# Újrakonfigurálható technológiák nagy teljesítményű alkalmazásai

## Gyakorlat: SSE utasításkészlet

**Szántó Péter**

**BME MIT, FPGA Laboratórium**

# Vektorizáció

- **Inline assembly**
  - 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

# SSE utasítások

- **70 új utasítás**
- **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, 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 Bridge (2. gen. Core iX)
- **Compiler:**
  - GCC v4.5
  - Intel C Compiler 11.1
  - VS 2010
- **Operációs rendszer**
  - Linux kernel 2.6.30, Windows 7

# 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)

- **Intrinsic**
  - 4xFP: __m128
  - 4xINT: __m128i
  - 2xDP FP: __m128d
- **Assembly**
  - x32: xmm0…xmm7 regiszterek
  - x64: xmm0…xmm15 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
  - MOVAPS – __mm_store_ps
  - MOVSSS – __mm_store_ss (csak 0. szó)

# 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)))

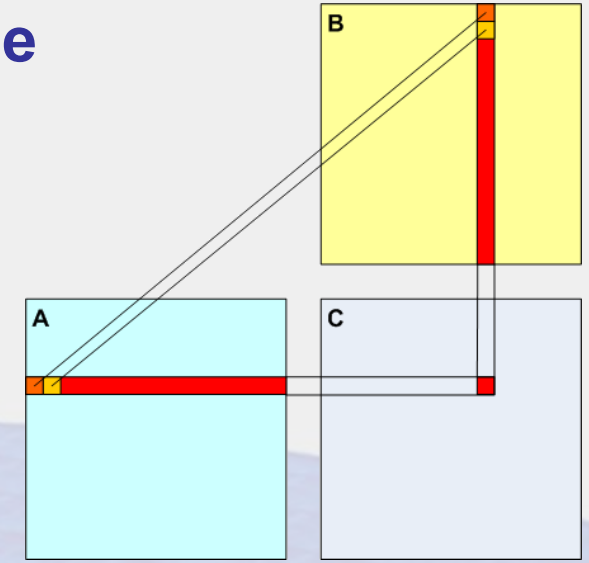
# Mátrix szorzás (x86)

```
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 jelent (elsősorban mov)
- **Lehetőség van assembly betétek beillesztésére**
  - Visual Studio csak x86 esetén támogatja
  - x64 esetén a linkerben fűzhető össze a külön fordított ASM betét
  - Mátrix szorzás példánkabn 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 @ 2,5 GHz**
  - x86 kód: 0,21 Gflops/s
  - SSE intrinsic: 0,88 Gflops/s
  - SSE ASM: 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;
```