Heterogén számítási rendszerek

Alapfogalmak, vektorizáció, OpenMP

Szántó Péter

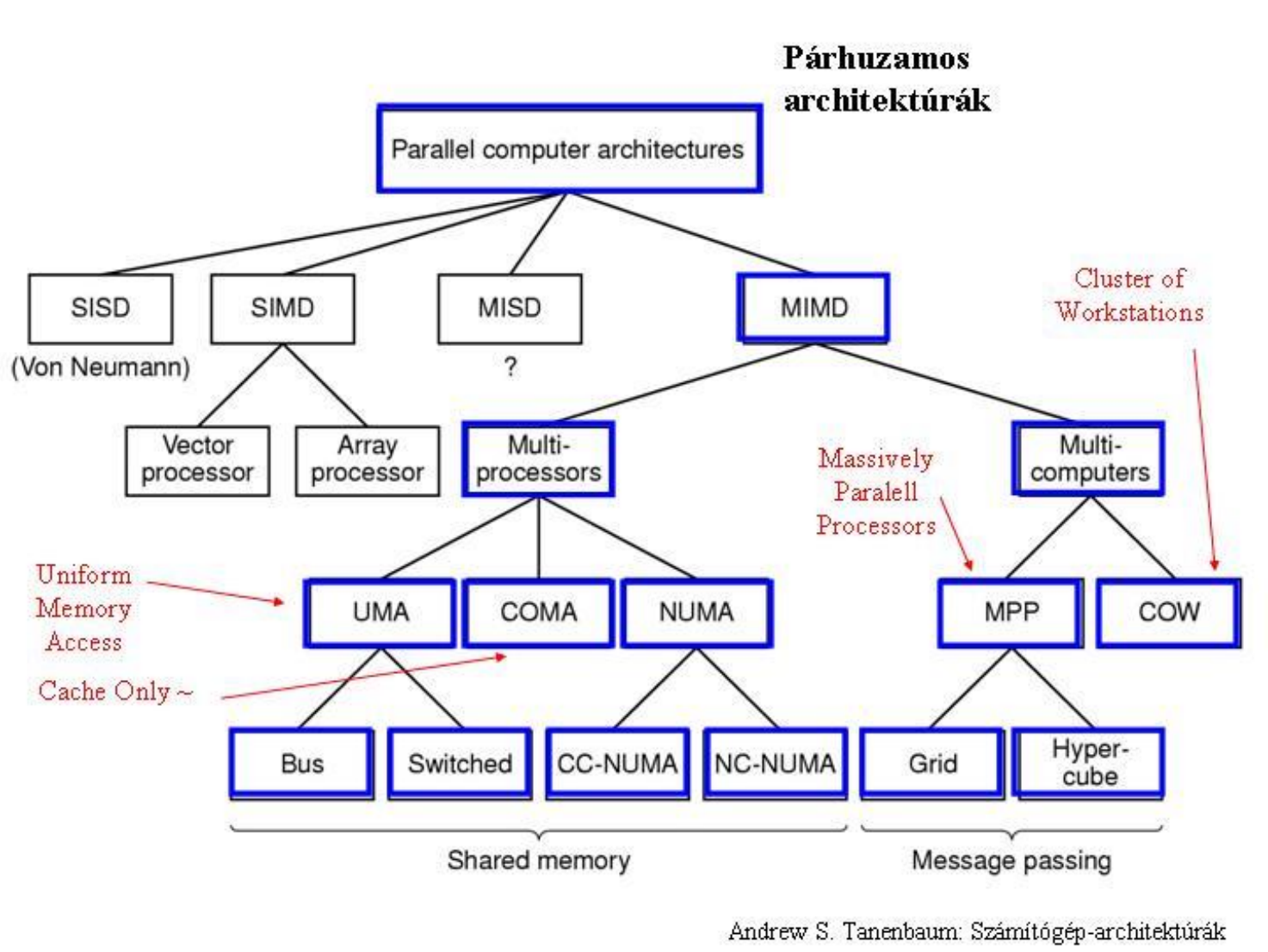
BME MIT, FPGA Laboratórium

Alapfogalmak

Szántó Péter

BME MIT, FPGA Laboratórium

Párhuzamos architektúrák



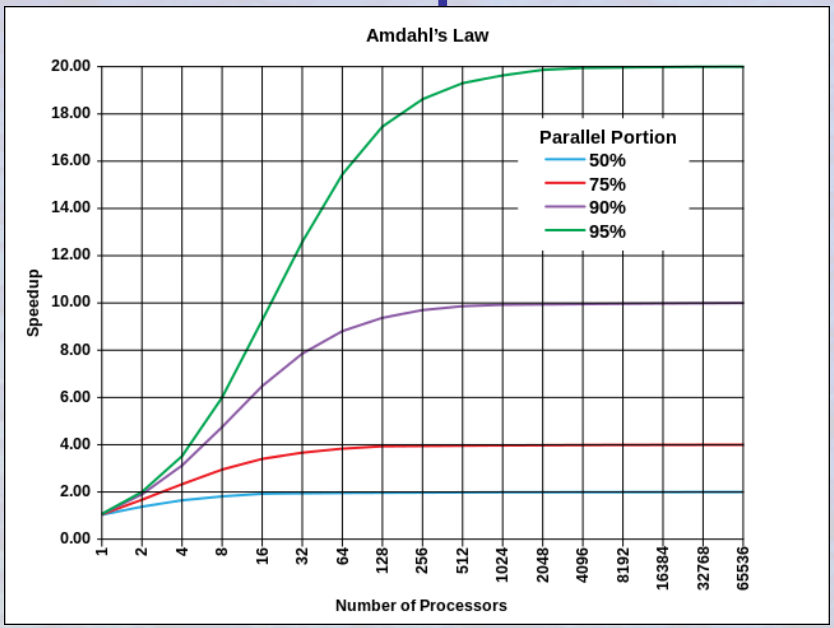
Andrew S. Tanenbaum: Számítógép-architektúrák

Amdahl törvény

- **Gyorsíthatunk, de…..**
  - B: Az algoritmus sorosan végrehajtandó része (0…1)
  - n: a párhuzamosítható részek gyorsítási szorzója

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

- **Még ha az algoritmus 95%-a párhuzamosítható is….**

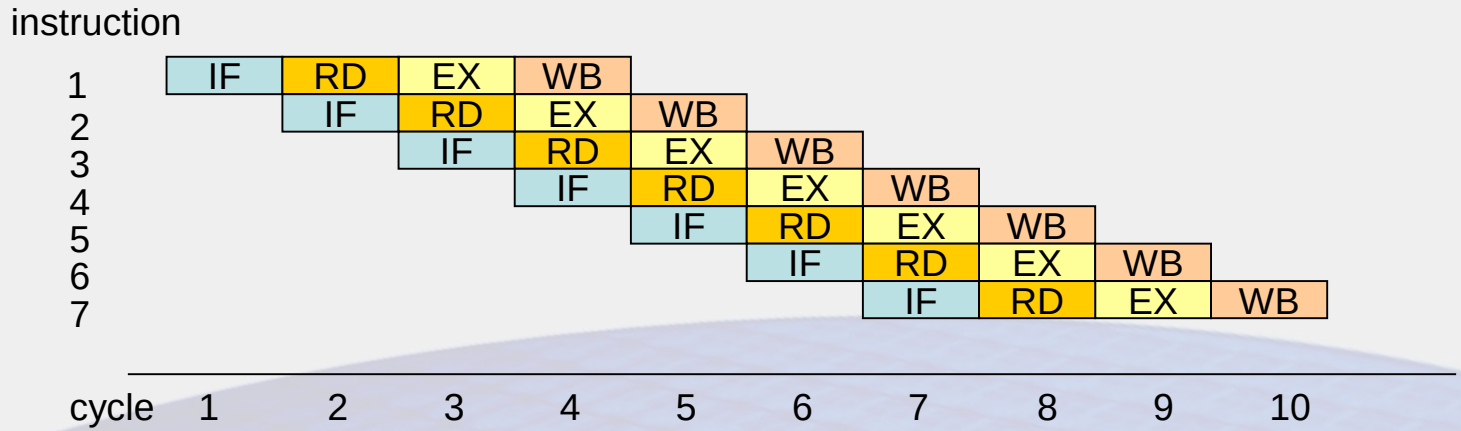


Alapfogalmak

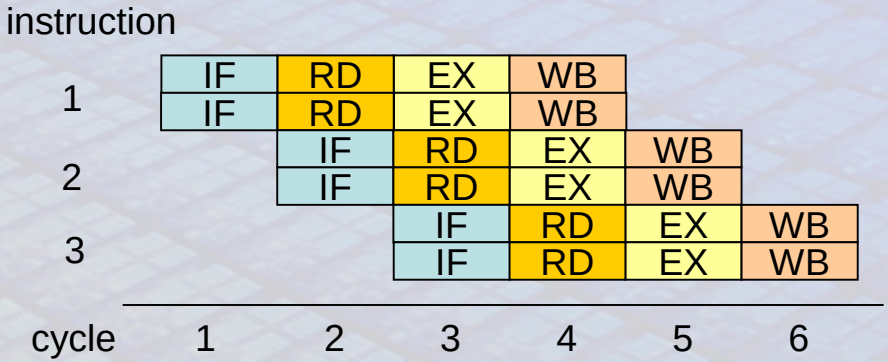
- **Symmetric MultiProcessing**
  - Minden mag látja a teljes memóriát
  - Minden mag teljes hozzáféréssel rendelkezik a memóriához
  - Tipikusan max. ~32 mag
- **Non-Uniform Memory Architecture (NUMA)**
  - Minden processzor saját memóriával rendelkezik, a processzoronkénti sávszélesség jóval nagyobb
  - A memória elérés függ a processzor – memória távolságtól
  - **ccNUMA**: cache coherent NUMA

Alapfogalmak

- **Pipeline végrehajtás (IF-ID-EX-WB)**



- **Szuperskalár processzor**
  - Több utasítás fetch órajelenként (!= pipeline)



Alapfogalmak

- **In-order utasítás végrehajtás**
  - Az assembly utasítások a kódnak megfelelő sorrendben hajtódnak végre (fetch – wait – execute - write)
- **Out-of-order (OoOE) utasítás végrehajtás**
  - A processzor átrendez(het)i az utasítások sorrendjét
  - Fetch - dispatch (wait) – execute – queue – write
- **Elágazás becslés**
  - Statikus (pl. backward igen, forward nem)
  - Dinamikus (pl. szaturációs számláló)
    - Pl. 2 biten 0…3 között számoljuk, hogy hányszor ugrottunk → Becslés: >= 2 esetén ugrás
    - Több számláló, PC néhány bitje választja ki, hogy adott elágazáshoz melyik tartozik

Alapfogalmak

- **SISD**
  - Single Instruction Single Data
- **SIMD**
  - Single Instruction Multiple Data
  - Intel MMX, SSEx, AMD 3DNow!, Intel AVX, Power AltiVec, ARM NEON
- **MIMD**
  - Multiple Instruction Multiple Data
  - Pl. GPU
- **VLIW: Very Long Instruction Word**
    - Utasítás szintű párhuzamosítás fordítási időben

Alapfogalmak (MT)

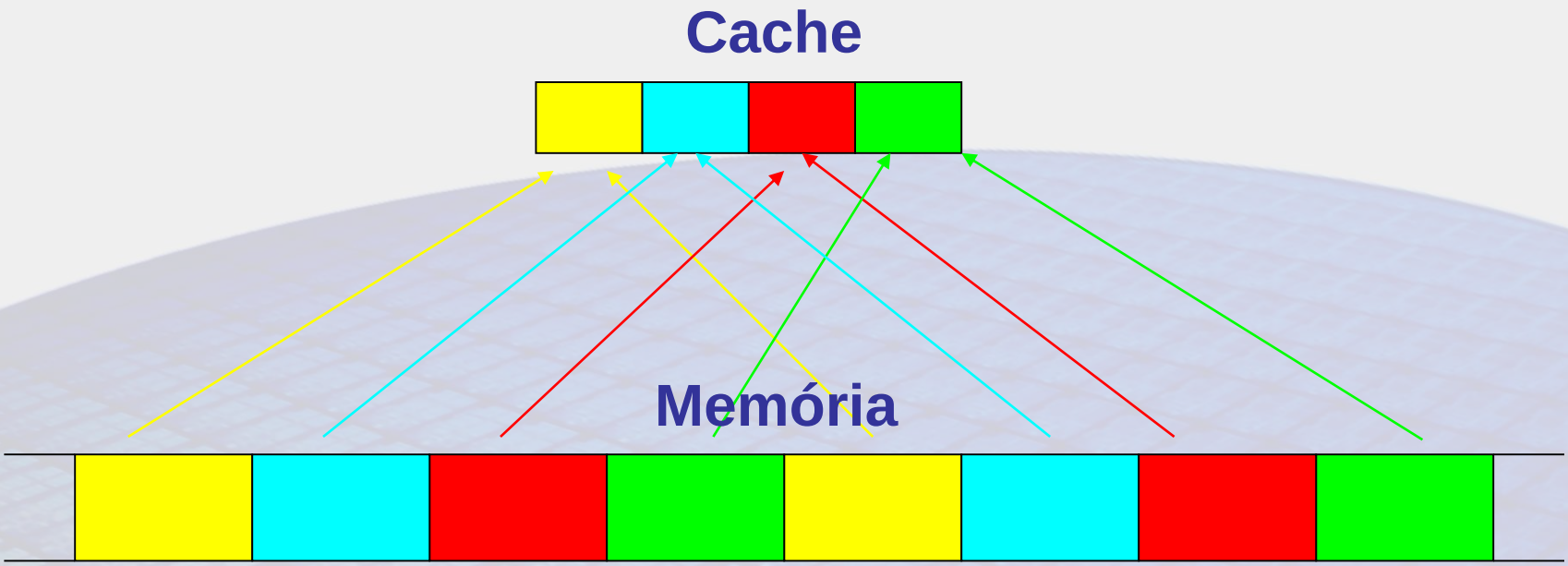
- **Többszálúság (multi-threading)**
  - **Temporal MT**: az egymást követő utasítások származhatnak más szálból
  - **Simultaneous MT**: a processzor feldolgozó egységei ugyanazon órajelben más-más szállal foglalkoznak (szuperskalár)
  - **Multi-core**: több, teljes processzor mag egy szilíciumon
- **Architektúrák**
  - Intel HT: SMT, 2 szál
  - IBM POWER5, POWER6: SMT, 2 szál
  - IBM POWER7-9: SMT, 4 szál
  - Sun Niagara: TMT, 8 szál

CPU Cache

- **Miért kell?**
  - DRAM hozzáférés tulajdonságai
  - Kis késleltetés
- **Cache-ben:**
  - Tárolt memória cím (TAG) – cache találat megállapítás
  - Adat – cache line
- **Memória írás**
  - Write-through: minden cache írás hatására frissül a RAM-ban az adat
  - Write-back: adat kiírás a cache frissítése esetén
- **Cache koherencia**
  - Több CPU mag, DMA képes HW (cache invalidálás)

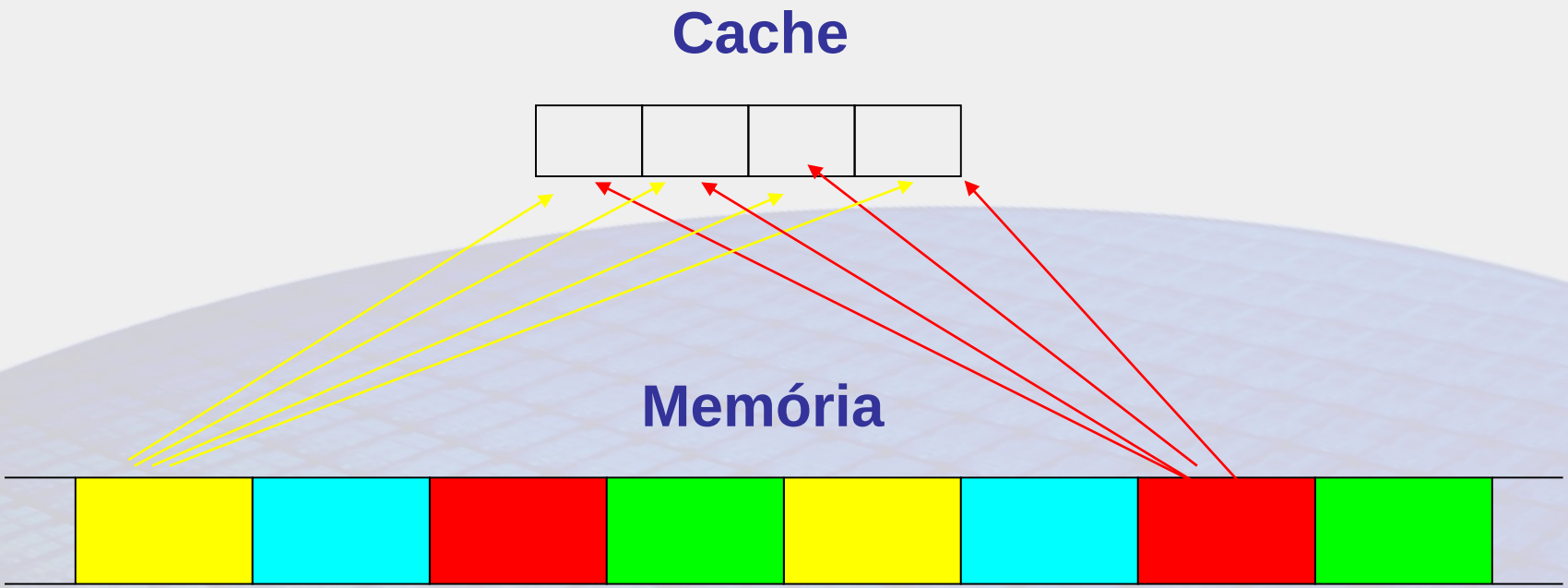
Direct-Mapped Cache

- A memória minden egyes címe pontosan egy Cache blokkban lehet cache-elve



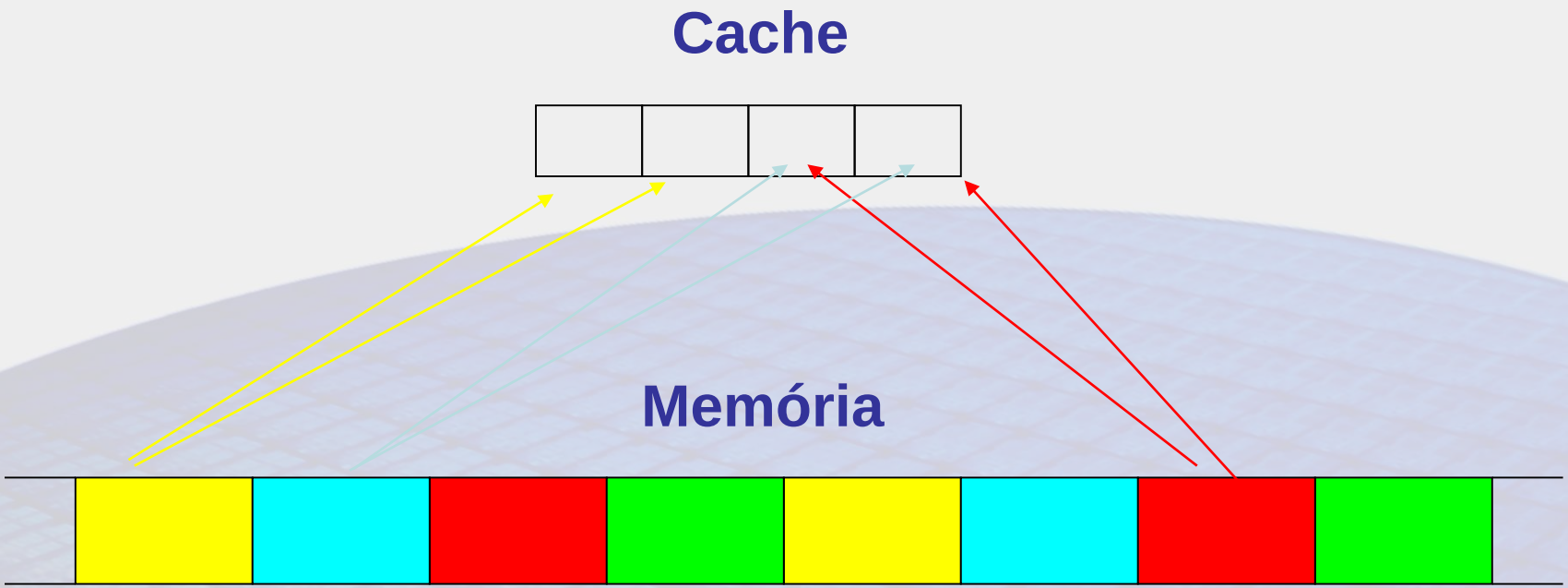
Full-Associative Cache

- **Bármelyik memória a Cache bármelyik blokkjában lehet cache-elve**



Set-Associative Cache

- N-way associative (N utas): a memória adott címe N cache-blokkban lehet cache-elve



- Pl. Core i7: 8-way L1 & L2 cache; 16-way L3

Vektorizáció

Szántó Péter

BME MIT, FPGA Laboratórium

SIMD utasítások (1.)

- Single Instruction Multiple Data
- Ugyanazt a műveletet több adaton hajtjuk végre
- Tipikusan:

m1	=	f1	f2	f3	f4
+		+	+	+	+
m2	=	f5	f6	f7	f8
=		=	=	=	=
m1+m2	=	f1+f5	f2+f6	f3+f7	f4+f8

SIMD utasítások (2.)

- Vannak kivételek
- Pl. horizontális FP összeadás:

  dst[31:0] := a[63:32] + a[31:0]

  dst[63:32] := a[127:96] + a[95:64]

  dst[95:64] := b[63:32] + b[31:0]

  dst[127:96] := b[127:96] + b[95:64]
- Pl. permutálás

  dst[31:0] := SELECT4(a[127:0], imm8[1:0])

  dst[63:32] := SELECT4(a[127:0], imm8[3:2])

  dst[95:64] := SELECT4(a[127:0], imm8[5:4])

  dst[127:96] := SELECT4(a[127:0], imm8[7:6])

SIMD utasításlészletek

- **x86/x64, majd részletesebben**
  - 3DNow!, MMX
  - SSEx, AVX1, AVX2, AVX512
- **ARM**
  - Neon, Scalable Vector Extension (SVE)
  - 128 bites vektor: 8, 16, 32, 64 bites egész; 32, 64 bites float
- **PowerPC**
  - AltiVec = VMX
  - 128 bites vektor: 8, 16, 32 bites egész komponensek; 32 bites float

Vektorizáció

- **Inline assembly**
  - Jobbára minden fordító támogatja (kivéve VS x64 ☹)
- **Intrinsic („beépített függvény”)**
  - Fordító specifikus
- **Automatikus vektorizáció**
  - GCC4: gyakorlatilag minden architektúrára

```
gcc main.cpp -o main.exe -O2
   -ftree-vectorize -msse2
   -ftree-vectorizer-verbose=5
```

  - ICC: Intel
  - Visual Studio 2012-től (automatikus, report: **/Qvec-report:2)**

SSE utasítások

- **70 új utasítás**
- **8, ill. 16 regiszter (x86 ill. x64 esetén)**
- **Aritmetikai**
  - SP: add, sub, mul, div, rcp, min, max
  - Egész: avg, sad, min, max
- **Logikai**
  - and, nand, or, xor
- **Komparálás – FP**
- **Konverzió – INT/FP**
- **Adatmozgatás (mov, load, store – alignment!)**
- **Cache fetch**

SSE2/SSE3/SSE4

- **SSE2: 144 utasítás (Intel P4, AMD Opteron/Athlon64)**
  - DP & INT8/INT16/INT32 aritmetikai, logikai és komparálás utasítások
  - Shuffle, interleave
- **SSE3, SSSE3 (13 + 16 utasítás)**
  - Horizontális aritmetika, pl. HADDPS (*Horizontal-Add-Packed-Single*)
    - Bemenet: R0: A0, A1, A2, A3; R1: B0, B1, B2, B3
    - Kimenet: A0 + A1, A2 + A3, B0 + B1, B2 + B3
  - MAC (INT8)
- **SSE4.1/4.2/4a**
  - INT32 MUL, POPCNT
- **Utasítás lista: http://softpixel.com/~cwright/programming/simd/index.php**

AVX utasításkészlet

- **Advanced Vector Extensions**
  - 16 db 256 bites regiszter (YMM0…YMM15)
  - SSE utasítások az alsó 128 bitet használják
  - 3 operandusú utasítások (SSE: 2)
- **CPU**
  - Intel Sandy/Ivy Bridge (2. & 3. gen. Core iX)
- **Compiler:**
  - GCC v4.5
  - Intel C Compiler 11.1
  - VS 2010+
- **Operációs rendszer**
  - Linux kernel 2.6.30+, Windows 7+

AVX2, AVX-512

- **AVX2**
  - Jobbára minden SSE utasítás elérhető 256 biten
    - Pl. sokkal szabadabb permutálás
    - Gather utasítások
    - Integer aritmetika (8, 16, 32 bit)
- **AVX-512**
  - 512 bites vektorok
  - 32 db 512 bites regiszter
  - Alapvetően a Xeon Phi gyorsító utasításkészlete
  - Skylake alapú E5/E7 Xeon-ok támogatják

SSE Intrinsic-ek

- xmmintrin.h: SSE + MMX (P3, Athlon XP)
- emmintrin.h: SSE2 (P4, Ahtlon 64)
- pmmintrin.h: SSE3 (P4 Prescott, Ahtlon 64 San Diego)
- tmmintrin.h: SSSE3 (Core 2, Bulldozer)
- popcntintrin.h: POPCNT (Core ix, Phenom)
- ammintrin.h: SSE4A (Phenom)
- smmintrin.h: SSE4_1 (Core ix, Bulldozer)
- nmmintrin.h: SSE4_2 (Core ix, Bulldozer)
- wmmintrin.h: AES (Core i7 Westmere, Bulldozer)
- immintrin.h: AVX (Sandy Bridge, Bulldozer)

Speciális adattípusok (VS)

- **SSE Intrinsic**
  - __m128i: 4xInt/8xShort/16xChar
  - __m128: 4xFP
  - __m128d: 2xDP
- **AVX Intrinsic**
  - __m256i: 8xInt/16xShort/32xChar
  - __m256: 8xFP
  - __m256d: 4xDP
- **Assembly**
  - SSE: xmm0…xmm15 regiszterek
  - AVX: ymm0…ymm15 regiszterek

SSE load/store

- **Unaligned**
  - MOVUPS – __mm_loadu_ps
- **16-byte aligned**
  - MOVAPS – __mm_load_ps
- **0. szó betöltése, 1-3 szavak törlése**
  - MOVSS – __mm_load_ss
- **Store**
  - MOVUPS – __mm_storeu_ps; unaligned
  - MOVAPS – __mm_store_ps; aligned
  - MOVSSS – __mm_store_ss (csak 0. szó)
- **Aligned malloc**

```
cA = (float*)(_aligned_malloc(size, 16));
```

SSE cache függvények

- **Prefetch („előolvasás”)**
  - PREFETCH – __mm_prefetch
- **Store a cache felülírása nélkül**
  - Nagyon hatékony lehet, ha a kimenő adatot nem olvassuk vissza
  - MOVNTQ – __mm_stream_ps

SSE ALU utasítások

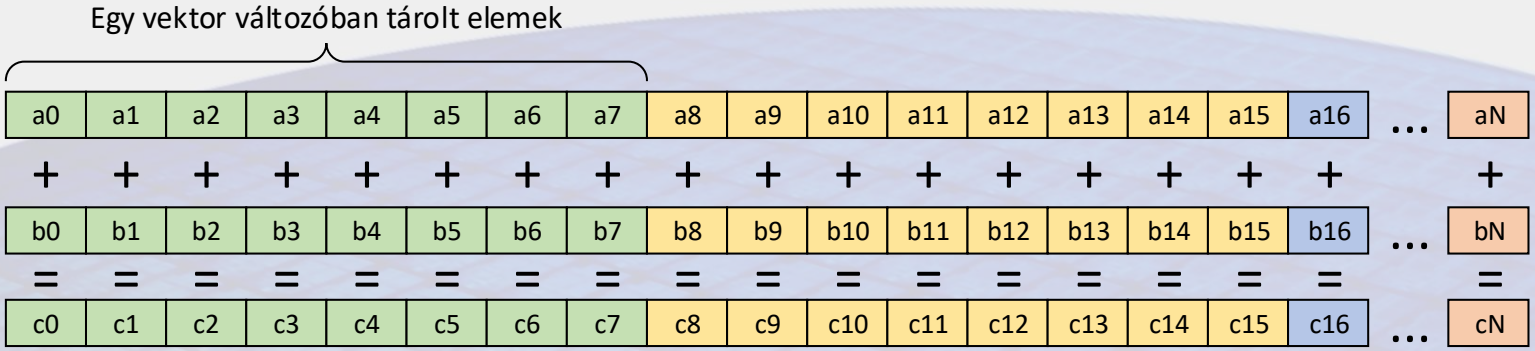
- **32 bites FP aritmetikai műveletk**
  - ADDPS – __mm_add_ps
  - MULPS – __mm_mul_ps
  - MINPS – __mm_min_ps, …..
- **Logikai műveletek**
  - ANDPS – __mm_and_ps
  - ORPS – __mm_or_ps, …..
- **Shuffle**
  - SHUFPS
  - __mm_shuffle_ps(m1, m2m __MM_SHUFFLE(1,0,3,2)))

Vektor összeadás (1)

- **Vektorok összeadása:**

$$\mathbf{C} = \boldsymbol{A} + \boldsymbol{B} = (\boldsymbol{a_0} + \boldsymbol{b_0}, \boldsymbol{a_1} + \boldsymbol{b_1}, \ldots \boldsymbol{a_{N-1}} + \boldsymbol{b_{N-1}})$$

- **Triviálisan vektorizálható**



- **Egy vektor utasítással 8 elemet adunk össze (pl. AVX, float elemek)**

Vektor összeadás (2)

- C kód

```
float a[N], b[N], c[N];
for (i=0; i<N; i=i+1){
   c[i]=a[i]+b[i];
}
```

```
float *a, *b, *c;
for (i=0; i<n; i=i+1){
   *(c+i)=*(a+i) + *(b+i);
}
```

- Vektorizáció AVX intrinsic-ekkel

```
float *a, *b, *c;
a = (float*)(_aligned_malloc(N * sizeof(float), 32));
b = (float*)(_aligned_malloc(N * sizeof(float), 32));
c = (float*)(_aligned_malloc(N * sizeof(float), 32));
for (i=0; i<N; i=i+8){
   av = _mm256_load_ps(a+i);
   bv = _mm256_load_ps(b+i);
   cv = _mm256_add_bps(av, bv);
   _mm256_store_ps(c+i, cv);
}
```

Elágazások

- **Single Instruction -> NEM lehet az egyes vector elemekben más-más helyre ugrani!**

```
int i, a[8];
for (i=0; i<8; i++){
    if (a & 1)
        a = a>>1;
    else
        a = ~(a>>1);
}
```

~x86 ASM

```
and     eax, 1
test    eax, eax
je      LN5
mov     eax a[i]
sar     eax, 1
jmp     LN6
$LN5    mov     eax, a[i]
        sar     eax, 1
        not     eax
$LN6
```

AVX2

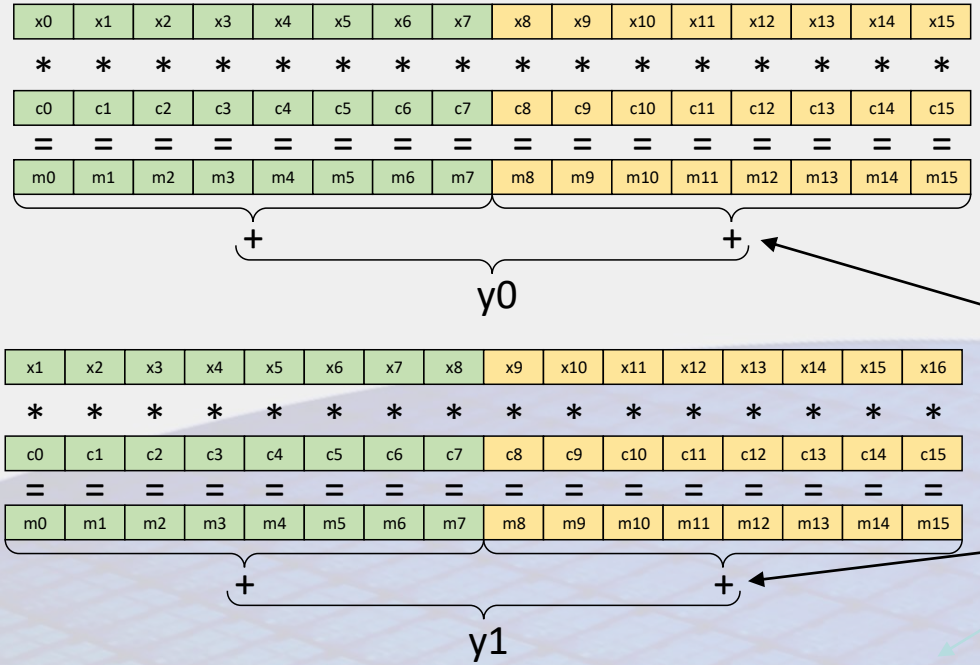
```
__mm256i a, sh, nsh, mask, nmask;
sh = _mm256_srli_epi32(a);
nsh = _mm256_xor_si256(sh, sh);
c1 = _mm256_set1_epi32(1);
mask = _mm256_and_si256(a, c1);
mask = _mm256_cmpeq_epi32(and, c1);
nmask = _mm256_xor_si256(mask, mask);
mask = _mm256_and_si256(mask, sh);
nmask = _mm256_and_si256(nmask, nsh);
a = _mm256_or_si256(mask, nmask);
```

1D konvolúció (1)

- **Konvolúció**
  - $y_k = \sum_{i=0}^{N-1} x_{k+i} * c_i$
- **Vektorizációs lehetőségek:**
  - Egyetlen kimenet rész-számításait vektorizáljuk
  - Egy utasítással párhuzamosan több kimenetet számolunk

1D konvolúció (2)

- **Egy kimenet számítása**

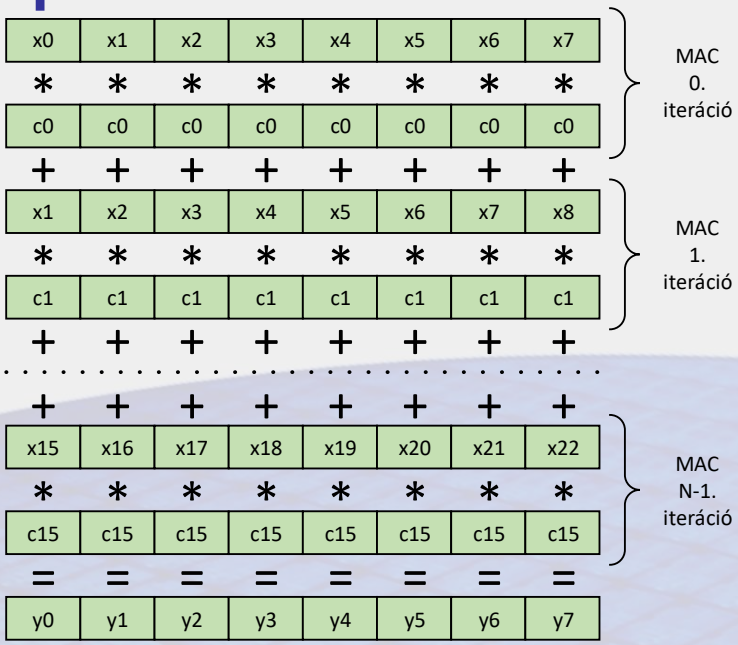


Vektor elemek horizontális összegzése

- **Horizontális összeadás kevéssé vektorizálható**
  - Páronkénti elemek összeadása általában van, de ez kevés, ez a rész esetleg lehet szekvenciális
- **Bemeneti vektorok: unaligned töltés vagy shuffle, permute, rotate jellegű utasítások**

1D konvolúció (3)

- Több kimenet párhuzamos számítása: N=16



- NINCS horizontális művelet, a párhuzamosan számított eredmények egy vektort alkotnak
- Bemeneti vector: ugyanaz a probléma, mint előbb
- Együttható többszörözés: vagy konstans vagy shuffle/permute utasítással

1D konvolúció (4)

- **Bemeneti adattömb “shiftelése”**
  - unaligned load: AVX esetén _mm256_loadu_ps()
    - Tipikusan lassabb, mint az aligned
    - Minden utasításkészletben van
  - Vektor utasításokkal
    - Permute, shuffle általában van, pl. AVX2:
      - _mm256_permutevar8x32_ps (__m256 a, __m256i idx)

```
for i=0…7
   src = idx[32*i+2:32*i]
   dst[i*32+31:i*32]=a[src*32+31:src*32]
```

    - AVX2-ben, AVX256-ban rotate jellegű utasítások is vannak (_mm256_alignr_*)

1D konvolúció (5)

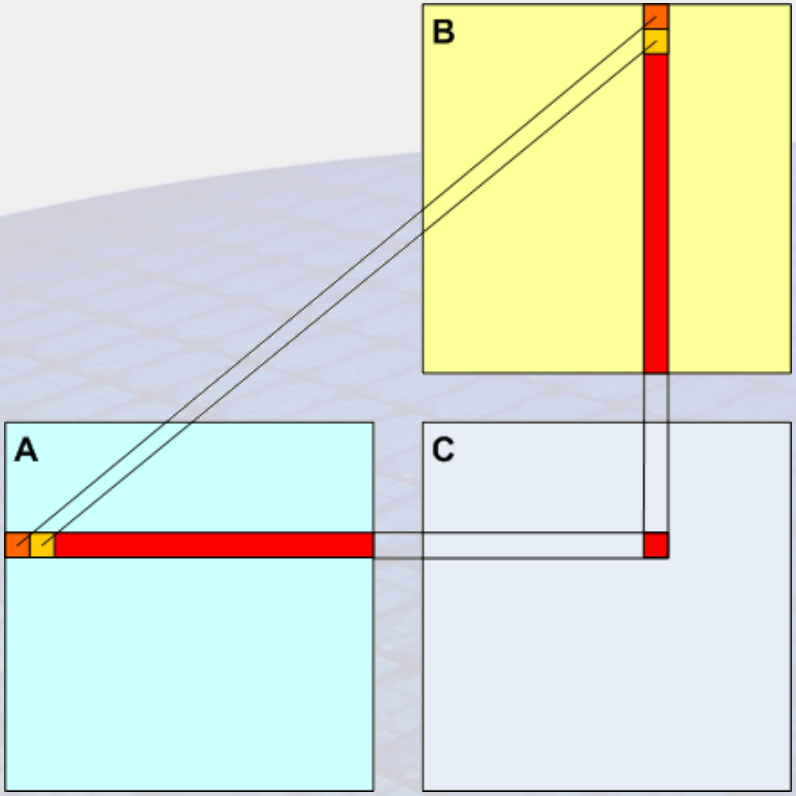
- Egy lehetséges megoldás bemeneti vector egy elemmel történő “shiftelése”

```
// Két egymás melletti vector betöltése
op0 = _mm256_load_ps(a + 0);
op1 = _mm256_load_ps(a + 8);

// 0. vektor shiftelése egyel (0. elem nem változik)
idx = _mm256_set_epi32(6, 5, 4, 3, 2, 1, 0, 0);
res = _mm256_permutevar8x32_ps(op0, idx);

// “Multiplexálás” a blend utasítással
mask = _mm256_set_ps(0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0);
res = _mm256_blendv_ps(res, op1, mask);
```

- Unaligned load lehet gyorsabb….

Mátrix szorzás

- **Egy kimenet előállítása:**
  - A egy során, B egy oszlopán lépkedünk végig
  - A megfelelő részszorzatokat összegezzük

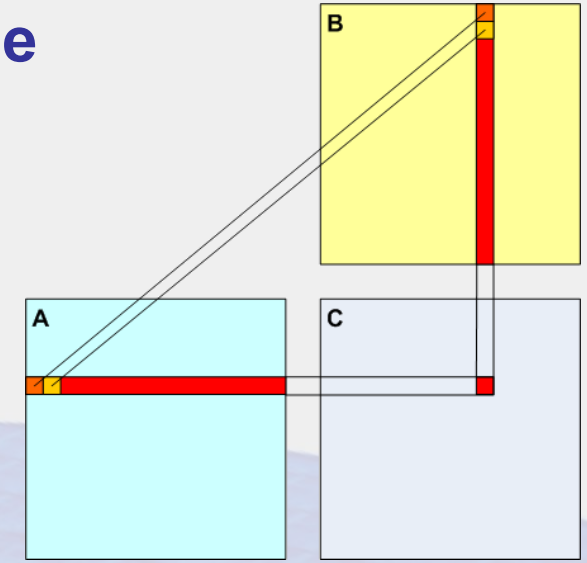


Mátrix szorzás (x86)

```
for (row=r_start; row<r_end; row=row+1){
    for (col=0; col<N; col=col+1){
        float* a_l = a_base + row*N;
        float* b_l = b_base + col;
        float* c_l = res_base + row*N + col;

        float sum = 0;
        for (i=0; i<N; i++){
            sum = sum + *(a_l) * *(b_l);
            a_l = a_l + 1;
            b_l = b_l + N;
        }
        *(c_l) = sum;
    }
}
```

Mátrix szorzás (SSE intrinsic)

- **SSE: 4 db lebegőpontos szám kezelése „egyszerre”**
- **Praktikusan egyszerre 4 kimenetet számolunk (ezek a memóriában címfolytonosan helyezkednek el)**
- **Ehhez B-ből folyamatosan 4 egymás melletti értéket olvasunk**



- **Egy rész-szorzat számításhoz A-ból csak egy érték szükséges**
  - A-ból negyed olyan gyakran 4 egymás meletti értéket töltünk

Mátrix szorzás (SSE intrinsic)

```
for (col=0; col<N; col=col+4)
{
    a_addr = a_base + N*row;
    b_addr = b_base + col;

    r_sse = _mm_set_ps(0.0f, 0.0f, 0.0f, 0.0f);
    for (i=0; i<N; i=i+4){
        a_sse = _mm_load_ps((a_addr));

        b_sse = _mm_load_ps((b_addr));
        b_br  = _mm_shuffle_ps(a_sse, a_sse, _MM_SHUFFLE(0,0,0,0));
        b_br  = _mm_mul_ps(b_sse, b_br);
        r_sse = _mm_add_ps(r_sse, b_br);
        b_addr = b_addr + N;
        ..........
   }
 _mm_stream_ps((res_base+row*N+col), r_sse);
}
```

Ezt 4x ismételjük a 4 A mtx. elemre

Eredmények

- **CPU: Core i5 @ 2,5 GHz**
  - x86 kód: 0,21 Gflops/s
  - SSE kód: 0,88 Gflops/s
- **További (nem nagyon fájdalmas) optimalizációs lehetőségek**
  - Több kimenet számítása a ciklusmagban → betöltött A mátrix elemeket többször használjuk fel
  - Segíthet még: egymást követő, adatfüggést tartalmazó utasítások „széttolása”, azaz a memória olvasás és az aritmetikai késleltetések elfedése
    - Ezt x86-on az elérhető viszonylag alacsony regiszter szám korlátozza

Mátrix szorzás (SSE ASM)

- **Az intrinsic utasításokat a fordító optimalizálja**
  - Jelen esetben ez viszonylag sok felesleges utasítást jelenthet (elsősorban mov), de ez fordító függő
- **Lehetőség van assembly betétek beillesztésére**
  - Visual Studio csak x86 esetén támogatja
  - x64 esetén a linkerben fűzhető össze a külön fordított ASM betét
  - Mátrix szorzás példánkban a műveletek nagy része (ciklusmag) SSE utasításokból áll → célszerű az egész ciklust átírni ASM-be
  - x86 esetén 8 db SSE regiszterünk van → elegendő arra, hogy 4*4 mátrix elemet számítsunk egy ciklus iterációban

Mátrix szorzás (SSE ASM)

```
__declspec(align(16)) volatile
   float fzero[4] = {0,0,0,0};
__asm{
    push    eax
    push    ebx
    push    ecx
    push    edx
    movaps  xmm4, fzero
    movaps  xmm5, fzero
    movaps  xmm6, fzero
    movaps  xmm7, fzero
    mov     edx, a_addr
    mov     ecx, b_addr
    mov     eax, c_addr
    mov     ebx, N/4
```

```
loop1:
    movaps  xmm0, [ecx]
    shufps  xmm0, xmm0, 0x0
    movaps  xmm2, [edx+0x0]
    movaps  xmm3, [edx+0x10]
    mulps   xmm2, xmm0
    mulps   xmm3, xmm0
    addps   xmm4, xmm2
    addps   xmm5, xmm3
    movaps  xmm2, [edx+0x20]
    movaps  xmm3, [edx+0x30]
    mulps   xmm2, xmm0
    mulps   xmm3, xmm0
    addps   xmm6, xmm2
    addps   xmm7, xmm3
    add     edx, N*4
.....
    add     ecx, 0x10
    dec     ebx
    jnz     loop1
```

Ezt 4x ismételjük a 4 A mtx. elemre

Eredmények (2)

- **CPU: Core i5 @ 2,7 GHz (2 mag, 4 szál)**

	1 szál	OpenMP (4 szál)
x86 (C kód)	0,21 Gflops/s	0,38 Gflops/s
SSE intrinsic	0,82 Gflops/s	1,53 Gflops/s
SSE assembly	1,94 Gflops/s	3,2 Gflops/s

- **Többszálusítás**
  - OpenMP-vel a legkülső ciklus triviálisan párhuzamosítható
  - Az OpenMP #pragma-án belüli kód nem tartalmazhat ASM betétet → ASM kód külön függvény, ez hívható több szálon

x86 kód + OpenMP

```
int row, col, i;

#pragma omp parallel for private(col, i)
for (row=r_start; row<r_end; row=row+1)
{
    for (col=0; col<N; col=col+1)
    {
        float* a_l = a_base + row*N;
        float* b_l = b_base + col;
        float* c_l = res_base + row*N + col;

        float sum = 0;
        for (i=0; i<N; i++)
        {
        ……
```

Párhuzamos részen kívül deklarált, szál-függő változók private-ek!!

„Bit hekkelés”

- **Párhuzamos bitszintű műveletvégzés**
  - 1 bitek megszámolása (population count)
  - Legnagyobb helyiértékű 1 bit pozíciója
  - Bit sorrend fordítás
- **Sok jó ötlet:**
  http://graphics.stanford.edu/~seander/bithacks.html

POPCNT (1.)


```
b = (b & 0x55555555) + (b >> 1 & 0x55555555);
b = (b & 0x33333333) + (b >> 2 & 0x33333333);
b = (b + (b >> 4)) & 0x0F0F0F0F;
b = b + (b >> 8);
b = (b + (b >> 16)) & 0x0000003F;
```

POPCNT (2.)

```
b = (b & 0x55555555) + (b >> 1 & 0x55555555);
b = (b & 0x33333333) + ((b >> 2) & 0x33333333);
b = (b + (b >> 4)) & 0x0F0F0F0F;
b = b + (b >> 8);
b = (b + (b >> 16)) & 0x0000003F;
```

- {A, B, C, D} * 0x01010101 = {A+B+C+D, B+C+D, C+D, D}

```
i = i - ((i >> 1) & 0x55555555);
i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
i = (i + (i >> 4)) & 0x0f0f0f0f;
i = (i * 0x1010101) >> 24;
```

OpenMP

Szántó Péter

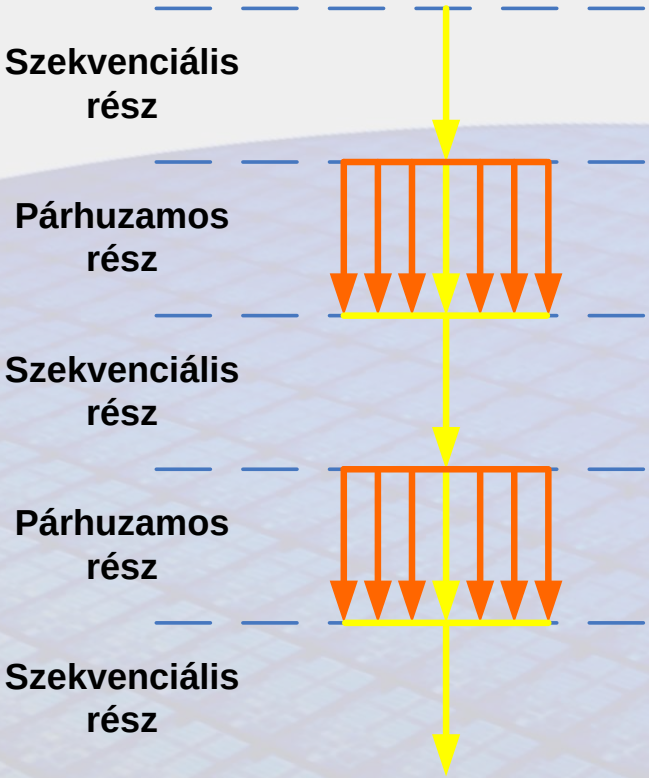
BME MIT, FPGA Laboratórium

Párhuzamos programozás

- **Szálkezelés könyvtárakkal**
  - Win32 API, POSIX Pthread library, Boost, ….
- **Fordító direktívák**
  - Osztott memória modell
  - OpenMP (pragma gyűjtemény)
- **Üzenetküldés alapú megoldások**
  - Az adatmegosztás explicit (a programozó feladata)
  - MPI: Message Passing Interface
  - PVM: Parallel Virtual Machine

OpenMP modell

- SPMD: Single Program Multiple Data
- Osztott memória: minden szál hozzáfér
- „Fork – join” modell



Nyelvi elemek

- **Párhuzamos programrészek**
  - #pragma omp parallel
- **Worksharing**
  - #pragma omp for, #pragma omp sections
- **Adatkörnyezet**
  - #pragma omp parallel shared/private (…)
- **Szinkronizáció**
  - #pragma omp barrier
- **Futási idejű, ill. környezeti változók**
  - int my_thread_id = omp_get_num_threads();
  - omp_set_num_threads(…);

Párhuzamos blokk

- **A létrehozott szálak számát az OMP_NUM_THREADS környezeti változó (vagy explicit megadás) adja meg**
- **Minden szál a párhuzamos blokkon belüli kódot hajtja végre**
- **NINCS szinkronizálás a szálak között**
  - Nem detirminisztikus hogy a szálak ugyanazt a kódrészletet mikor érik el
- **Miután minden szál végrehajtotta a párhuzamos blokkot a master szál kivételével megszűnnek**

Párhuzamos blokk

- A #pragma hatóköre egy strukturális blokk: {……}
- A blokkon belüli kifejezések több szálon hajtódnak végre

```
……
#pragma omp parallel
{
   print(„Hello world!”)
}
```

- Eredmény: a szálak számának megegyező számú „Hello world!” felirat (nem feltétlenül egymás után)

Párhuzamos blokk

- **Egyedi szál azonosító**
  - Különböző kód is végrehajtható

```
……
#pragma omp parallel
{
   myid = omp_get_thread_num();
   if (myid==0)
       master_function();
   else
       thread_function();
}
……
```

Párhuzamos blokk - végrehajtás

- **Dinamikus mód (alapértelmezett)**
  - A létrehozott szálak száma dinamikusan változik
  - A beállított szám felső limit
- **Statikus mód**
  - A szálak száma a programozó által beállított
  - #pragma omp parallel for num_threads(4)
  - omp_set_num_threads(4);
- **A fordító létrehozhat szekvenciális kódot**
- **Végrehajtási mód beállítása**
  - OMP_DYNAMIC környezeti változó
  - omp_set_dynamic() függvény

Párhuzamos blokk paraméterek (1)

- **shared(var1,var2,…)**
  - Szálak között megosztott változó
- **private(var1,var2,…)**
  - Minden szálnak saját példánya van a változóból
- **firstprivate(var1,var2,…)**
  - Inicializálás a párhuzamos blokkba lépés előtt, private változókba átmásolódik az inicializálási érték
- **lastprivate(var1,var2,…)**
  - Megtartja értékét a párhuzamos blokk után. Azaz a legutolsó iteráció értéke átmásolódik a szekvenciális (master) blokk változójába
- **default(shared|private|none)**
  - Változók alaptípusa
  - Egyébként a deklaráció helyétől függ!

Párhuzamos blokk paraméterek (1)

- **nowait**
  - A szálak nem várják be egymást egy párhuzamos rész végén
- **if(expression)**
  - Feltételes párhuzamosítás
- **schedule(type [,chunk])**
  - Ciklus iterációk ütemezése az egyes szálak között
- **reduction(operator|intrinsic:var1,var2…)**
  - Redukciós operátor/változók megadása

Ciklus párhuzamosítás

- **FOR ciklus többszálú végrehajtása**

```
int i;
float a[N],b[N],r[N];
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<N; i++){
       res[i]=a[i]*b[i];
   }
}
```

```
int i;
float a[N],b[N],r[N];
#pragma omp parallel for
for (i=0; i<N; i++){
   res[i]=a[i]*b[i];
}
```

- **Szinkronizáció**
  - Ciklus mag vége (kivéve: #pragma omp for nowait)
  - Teljes ciklus végrehajtása

Egymásba ágyazott ciklus

```
int i, j;
#pragma omp parallel for
for (j=0; j<N; j++){
  for (i=0; i<N; i++){
    res[i]=a[i]*b[i];
  }
}
```

- Alapesetben csak a j (külső) ciklusváltozó private
- i-ből is minden szálnak saját példányra van szüksége

```
#pragma omp parallel for private(i)
```

Schedule

```
for (i=0; i<1100; i++) { res[i]=a[i]*b[i] }
```

- #pragma omp parallel for schedule (static, 250)

250	250	250	250	100
t0	t1	t2	t3	t0

- #pragma omp parallel for schedule (dynamic, 200)

200	200	200	200	200	100
t3	t1	t2	t0	t1	t3

- #pragma omp parallel for schedule (guided, 100)

137	120	105	100	100	100	100	100	100	100	38
t0	t3	t0	t1	t2	t3	t0	t1	t2	t3	t0

- #pragma omp parallel for schedule (auto)

Sections

- 1 section egyszer hajtódik végre
- Ha sok section van, egy szál többet is végrehajthat
- Ha kevés a section-ök száma, néhány szál nem csinál semmit
- A section – szál összerendelés előre nem ismert

```
#pragma omp parallel
   default(none)
   shared(val,thread){
#pragma omp sections
{
    #pragma omp section
    {
        foo1();
    }
    #pragma omp section
    {
        foo2();
    }
}
```

1. szál hajtja végre

2. szál hajtja végre

Single

- **A single blokk egyszer, egyetlen szál által hajtódik végre**
  - Pl. egy szál inicializál:

```
#pragma omp single
{
  a = 10;
}
#pragma omp for
{
  for (i=0; i<N; i++)
    b[i] = a;
}
```

Pl.: összegzés

- N elemű tömb elemeinek összege

```
sum = 0;
#pragma omp parallel for
for (i=0; i<N; i++)
 sum += a[i];
```

HIBÁS!! Miért?

Critical

- **Mindegyik szál végrehajtja, de egy időben csak egy (!= single !!!)**

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++){
      lsum = lsum + A[i];
   }
   #pragma omp critical{
      sum += lsum;
   }
}
```

Szálankénti rész-összegek

Rész-összegek szekvenciális összegzése

Redukciós műveletek

- Redukciós operátorok biztonságos végrehajtása
- Előző összegzéses példa szebben:

```
sum = 0;
#pragma omp parallel for      \
  default(shared) private(i)  \
  schedule(static,chunk)      \
  reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + A[i];
```

- Redukciós operátorok: +,*,-,&, |, ^, &&, ||

Barrier

```
int x = 2;
#pragma omp parallel shared(x)
{
  int tid = omp_get_thread_num();
  if (tid == 0)
    x = 5;
  else
   printf("[1] thread %2d: x = %d\n",tid,x);

  #pragma omp barrier
   printf("[2] thread %2d: x = %d\n",tid,x);
 }
```

Néhány szál esetén x itt még lehet 2

Szinkronizáció után minden szálnál x=5

Szinkronizáció

- **#pragma omp master {…….}**
  - Csak a master szál hajtja végre
- **#pragma omp critical {…….}**
  - Egy időben csak egy szál hajtja végre
- **#pragma omp barrier**
  - Szálak bevárják egymást
- **Egyéb**
  - ATOMIC
  - ORDERED – a ciklus iterációk végrehajtási sorrendje ugyanaz, mint szekvenciális esetben
  - FLUSH – cache kezelés