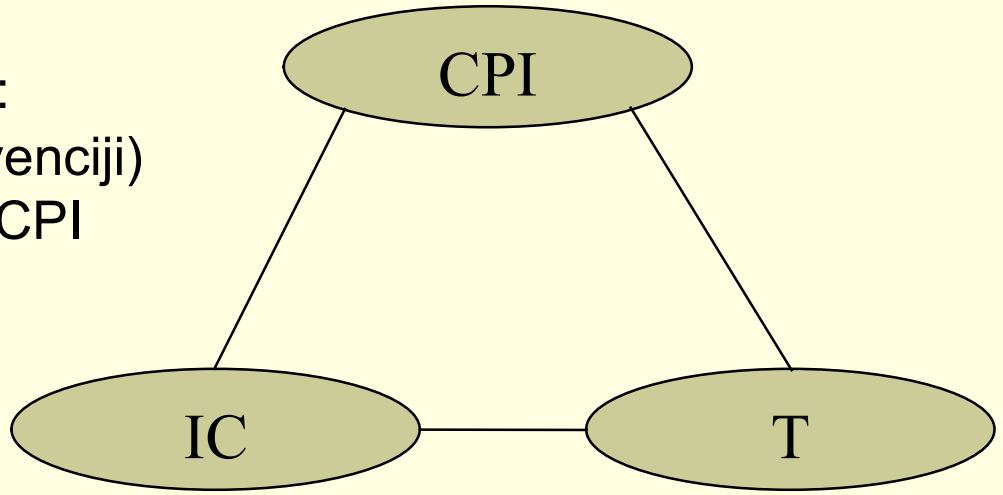
1. Cilj profiliranja je pronaći dijelove programa koji troše najviše resursa (usko grlo)
   - Vrijeme
   - Energija
   - Sklopovski: memorija, dma, disk, sabirnica
2. Definirati strategiju ubrzavanja identificiranih dijelova
3. Strategije ubrzavanja:
   - Asembler (SIMD-izacija)
   - Biblioteke (IPP, MKL, boost, BLAS, LAPACK)
   - Paralelizam (TBB, OpenMP, CUDA, OpenCl, MPI)
   - Novi pristupi (DSL, Go, Cilk+)

Osnove profiling-a

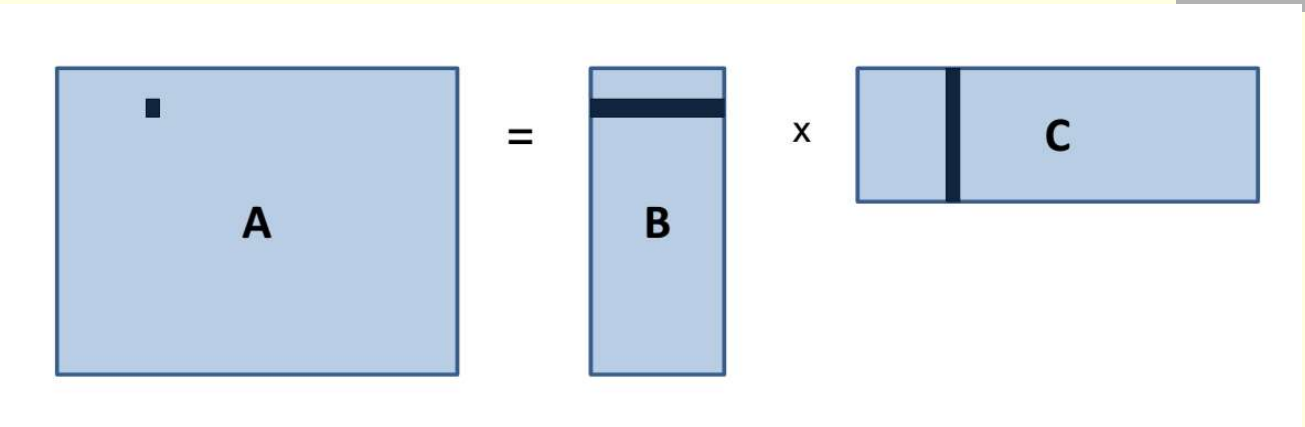
$$T = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clockcycle}} = \text{CPU time}$$

Računske performanse ovise o:

- Vremenskom taktu T (frekvenciji)
- Broju ciklusa po instrukciji CPI
- Broju instrukcija IC



Osnove: Množenje matrica

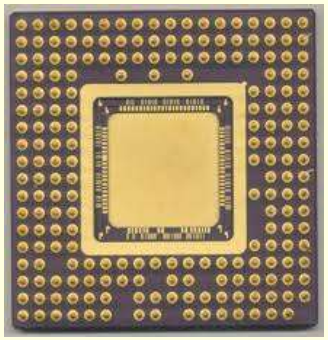


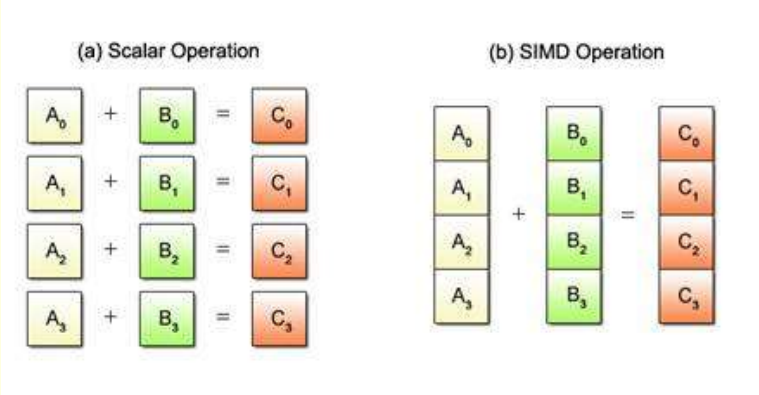

```
for(int i = 0; i < x; i++)
    for(int j = 0; j < y; j++)
        for(int k=0; k < z; k++)
            A[i][j] += B[i][k] * C[k][j];
```

- Matrica B: pristup po retcima
- Matrica C: pristup po stupcima

Jednoprocesorski paralelizam

- ILP – Instruction Level Parallelism
  - Arhitekti računala skrivaju paralelizam (2002)
  - Protočna arhitektura (preklapanje dijelova instrukcija)
  - Superskalarno izvođenje (obrada više instrukcija u isto vrijeme)
  - VLIW: Prevoditelj određuje ILP
  - Vektorska obrada: Grupe sličnih nezavisnih operacija (SIMD)
  - “Out of Order Execution”: Instrukcije se prividno izvode po redu u toku instrukcija, stvarno ovisno o slobodnim resursima



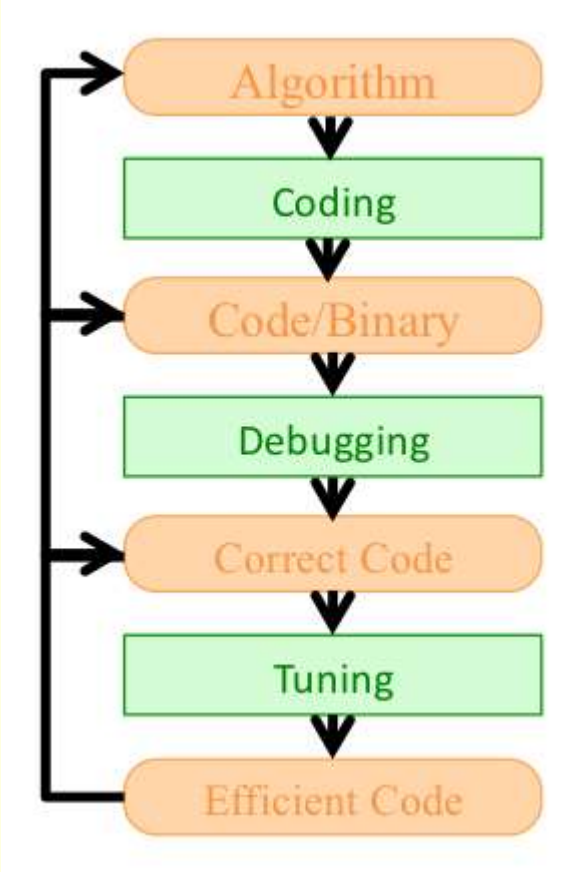


Ograničenja ILP-a

- Hazardi:
  - Strukturni – računski resursi
  - Podatkovni – međuovisnost podataka
  - Kontrolni – Upravljanje tijekom izvođenja
- Granice (Wall)
  - Power wall
  - ILP wall
  - Memory wall

Performance profiling

- Razvojni ciklus
- Tuning:
  - Mjerenje performansi
  - Analiza mjerenja
  - Modifikacija algoritma
  - Ponovno mjerenje
  - Analiza razlika



Kako mjeriti performanse

- Mjerenje vremena izvođenja
  - Načelno
  - Ne mogu se otkriti kritične točke
- Integracija programskog koda za mjerenje
  - C/C++: clock_t, clock() iz <time.h>
  - Nije pogodno za održavanje
- Korištenja alata za profiling
  - Vtune Amplifier, gprof, …
  - Detaljna analiza
  - Povezanost sa izvornim kodom

Tipovi *profiling*-a

- Statističko uzorkovanje (*statistical sampling*)
  - Periodičko prekidanje izvođenja i pamćenje lokacije
  - Analiza statističke distribucije
  - Vremensko skupljanje podataka (vrijeme)
  - Uniforman *overhead*
- Praćenje događaja (*event tracing*)
  - Skupljanje individualnih događaja (pozivi funkcija, razmjena poruka)
  - Detaljna analiza događaja
  - Velika količina podataka i *overhead*

Gnu profiler: gprof

- Unix/Linux
  - gcc –pg program.cpp –o program
  - ./program
  - gprof program > program.prof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 92.63      2.75     2.75     4096   671.68   671.68  dct()
  2.69      2.83     0.08    12288     6.51     6.51  pipe(rlstruct*, pestruct*)
  1.68      2.88     0.05     4096    12.21    12.21  rgb2yuv()
  1.35      2.92     0.04    12288     3.26     3.26  runlen(int*, rlstruct*, int)
  0.67      2.94     0.02   823296     0.02     0.04  codeinsert(int)
  0.34      2.95     0.01   823296     0.01     0.01  status(pestruct*, int*, int)
  0.34      2.96     0.01   113555     0.09     0.09  order(char*, int*, int, int, int*)
  0.34      2.97     0.01     4096     2.44     2.44  getblock(int)
  0.00      2.97     0.00    12288     0.00     0.00  zigzag(int*)
  0.00      2.97     0.00    12288     0.00    13.03  entropy(int*, int)
```

GNU profiler: gprof

- *Flat profile*
  - Provedeno vrijeme u svakoj funkcij
  - Broj poziva funkcija
  - Jednostavan pronalazak vremenski kritičnih funkcija
- *Call graph*
  - Analiza po funkcijama (tko je pozvao, koga sam pozvao, koliko puta)
  - Procjena koliko vremena je provedeno u pozvanim funkcijama
  - Potencijalna mjesta za uklanjanje poziva

Vtune Amplifier

- Performanse CPU, GPU
- Skalabilnost, Propusnost, Dretve
- Vizualizacija rezultata
- *Hotspot*
  - Mjesto u programu s izraženom aktivnosti
    - Vrijeme
    - Pristup memoriji

Vtune Amplifier

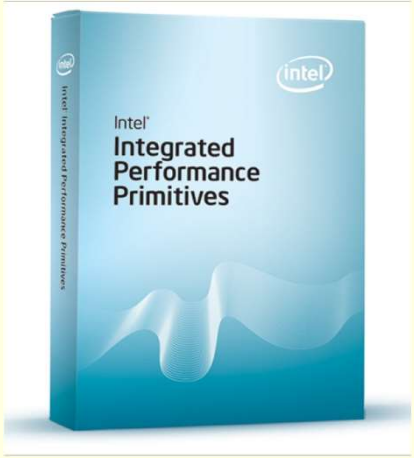
- Identifikacija *hotspot-ova*
- Identifikacija neučinkovitih dijelova programa
- Identifikacija dijelova koji su pogodni za optimizaciju
- Analiza sinkornizacijskih objekata koji utječu na iskorištenost sustava
- Analiza I/O operacija
- Aktivnost dretvi i tranzicije
- Sklopovski kritične točke u programu

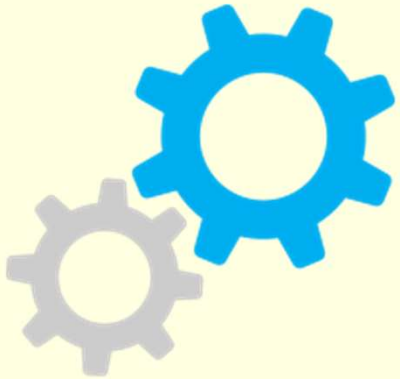
Terminologija

- **Target***:* izvršni program koj se analizira
- **Baseline:** osnovna mjera i model koji se koristi za usporedbu
- **CPU time:** vrijeme kojeg program izvodi na procesoru (za više dretvi, CPU vremena svih dretvi su zbrojena u programsko CPU vrijeme)
- **Elapsed time:** ukupno vrijeme (wall clock time) potrebno za izvođenje programa
- **Hotspot:** Dio programa sa značajnim doprinosom u ukupnom vremenu izvođenja programa
- **CPI rate:** broj taktova po instrukciji (*clocks per instruction)*

Integrated Performance Primitives IPP

- Biblioteka optimiranih funkcija za multimediju, obradu i kodiranje audio i video podataka, vektorsku manipulaciju, konverziju, kriptografiju itd.
- Zasniva se na apstrakciji procesorskih svojstava kao što su MMX, SIMD, SSEx.
- Nedostatak: Ovisnost o arhitekturi procesora i ekstenzija





Integrated Performance Primitives IPP

Code of Domain	Header file	Prefix	Description
CC	ippcc.h	ippi	color conversion
CH	ippch.h	ipps	string operations
CORE	ippcore.h	ipp	core functions
CP	ippcp.h	ipps	cryptography
CV	ippcv.h	ippi	computer vision
DC	ippdc.h	ipps	data compression
I	ippi.h	ippi	image processing
S	ipps.h	ipps	signal processing
VM	ippvm.h	ipps	vector math
E*	ippe.h	ipps	embedded functionality

* available only within the Intel® System Studio suite

Intel IPP

- Imenovanje funkcija:

ipp<data-domain><name>_<datatype>[_<descriptor>](<parameters>);

- Primjer:

```
ippiRGBToYUV_8u_C3R();
```

Tip podataka

Naziv funkcije: Pretvorba RGB u YUV

Format zapisa podataka

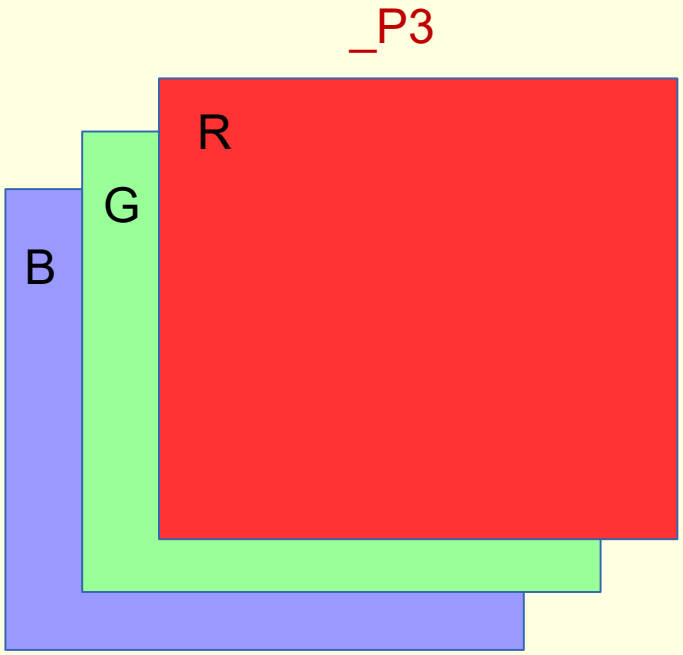
IPP funkcija, i: image processing

Intel IPP – Image processing

- Format zapisa: raspored komponenti slikovnih podataka
  - Kanalni raspred (Channel Data Layout)
    - Oznaka: “_C”
  - Planarni format (Planar Data Layout)
    - Oznaka: “_P”

_C3

RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB



Primjer: RGB → YUV

- IppStatus ippiRGBToYUV_<mod>

  (const Ipp8u* pSrc, int srcStep, Ipp8u* pDst, int dstStep, IppiSize roiSize);

  - <mod>: 8u_C3R, 8u_AC4R
- IppStatus ippiRGBToYUV_8u_P3R

  (const Ipp8u* const pSrc[3], int srcStep, Ipp8u* pDst[3], int dstStep, IppiSize roiSize);
- IppStatus ippiRGBToYUV_8u_C3P3R

  (const Ipp8u* pSrc, int srcStep, Ipp8u* pDst[3], int dstStep, IppiSize roiSize);

IPP – Korisne funkcije

IppStatus ippiDCT8x8FwdLS_<mod>()

IppStatus ippiDCT8x8Inv_<mod>()

Više u opsežnoj dokumentaciji

https://software.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top.html

Jednoprocesorski sustavi

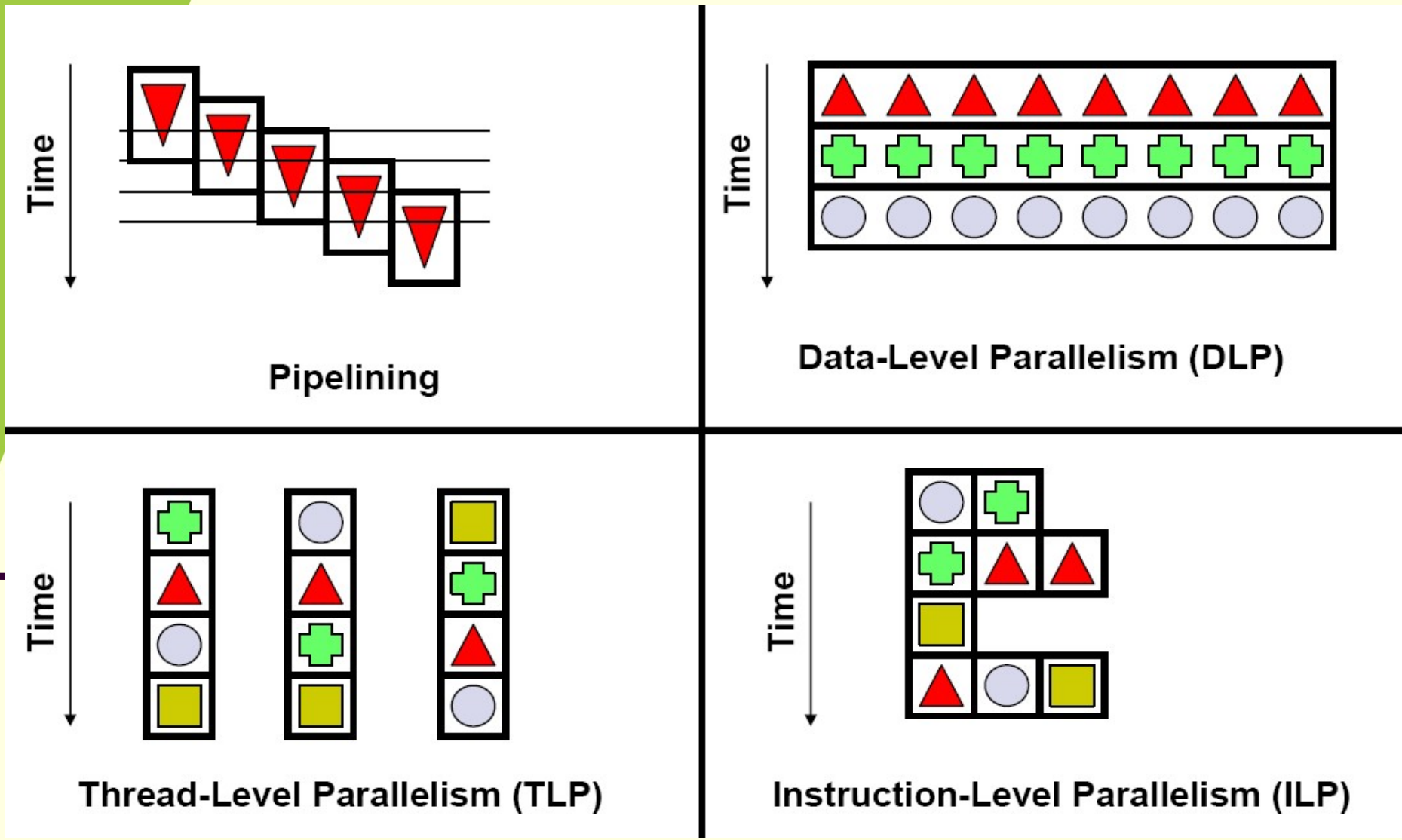
Problemi sa skaliranje uniprocesorske arhitekture:

- Disipacija snage
- Efikasnost
- Kompleksnost
- Kašnjenje u interkonekcijama
- Usporen rast u performansama
- ILP – ne može više dodatno povećavati performanse

Multimedija na višejezgrenim arhitekturama

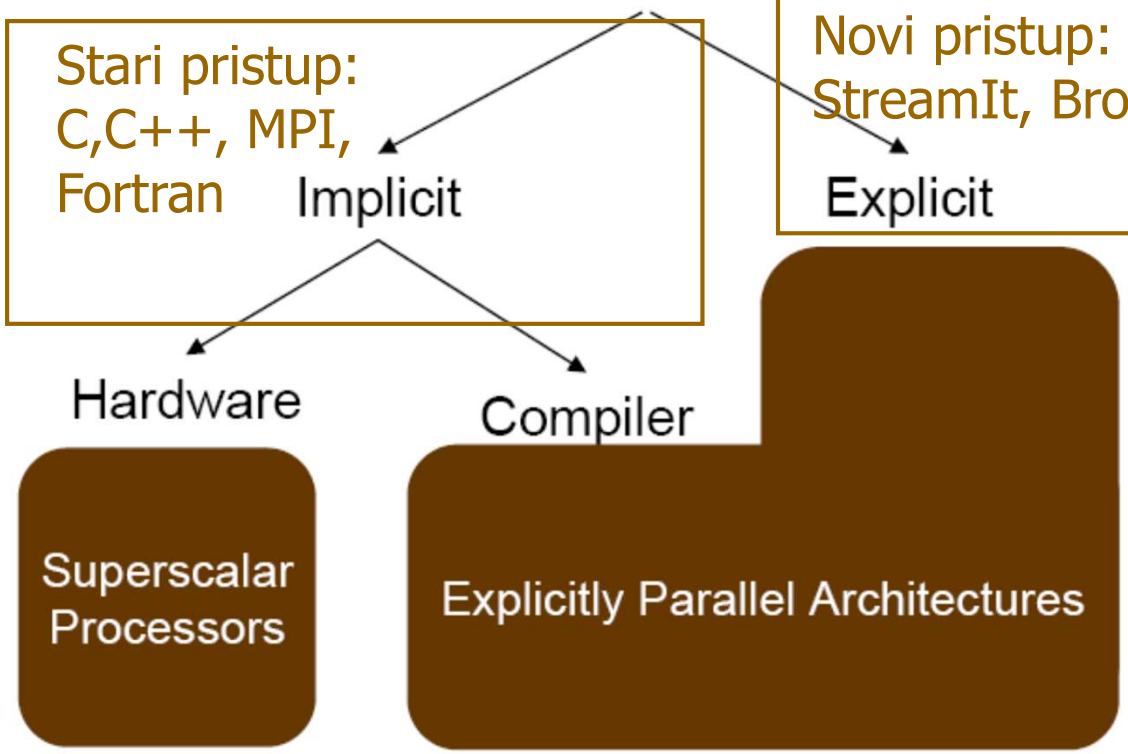
- PARALELIZAM U MM APLIKACIJAMA !!!
- Podatkovni paralelizam DLP (data level)
- Paralelizam među zadatcima TLP (task, thread level)
- Protočni paralelizam PLP (pipeline level)
- Instrukcijski paralelizam ILP (instruction level)

Tipovi paralelizma



Paralelizam u MM aplikacijama

Kako ga opisati i iskoristiti



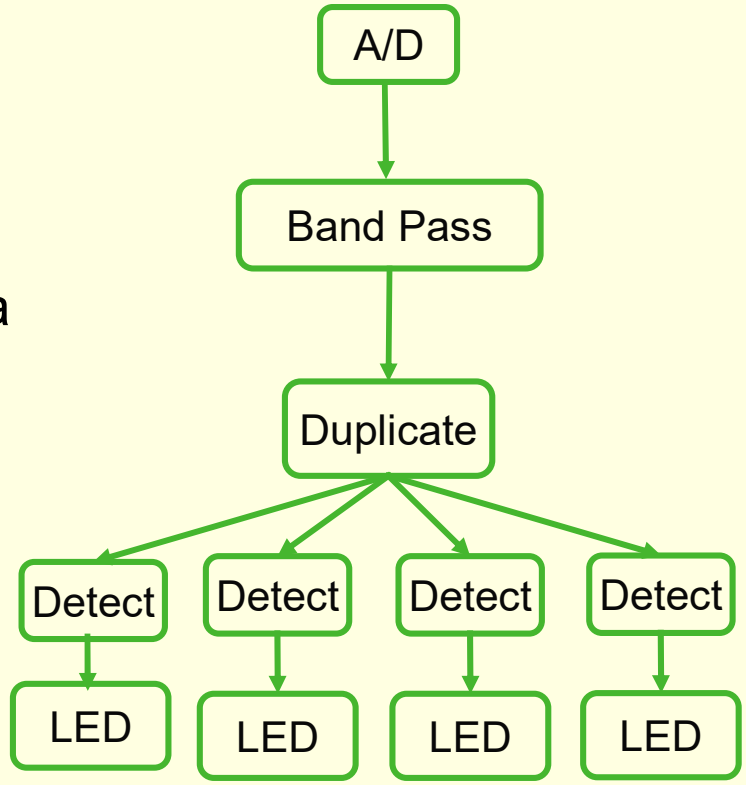
Stari pristup:
- Neprirodno
- Neprenosivost
- Verifikacija!

Novi pristup:
- Prirodno
- Prenosivost
- Verifikacija

http://cag.csail.mit.edu/streamit

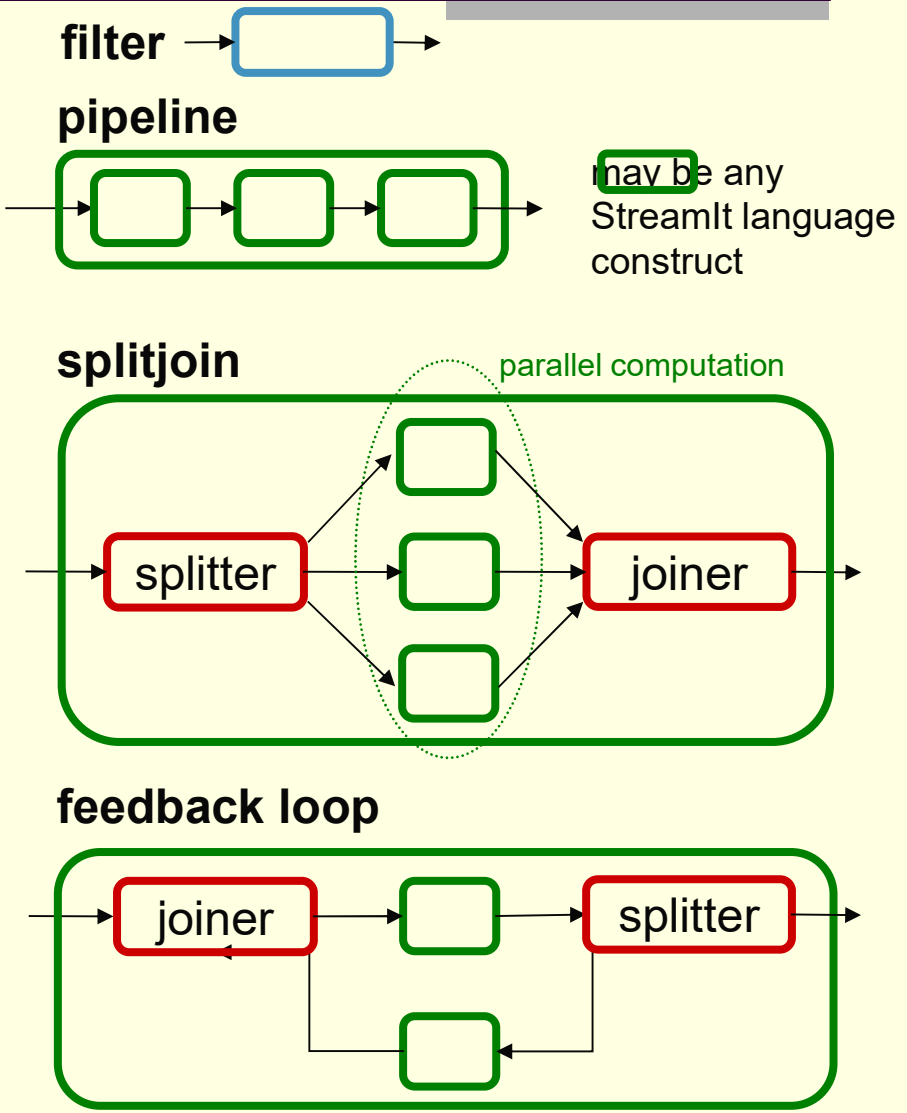
StreamIt (http://cag.csail.mit.edu/streamit)

- SDF Model [Lee_87]
- Graf autonomnih aktora
- Aktori imaju lokaliziran adresni prostor
- Komunikacija preko FIFO kanala
- Statički definirana brzina I/O
- Prevodioc određuje redosljed izvođenja



Programski jezik StreamIt

- SDF s dinamičkim proširenjima
- Paralelizam i komunikacija: eksplicitno izraženi u programu
- Neovisnost o arhitekturi
- Portabilnost
- Modularnost (lako slaganje filtera u složenije grafove toka)
- Skalabilnost
- Osnovne konstrukcije:
- Filter (osnovna jedinica)
- Pipeline (sekvenca filtera)
- Splitjoin (scatter-gather)
- Feedback loop

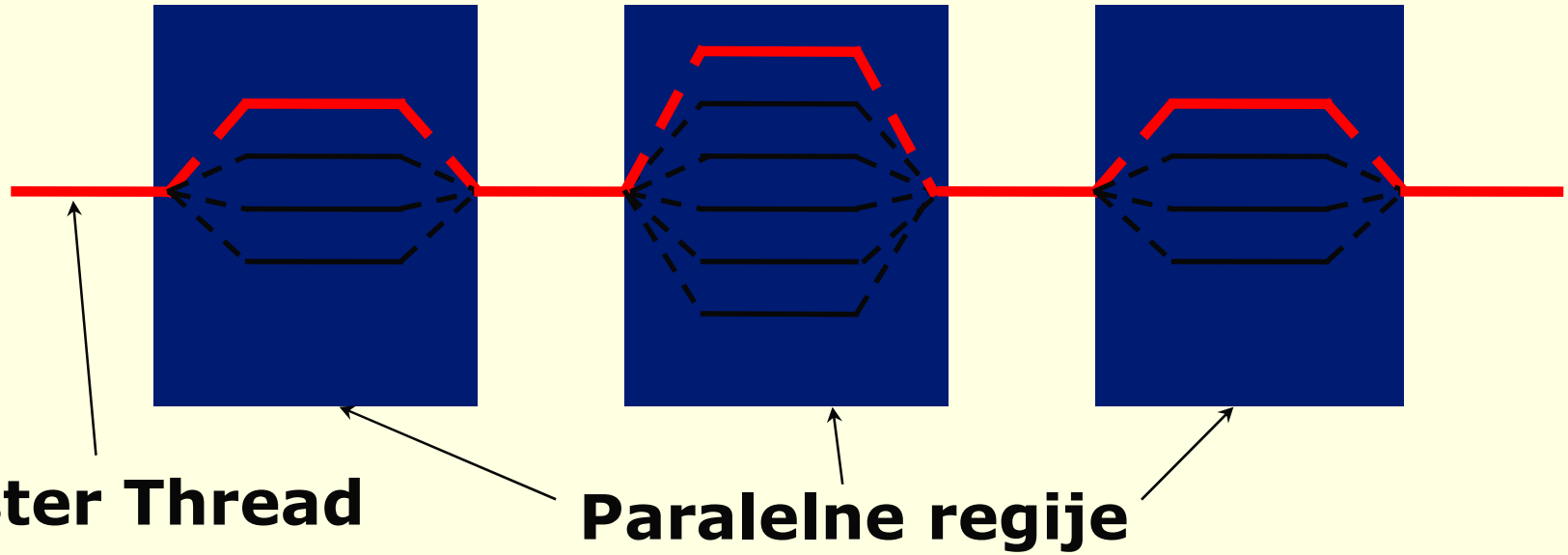


OpenMP

- Skup direktiva i rutina (biblioteka): omogućuje portabilne paralelne aplikacije na SMP arhitekturama
- C/C++, Fortran
- Direktive prevoditelju (compiler directives)
- Data parallelism model i Task parallelism
- Inkrementalni paralelizam
- Kombinira serijski i paralelni kod

OpenMP biblioteka

- Fork-join paralelizam (zasnovan na multithreadingu)
- Master thread pokreće druge threadove
- Inkrementalno dodavanje parlelizma – sekvencijalni program evoluira u paralelni

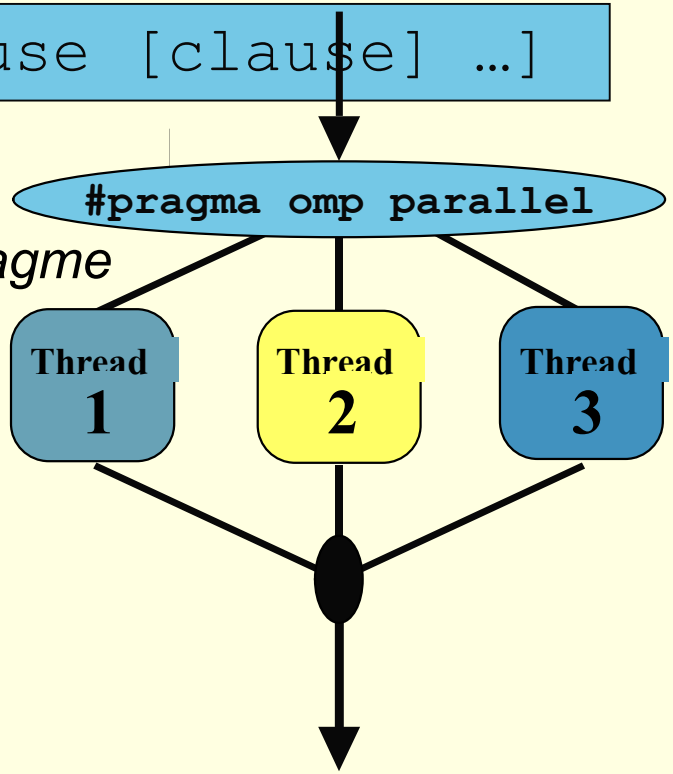


OpenMP biblioteka - sintaksa

Većina OpenMP ključnih riječi je u formi direktiva prevoditelju ili pragma

```
#pragma omp construct [clause [clause] …]
```

- *Paralelne regije*
- *Threadovi se kreiraju pri prolazu pragme*
- *Blokiranje na kraju regije*




```
#pragma omp parallel
{
block
}
```

OpenCL

Heterogeni multiprocesorski sustavi

- CPU, GPU, akcelerator

Model podatkovnog paralelizma (CUDA)

Model paralelizma među zadacima

- Thread -> Work item
- Thread block -> Work group

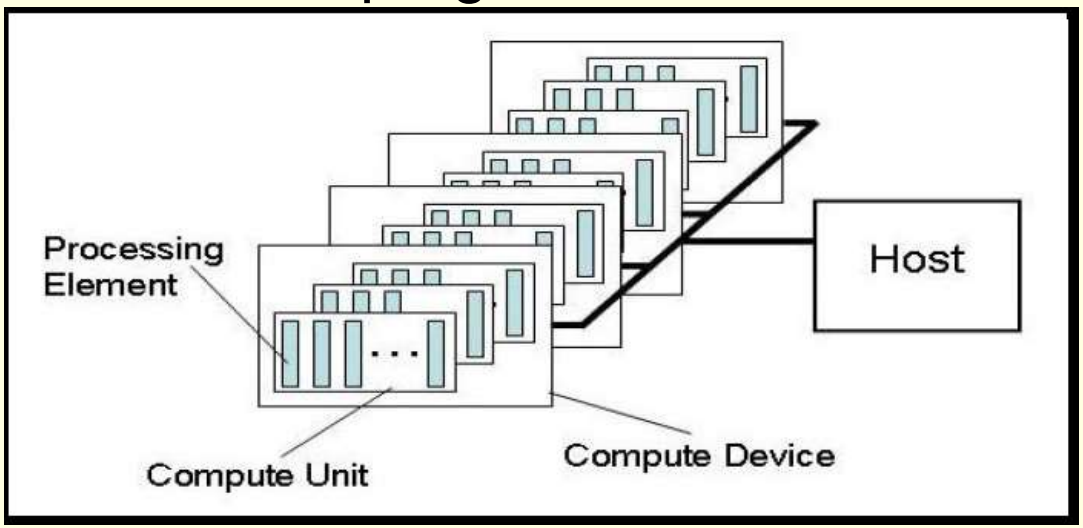
Paralelizam, paralelizam, paralelizam

- Tisuće niti za punu paralelizaciju
- Fokus na “on-chip” memoriju i komunikaciju

OpenCL

Heterogeni multiprocesorski sustavi

- OpenCL device: skup jedne ili više računskih jedinica (compute unit: host + devices (x86 host + GPU device))
- Compute unit: Skup procesnih elemenata (processing elements)
- Processing elements: izvode programski kod



OpenCL izvedbeni model

Kernel (jezgra)

- Osnovna jedinica izvedbenog koda (C funkcija)
- Podatkovno-paralelan, zadatkovno-paralelan

Program

- Skup jezgri i drugih funkcija (dll)

Aplikacija

- Kreira kontekst za upravljanje i izvođenje OpenCL jezgri, razmjenu podataka host-device
- Rep instanci OpenCL jezgri

OpenCL program

Dinamički model prevođenja (OpenGL)

- API pozivi koji omogućuju dinamičku optimizaciju za postojeći OpenCL device

Kreiranje jezgre:

1. OpenCL program spremljen u tekstualnom obliku (učitan u memoriju)
2. Kreiranje programa, API poziv clCreateProgramWithSource()
3. Prevođenje programa za neki OpenCL uređaj, API poziv clBuildProgram()
   - x86: instrukcijski kod
   - GPU: IL (PTX) reprezentacija programa
4. Ekstrakcija jezgre, API poziv clCreateKernel()
5. Prijenos argumenata i “dispatching”, API clSetKernelArg() I clEnqueueNDRangeKernel()

Password hashing

- Introduced in 1970s
- Sense of security which is often false
- Fast hashes
  - MD4, MD5, SHA-1
  - Designed for MACs and digital signatures
  - Must be easy to compute
- Slow hashes
  - bcrypt, PBKDF2, scrypt
  - Designed for password hashing
  - Inefficient and difficult to compute

način rada	cijena	sol	sažetak
$2y$	10	$VaohDzrEa32G3DuTqduoCe	05cwi9EsLCniXVABkXeC2xq4z74ZbYG

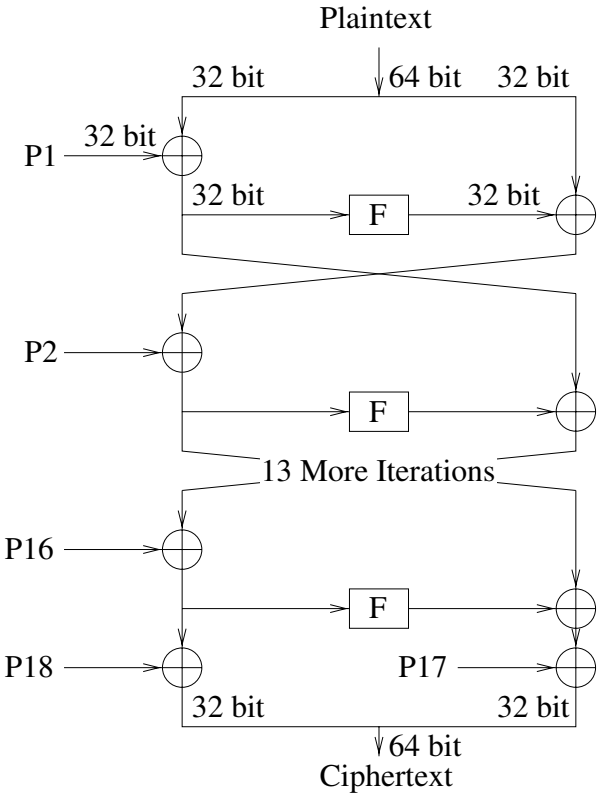
Motivation

- Bcrypt is:
  - Popular (often used): WWW, OpenBSD, SUSE Linux
  - Slow
  - Sequential
  - Random memory footprint
  - Designed to be resistant to brute force attacks and to remain secure despite hardware improvements
- You could almost think why even bother optimizing

Bcrypt (Provos and Mezieres)

- Based on Blowfish block cipher (Bruce Schneier)
- Expensive key setup
- User defined cost setting
  - Cost setting between 4 and 31 inclusive is supported
  - Cost 5 is traditionally used for benchmarks for historical reasons
  - All given performance figures are for bcrypt at cost 5
  - Current systems should use higher cost setting
- Pseudorandom memory accesses
- Memory usage
  - 4 KB for four S-boxes
  - 72 B for P-box

Blowfish encryption



- Symmetric-key cipher
- 64-bit input block, 32 to 448-bit key
- Feistel network
- Pseudorandom memory accesses
  - 32-bit loads from four 1 KB S-boxes initialized with digits of number π

$$R_i = L_{i-1} \oplus P_i \quad (1)$$

$$L_i = R_{i-1} \oplus F(R_i) \quad (2)$$

$$F(a, b, c, d) = ((S_1[a] + S_2[b]) \oplus S_3[c]) + S_4[d] \quad (3)$$

Niels Provos and David Mazieres, "A Future-Adaptable Password Scheme", The OpenBSD Project, 1999

EksBlowfish

Ekspensive key schedule Blowfish

Algorithm 1 EksBlowfishSetup(cost, salt, key)

1: $state \leftarrow InitState()$
2: $state \leftarrow ExpandKey(state, salt, key)$
3: $repeat(2^{cost})$
4: $\quad state \leftarrow ExpandKey(state, 0, salt)$
5: $\quad state \leftarrow ExpandKey(state, 0, key)$
6: $return\ state$

- Order of lines 4 and 5 is swapped in implementation

Niels Provos and David Mazieres, "A Future-Adaptable Password Scheme", The OpenBSD Project, 1999

bcrypt

Algorithm 2 bcrypt(cost, salt, pwd)

1: $state \leftarrow EksBlowfishSetup(cost, salt, key)$
2: $ctext \leftarrow$ "*OrpheanBeholderScryDoubt*"
3: $repeat(64)$
4: $\quad ctext \leftarrow EncryptECB(state, ctext)$
5: $return\ Concatenate(cost, salt, ctext)$

Output: (bcrypt_indicator, cost, 128-bit base-64 22 chars salt, 184-bit base-64 31 chars hash)

$2a$12$GhvMmNVjRW29ulnudl.LbuAnUtN/LRfe1JsBm1Xu6LE3059z5Tr8m

Architecture

Epiphany

- 16/64 32-bit RISC cores operating at up to 1 GHz/800 MHz
  - Chips used in testing operate at 600 MHz
- Pros
  - **Energy-efficient** - 2 W maximum chip power consumption
  - 32 KB of local memory per core
  - 64 registers
  - FPU can be switched to integer mode
  - Dual-register (64-bit) load/store instructions
- Cons
  - FPU in integer mode can issue only add and mul instructions
  - Only simple addressing modes
    - Index scaling would be helpful for S-box lookups

Implementation

Epiphany

- John the Ripper prepares data on ARM cores
- Bcrypt hashes computed on Epiphany
- Single instance does not have enough instruction level parallelism
  - FPU has four cycle latency
  - FPU does not have bitwise instructions
- Optimized in assembly
- Two instances overlapped to exploit dual-issue architecture
  - Integer ALU
  - FPU in integer mode

Architecture

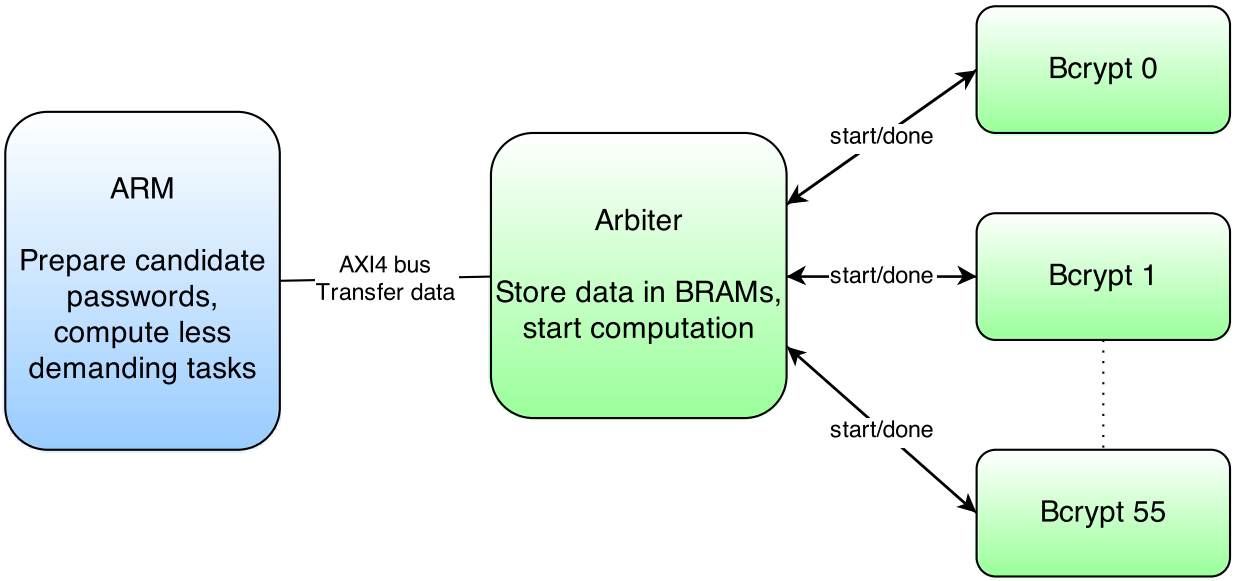
Zynq 7020

- Heterogeneous device
- Dual ARM Cortex-A9 MPCore
  - 667 MHz
  - 256 KB on-chip memory
- Advanced low power 28nm programmable logic
  - 85 K logic cells
  - 560 KB of block RAM
- AXI buses used for CPU-FPGA communication

Implementation

Zynq 7020

- John the Ripper prepares data on ARM cores
- Bcrypt instances compute hash
- Number of concurrent instances limited by available BRAM
- Large communication overhead for low cost setting
- Hardware defects of ZedBoard limit optimizations



Zynq 7045

- Architecture
  - ARM CPU and Zynq reconfigurable logic
  - Roughly 4 times bigger than Zynq 7020
- Implementation
  - ZedBoard implementation ported to a bigger device
  - Bottleneck: CPU-FPGA communication (for low cost setting)
  - Not possible to use all available resources due to hardware defects of ZC706 board

Theoretical Peak Performance Analysis

Theory

$$c/s = \frac{N_{ports} * f}{(2^{cost} * 1024 + 585) * N_{reads} * 16} \quad (4)$$

- N_{ports} - number of available read ports to local memory or L1 cache
- N_{reads} - number of reads per Blowfish round
  - 4 or 5 depending on whether reads from P-boxes go from one of those read ports we've counted or from separate storage such as registers
- $2^{cost} * 1024 + 585$ - number of Blowfish block encryptions in bcrypt hash computation
- f (in Hz) - clock rate

bcrypt(cost, salt, pwd)

```
1: state ← InitState()
2: state ← ExpandKey(state, salt, key)
3: repeat(2^cost)
4:     state ← ExpandKey(state, 0, salt)
5:     state ← ExpandKey(state, 0, key)
6: ctext ← "OrpheanBeholderScryDoubt"
7: repeat(64)
8:     ctext ← EncryptECB(state, ctext)
9: return Concatenate(cost, salt, ctext)
```

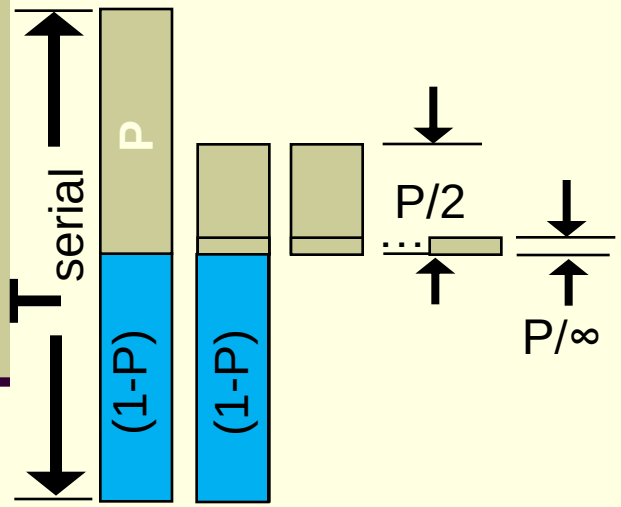
Takeaways

- Many-core low power RISC platforms and FPGAs are capable of exploiting bcrypt peculiarities to achieve comparable performance and higher energy-efficiency
- Higher energy-efficiency enables higher density
  - More chips per board, more boards per system
- It doesn't take ASICs to improve bcrypt cracking energy-efficiency by a factor of 45+
  - Although ASICs would do better yet

Amdahl's Law

- **Amdahl's Law:**
  - Is a law governing the *speedup* of using parallel processors on an application, versus using only one serial processor.
  - Describes the upper bound of parallel execution speedup.

n = ∞



0.5 + 0.0

$T_{parallel} = \{(1-P) + P/n\} T_{serial}$

n = number of processors

Speedup = $T_{serial} / T_{parallel}$

1.0/0.5 = 2.00

Parallel Efficiency

Parallel Efficiency:

- Is a measure of how efficiently processor resources are used during parallel computations.
- Is equal to (*Speedup / Number of Threads) * 100%.*

CONCURENCY != PARALLELISM

Composition
Dealing

Execution
Doing

Granularity

Definition:

- An approximation of the ratio of *computation* to *synchronization*.

The two types of granularity are:

- **Coarse-grained:** Concurrent calculations that have a large amount of computation between synchronization operations are known as *coarse-grained*.
- **Fine-grained:** Cases where there is very little computation between synchronization events are known as *fine-grained*.

Summary

- A *thread* is a discrete sequence of related instructions that is executed independently. It is a single sequential flow of control within a program.
- The *benefits* of using threads are increased performance, better resource utilization, and efficient data sharing.
- The *risks* of using threads are data races, deadlocks, code complexity, portability issues, and testing and debugging difficulty.
- Every process has at least one thread, which is the main thread that initializes the process and begins executing the initial instructions.
- All threads within a process share code and data segments.
- Concurrent threads can execute on a single processor. Parallelism requires multiple processors.

Summary (Continued)

- *Turnaround* refers to completing a single a task in the smallest amount of time possible, whereas accomplishing the most tasks in a fixed amount of time refers to *throughput*.
- Decomposing a program based on the number and type of functions that it performs is called *functional decomposition*.
- Dividing large data sets whose elements can be computed independently, and associating the required computation among threads is known as *data decomposition* in multithreaded applications.
- Applications that scale with the number of independent functions are probably best suited to functional decomposition while applications that scale with the amount of independent data are probably best suited to data decomposition.
- *Race conditions* occur because the programmer assumes a particular order of execution but does not use synchronization to guarantee that order.

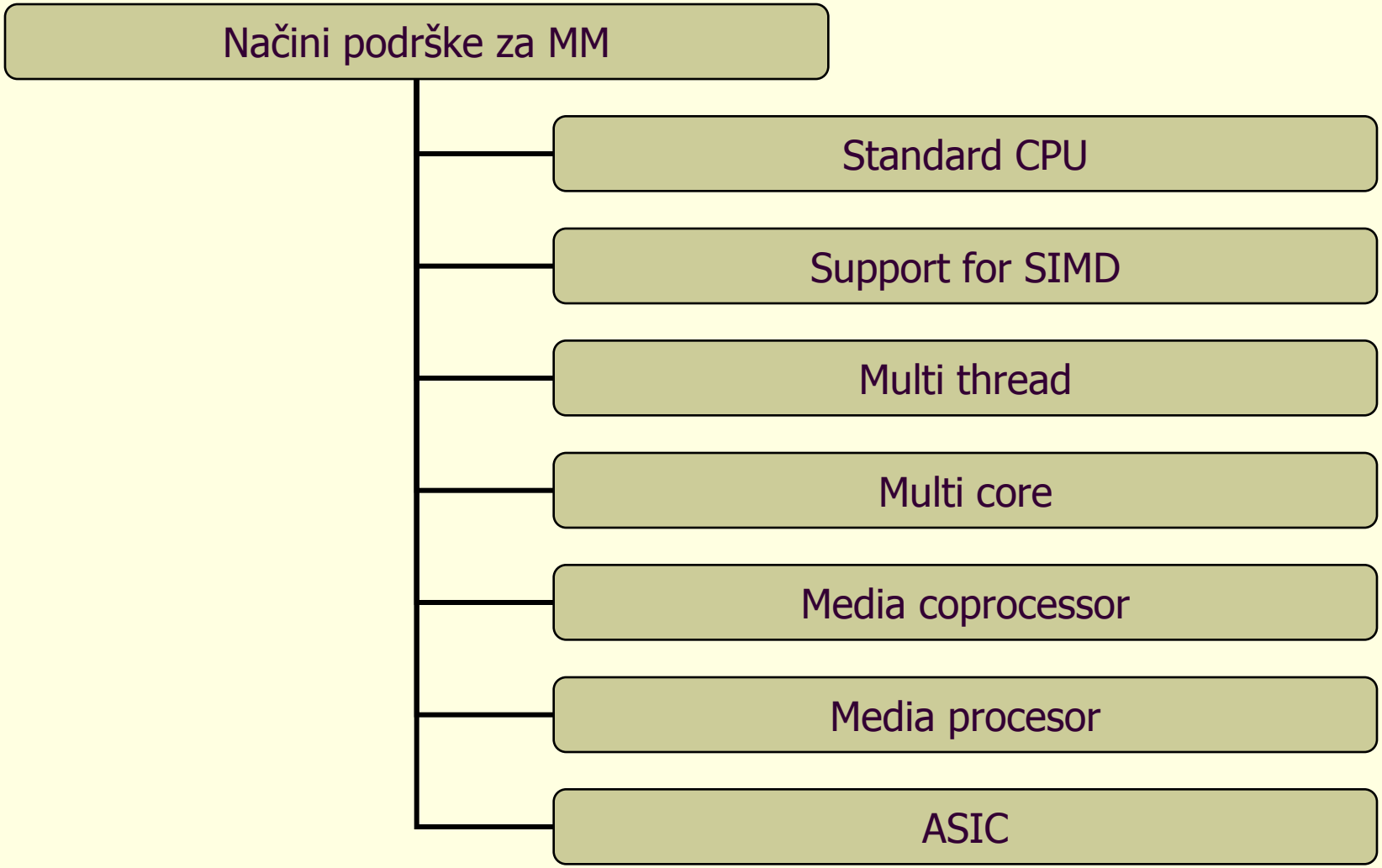
Summary (Continued)

- Storage conflicts can occur when multiple threads attempt to simultaneously update the same memory location or variable.
- *Critical regions* are parts of threaded code that access (read or write) shared data resources. To ensure data integrity when multiple threads attempt to access shared resources, critical regions must be protected so that only one thread executes within them at a time.
- *Mutual exclusion* refers to the program logic used to ensure single-thread access to a critical region.
- *Barrier synchronization* is used when all threads must have completed a portion of the code before proceeding to the next section of code.
- *Deadlock* refers to a situation when a thread waits for an event that never occur. This is usually the result of two threads requiring access to resources held by the other thread.

Summary (Continued)

- *Livelock* refers to a situation when threads are not making progress on assigned computations, but are not idle waiting for an event.
- *Speedup* is the metric that characterizes how much faster the parallel computation executes relative to the best serial code.
- *Parallel Efficiency* is a measure of how busy the threads are during parallel computations.
- *Granularity* is defined as the ratio of computation to synchronization.
- *Load balancing* refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.

Osnovni načini CPU podrške za MM



Standardni CPU

- Ako se prisjetimo Arhitekture računala onda znamo da obični procesori mogu izvesti jednu ALU operaciju po periodu
- Operacija je širine koju ima ALU

- Kako bi ubrzali izvođenje moramo mnogo pažnje posvetiti dohvatu podataka te optimizacijama petlji

Standardni CPU

- Za DSP operacije (npr DCT) izuzetno je korisno ako procesor ima sklopovski izvedeno množenje i pripadnu naredbu

- Dodatna značajna prednost ako postoji naredba množenja sa zbrajanjem (Multiply Accumulate, MLA)
  - Kod primjera računanja DCT, DFT i slično imamo većinu “leptir” operacija kod kojih je MLA osnovna karika

Standardni CPU ubrzanja

- Efikasno korištenje protočne strukture
  - Loop-unrolling
  - Branch prediction
- ....
- Nedovoljno !

SIMD

- Analizom mnogih aplikacija ustanovljeno:
  - Koje aplkacije su najzahtjevnije
  - Gdje je efikasnost najmanja
  - Koja su uska grla
  - ....
- Povećanje brzine nije rješenje
  - P4 fijasko
    - (enormni pipeline: energija, toplina, branch miss)
  - ARH1 !!

SIMD

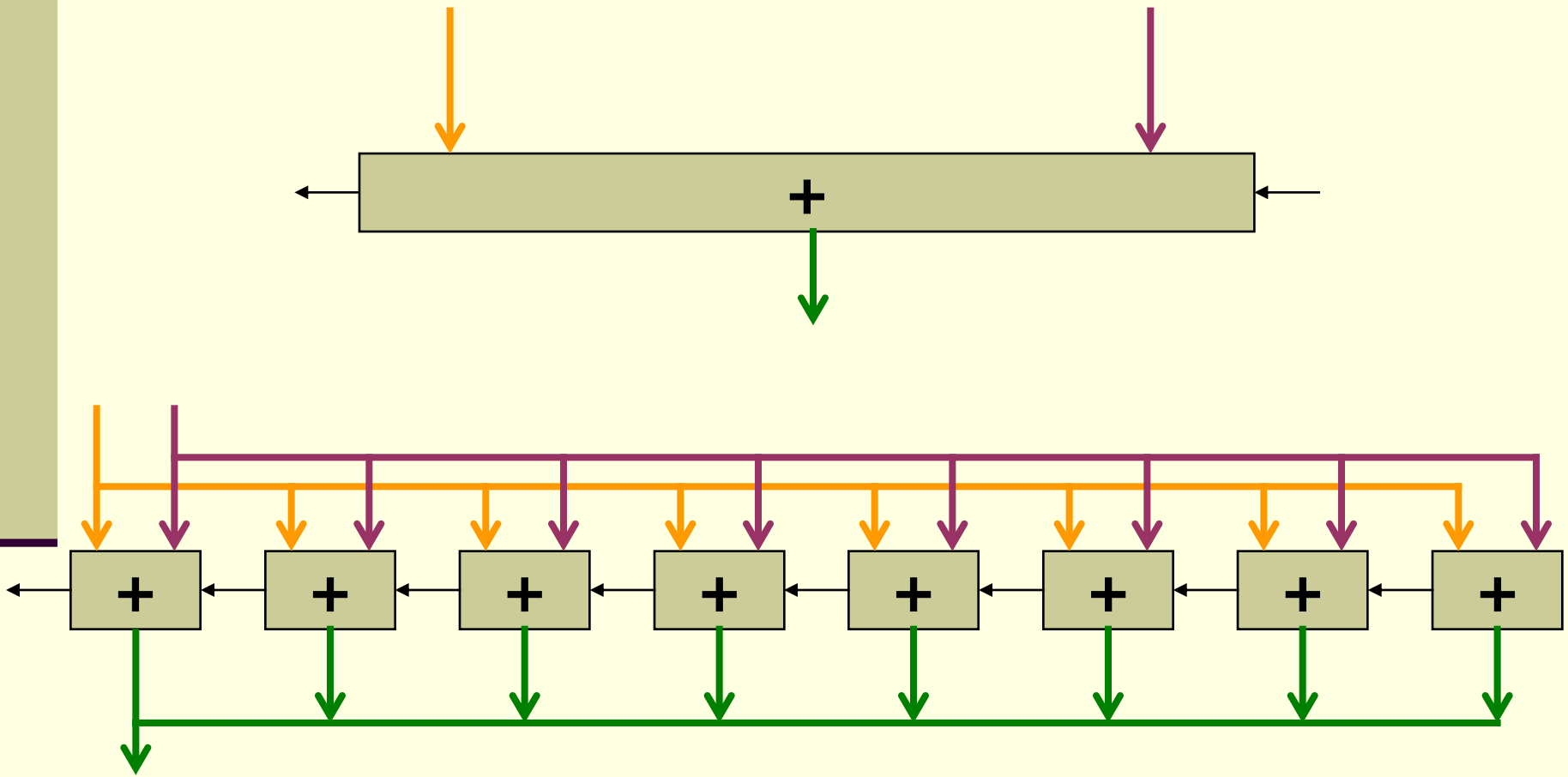
- Ispada da su najzahtjevnije aplikacije koje koriste vrlo jednostavne operacije i operande
  - 8 bita (video)
  - 16/32 bita audio
- ... Ali puuuuuno podataka

Podrška za SIMD

- SIMD (Single Instruction Multiple Data)
  - Arhitektura puta podataka u procesoru koja omogućuje obradu više podataka (u načelu manje preciznosti) sa jednom naredbom

- Osnovna ideja:
  - Ako imamo npr 64 bitovnu ALU onda je potpuno neefikasno s njom obrađivati 8 bitovne podatke
  - Reorganizirati ALU na način da se može “podijeliti”

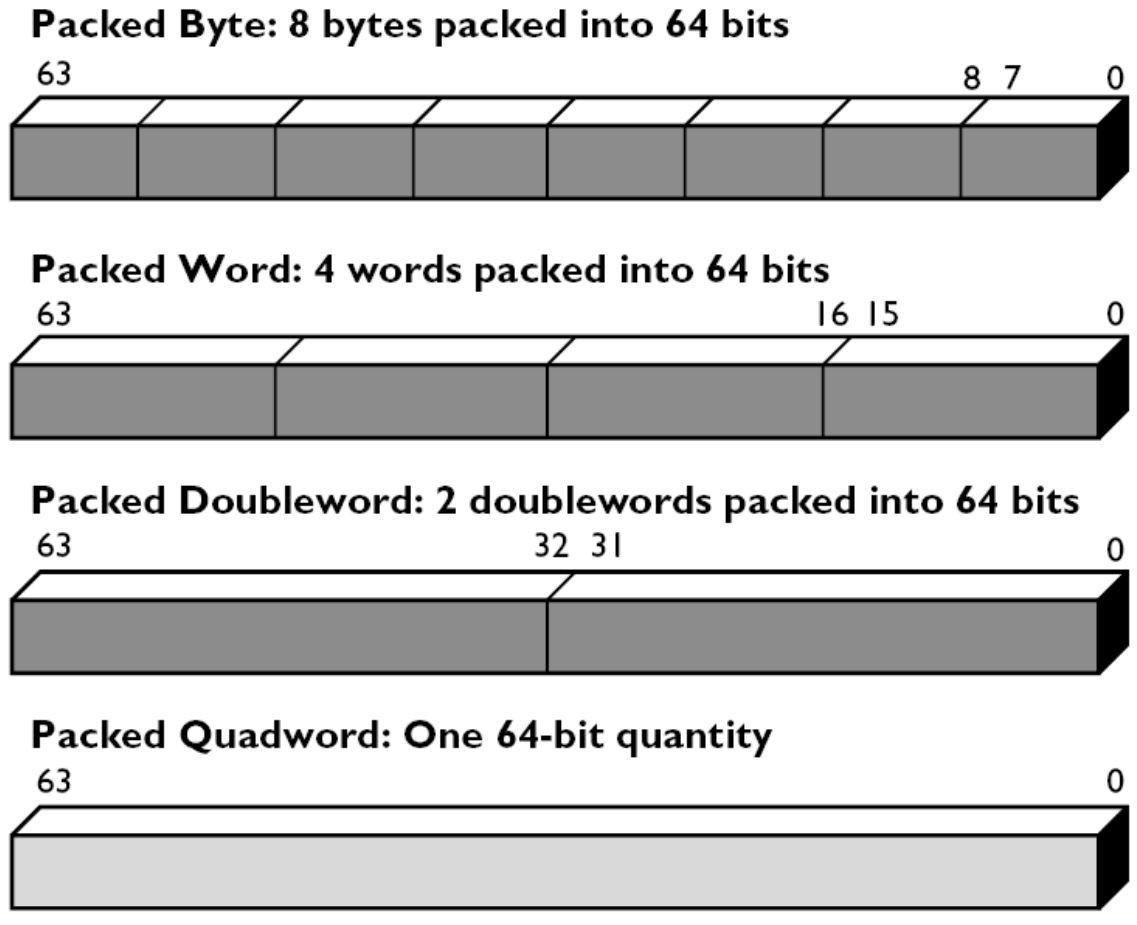
ALU – obična i SIMD



Najpoznatiji primjeri

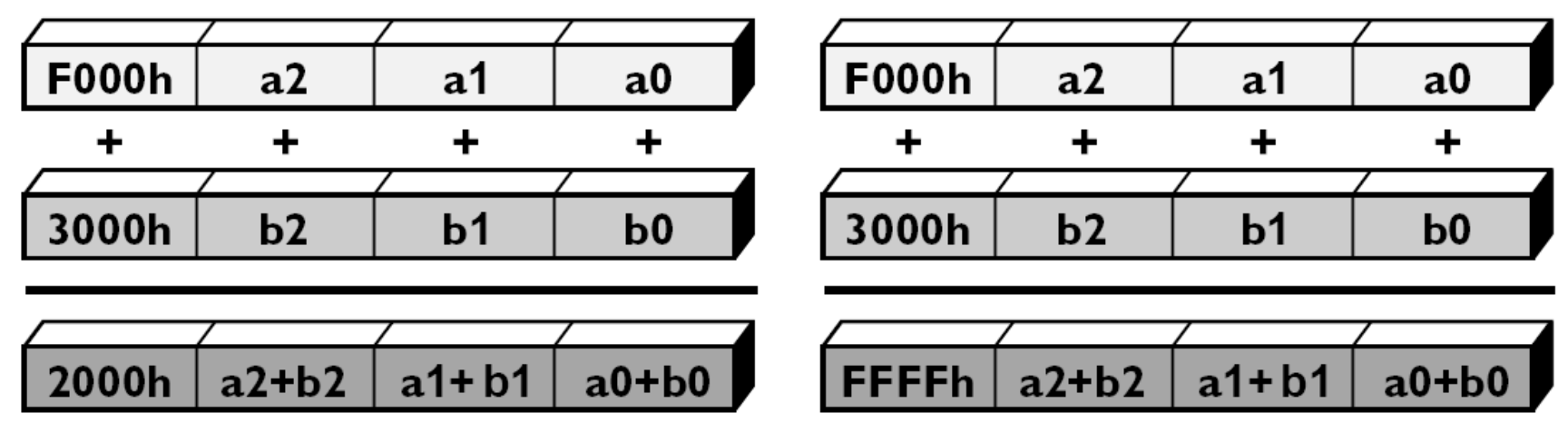
- MMX
- Stavljen na tržište 1996 (Pentium sa MMX i Pentium II)
- Uveden novi tip podataka: Packed 64 bit
- Zašto 64 bita?
  - zadovoljavajuće ubrzanje
  - Ne treba velike zahvate na arhitekturi

MMX tipovi podataka



Aritmetika sa zasićenjem

- Izuzetno korisno kod MM algoritama
- Značajno poboljšava performanse



wrap-around saturating

Rezultati...

- Inicjalno velika medijska “buka” ali rezultati nisu zadovoljili na dulje vrijeme..
- Ubrzanja su dobra u odnosu na uloženo ali su uočene potrebe za poboljšanjem
- Nedostaci:
  - Ne mogu se koristiti paralelno FPU i INT
  - Nedovoljna podrška za 8 i 32 bitovne podatke
  - Samo integer aritmetika
  - OS nije problem osvježiti

3DNow!

- AMD-ova proširenja MMX-a
  - Predstavljena 1998.
  - Podrška za 32 bitovni FP
  - Neke od ovih naredaba Intel uveo u SSE
  - Kasnije prošireno na 3DNow!+

SSE

- Streaming SIMD Extensions
- Predstavljeno 1999 sa Pentium III

- Popravljeni najvažniji nedostaci MMX-a
- 8 potpuno novih 128 bitovnih registara XMM0..XMM7 (još 8, ..XMM15 u 64-bit)
- Omogućena 128-bit (4x32bit) SIMD FP podrška: aritmetika, usporedbe, shuffle,...
- Integer SIMD podrška nad 64 bita
- Preko 60 novih naredbi

SSE nedostaci

- Spremanje stanja registara loše izvedeno
- Resursi za izvođenje zajednički sa FPU

SS.....

- SSE2
  - Pentium4(2001), 144 nove naredbe, 64 bitovni FP, proširenje MMX naredbi radi i sa XMM, performanse nisu naročito bolje od MMX
- SSE3
  - Pentium4(2004), horizontalne operacije, konverzije FP-INT jednostavnije,
- SSSE3
  - Intel Core, 16 novih naredaba, AMD ne podržava

SSE4

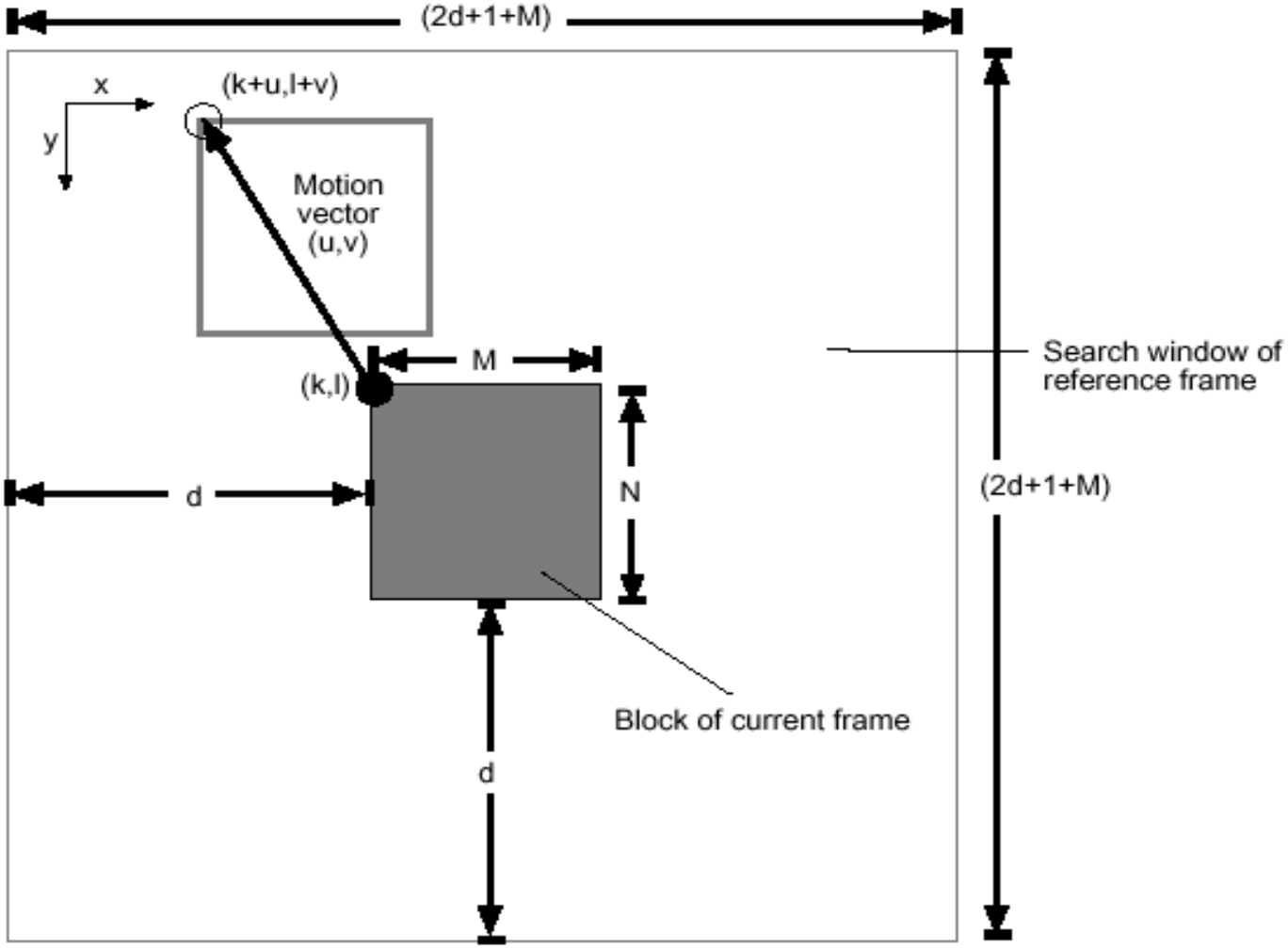
- 54 nove naredbe
  - SSE4.1 (47), SSE4.2 (7)
- Bolja implementacija nekih operacija

SIMD Zaključak

- SIMD proširenja donose značajna poboljšanja performansi
- Nažalost, korištenje SIMD zahtjeva ogromno znanje i puuuno vremena ali rezultira izuzetno efikasnim kodom
- Nove inačice SIMD dovode do sve boljih rezultata

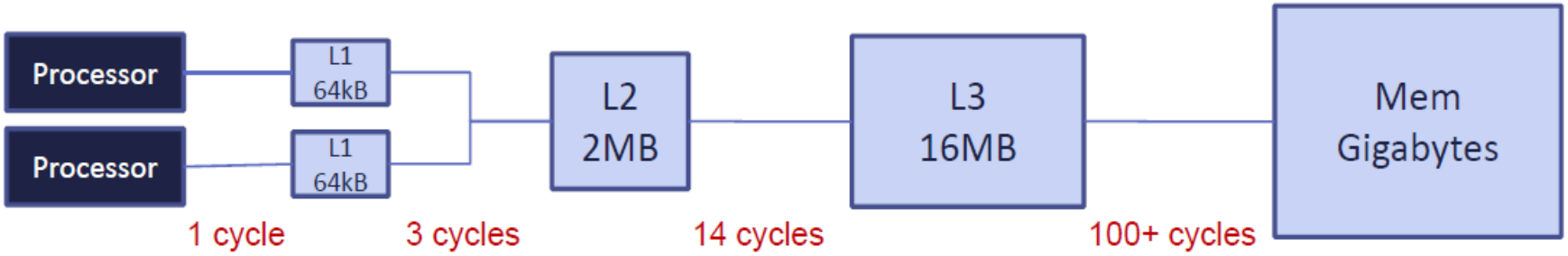
- No, i SIMD ima svojih granica .... Što dalje?

Algoritam potpunog pretraživanja



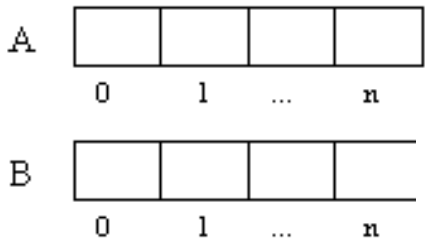
Standardni CPU – Memorijski sustav

- Memorijska hijerarhija
- Priručna memorija – najmanja i najbrža
- Ako uzorak pristupa programa odgovara heuristici sklopovlja za pristup cache-u: velika brzina



SIMD intrinsics (3)

$$\sum_{k=0}^{n} A[k] * B[k]$$



C = A[0] * B[0] + A[1] * B[1] + ... + A