



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Újrakonfigurálható technológiák nagyteljesítményű
alkalmazásai tantárgy
[VIMIM364]

Féléves feladat

DNS szekvencia illesztés x86, x64 CPU környezetben

Készítette:

Szikra István

URLJRN

Lovas Szilárd

SZE BCBED8

Konzulens:

Szántó Péter

BME-MIT



1	Bevezetés	3
2	A feladat specifikálása	5
2.1	A probléma leírása	5
2.2	Megvalósítandó funkcionalitás	7
2.3	Input adatok	8
2.4	Output adatok.....	9
3	Program dokumentáció	9
3.1	Általános leírás.....	9
3.2	Program dokumentáció	10
3.2.1	A főprogram működése.....	10
3.2.2	A minták összehasonlítása	11
3.2.3	A bázisok összehasonlítása (basediff)	11
3.2.4	A popcnt utasítás.....	12
3.3	Fejlesztési tapasztalatok.....	13
4	Felhasználói útmutató	13
5	Teszt eredmények	15
5.1	Tesztkörnyezet AMD.....	15
5.2	Tesztkörnyezet INTEL.....	15
5.3	Futtatási eredmények	16
6	Irodalomjegyzék	17
6.1	Bioinformatika:	17
6.2	Optimalizálás, CPU referencia, algoritmus:	17
6.3	Ábrák és egyéb felhasznált anyagok:.....	18
7	Lemelmelléklet.....	19



1 Bevezetés

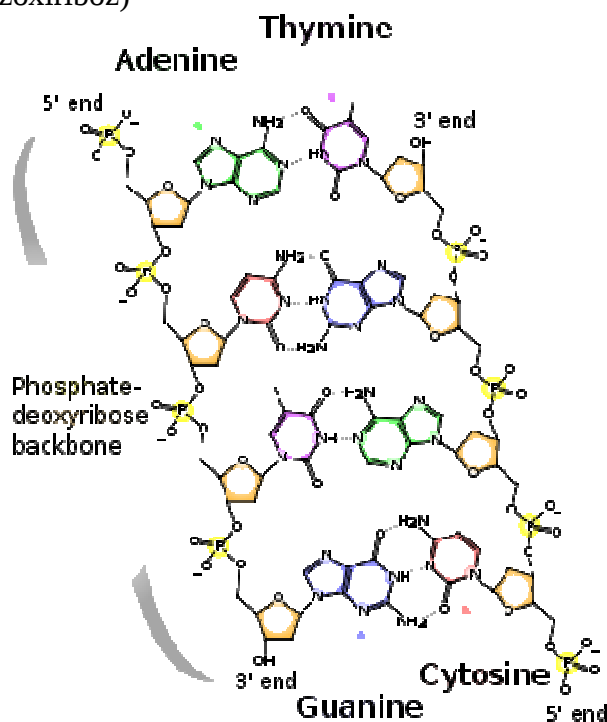
A bioinformatika - napjaink egyik intenzíven fejlődő interdiszciplináris tudományága. Egyik fő szakterülete a molekuláris bioinformatika, amely az élő szervezetekben fellelhető információhordozó molekulákat (a DNS-t) alkotó bázispárok sorrendjének meghatározásával, feldolgozásával, értelmezésével foglalkozik.

A DNS vagy teljes nevén a dezoxiribonukleinsav egy ismétlődő nukleotid egységekből álló nagyméretű molekula (polimer), amely az élő szervezetek felépítésében és reprodukciójában tölt be fontos szerepet.

- Az élő szervezeten belül lehetővé teszi a genetikai információ biztonságos tárolását, így például az elhalt sejtek helyére, az eredetivel (közel) megegyező sejtek keletkezhetnek.
- Szaporodásnál biztosítja az egymást követő generációk között a tulajdonságok továbbadását
- Ráadásul lehetővé teszi, hogy ezek a tulajdonságok a keresztezés és a mutáció során meg is változzanak. Ezen lehetőség hiányában nem lenne evolúció, amely az élőlények környezethez való alkalmazkodásának igen fontos képessége.

Ahogy az előbb említettük, a DNS molekula ismétlődő nukleotidok sorozatából áll. A nukleotidok három, egymáshoz kapcsolódó elemből épülnek fel, ezek:

- Egy nitrogén tartalmú szerves bázisból **Adenin, Timin, Citozin** vagy **Guanin**
- Egy pentóz cukorból (dezoxiribóz)
- És egy foszfátcsoporthoz



1. ábra a DNS molekula egy szakasza



Mivel a nukleotidok egymástól, csak a szerves bázisban különböznek, ezek alapján négy fajta építőkövet A, C, T, G különböztethetünk meg.

A DNS szekvencia tehát egy A, C, T, G betűkből álló sorozattal kódolható, amelynek hossza függ az adott élőlénytől. Egy vírus esetében pár ezer bázispár, míg az ember esetében mintegy 3.15 milliárd bázispár hosszúságú.

Példaként itt van két emlős DNS-ének egy-egy szakasza, amiből ha más nem is, az azért remekül látszik, hogy a háziegér és az ember között vannak apróbb genetikai eltérések.

- Mus musculus TGCGTTCTCA CCAGGCTCCT CGGCAATGCC A
- Homo sapiens TGCGATCTCA CCAAGATCTT CCGAAATGCC A

Habár a fenti szekvenciák meghatározására a 1970-es évektől több módszert is kidolgoztak, illetve a meglévőket tökéletesítették, valamennyi eljárás közös vonása, hogy a bázisok sorrendjét csak kis (maximum ~1000 bázis) méretű töredékekben (read-ekben) képes meghatározni. Fontos kiemelni azt is, hogy a sorrend meghatározásánál hibák is fellelhetnek.

A következő szakaszban pár, napjainkban használatos DNS szekvenáló berendezés adatait gyűjtöttük össze.

ABI 3730 XL

- Maximum 1,100 bázis / read
- 96 read / futtatás
- 1Mbázis / nap
- MetóduS: Sanger
- Költség: 1\$/Kbázis



Roche GS FLX / 454

- Maximum 400 bázis / read
- ~ 1 millió read / futtatás
- Maximum 400Mbázis / futtatás
- MetóduS: Pyrosequencing
- Költség: 0.1\$/Kbázis



Illumina Genom Analyzer Iix / Solexa

- Maximum 76 bázis / read
- ~ 160 millió read / futtatás
- Maximum 10Gbázis / futtatás
- MetóduS: Sequencing by synthesis
- Költség: 0.01\$/Kbázis





Ahogy látható, a különböző metódusok, különböző méretű read-eket eredményeznek. Minél hosszabbak a töredékek, annál könnyebb belőlük az átlapolódó szakaszok figyelembe vételével a teljes szekvencia rekonstruálása (de novo szekvenálás). Ugyanakkor ezek a módszerek rendszerint drágábbak, és lassabbak, mint a kisméretű töredékeket eredményező társaik.

A 100 bázisnál rövidebb töredékeket „short-read”-eknek nevezzük. Az ilyen, rövid töredékeket eredményező módszerek alkalmatlanok a de novo szekvenálásra, ugyanakkor kiemelkedő a szerepük más területen, például a gyógyszerkutatásban, és a genetikai eredetű betegségek felismerésében.

2 A feladat specifikálása

A feladat kiírásban short-read illesztés szerepelt.

2.1 A probléma leírása

Mivel a feladat kidolgozás kezdetén elég kevés információval rendelkezünk, irodalomkutatást végeztünk a témakörben. Viszonylag hamar ráakadtunk az interneten az erre a feladatra készült, többnyire szabad forráskódú programokra, mint például a Bowtie vagy a Burrows-Wheeler Aligner. Ezek dokumentációja, illetve a bennük használt módszerek nyújtottak segítséget, a probléma jobb megismerésében, amely a következő:

- Adott egy s hosszúságú R referencia szekvencia, ami származhat például a Human Genom Project-ből.
- Adott $m_1..m_k \in M$, azonos hosszúságú minták halmaza, jelölje ℓ a minták hosszát
- Határozzuk meg minden $m_j \in M$ – re az összes kezdő pozíciót R -ben, ahol m_j előfordul.

Az előbb leírt problémát módosítja, hogy a szekvenáló berendezések, a minták meghatározására csak bizonyos pontossággal képesek, vagyis nem csak a teljes egyezéseket kell elfogadnunk, hanem azokat is, ahol a bázisokban mért távolság kisebb egy adott értéknél. Hogy ez az érték mekkora, függ a minták minőségétől és hosszúságától.

Ha egy bázis hibás olvasásának a valószínűsége: ε , akkor annak a valószínűsége, hogy egy ℓ hosszú mintát pontosan h hibával sikerül elolvasnunk:

$$P = \binom{\ell}{h} \varepsilon^h (1 - \varepsilon)^{\ell-h}$$

(Feltételezve, hogy az olvasási hibák független események, illetve ε ugyanakkora m_j valamennyi bázis helyi értékére nézve)



Esetünkben $\ell = 25$ és egy szokásos 99%-os pontossággal ($\varepsilon = 0.01$) számolva meghatároztuk P és $\sum P$ értékét különböző h értékekre:

h	P	$\sum P$
0	0,777821359	0,777821359
1	0,196419535	0,974240895
2	0,023808429	0,998049323
3	0,00184375	0,999893073
4	0,000102431	0,999995504

Látható, hogy ilyen olvasási pontosság és méret mellett nem igazán érdemes a hibák számát 2-3 fölé engedni. A hibásan olvasott, de helyesen beazonosított minták száma már alig változik, ugyanakkor növekszik annak esélye, hogy egy hibátlan mintát nem a megfelelő helyre sikerül illeszteni (szélsőséges esetben, $h = \ell$ minden minta illeszkedik mindenhova)

Az olvasási hibákon túl, a feladat nagy mérete is nehezíti a megoldást.

Ha meg szeretnénk találni az összes olyan kezdőpontot R -ben, ahonnan egy m minta kisebb, mint h hibával illeszkedik, akkor első megközelítésben tehetjük azt, hogy végigtoljuk m -et R mentén, és minden egyes pozícióban megvizsgáljuk az eltérések számát. Ha a $\Delta \leq h$, akkor feljegyezzük pozíciót.

Ennek a megoldásnak a költsége egy mintára nézve $\Theta(sl)$, k darab mintára nézve pedig $\Theta(slk)$. Ha mondjuk, a Illumina Genom Analyzer IIX szekvenáló berendezés egy futtatásának az eredményét szeretnénk illeszteni az emberi genom-hoz, akkor $s = 3.15 \cdot 10^9$ $l = 25$ $k = 1.6 \cdot 10^8$ ezeket összeszorozva egy elég nagy szám jön ki, érdemes tehát más megoldás után nézni.

A keresés felgyorsításához általában valamilyen indexelési technikát alkalmaznak. R nagy mérete miatt készítsünk indexet M -re. Ennek költsége ideális esetben: $\Theta(kl)$. Ezek után R mentén haladva, minden lehetséges pozíciójában lévő ℓ hosszúságú szakaszra elvégezzük a keresést M -ben az index segítségével. A keresés eredményeként megkapjuk azokat az m mintákat amelyek legfeljebb h hibával illeszkednek az adott pozícióra. Ideális esetben a keresés költsége $O(1)$, tehát a teljes költség az index felépítéséből és az R -en való végig lépkedésből tevődik össze: $\Theta(s + kl)$ ami már jóval kedvezőbb a kiinduló $\Theta(slk)$ költségénél (természetesen $O(k)$ tárhely felhasználás árán).

Összefoglalva a következő elvárások vannak az index-szel kapcsolatban:

- gyorsan lehessen felépíteni és keresni benne
- gazdálkodjon jól a tárhellyel (M is nagy)
- a lekérdezés során kezelje a h hibát



Az első két elvárásnak remekül megfelel egy H hasító táblázat. Megfelelő f hasító függvény választásával az alapvető szótár műveletek várható ideje $O(1)$, és a szükséges tárhely $O(k)$ közelében tartható. Az ütközések feloldására pedig használhatunk láncolt listát.

A legnagyobb hibája a fenti megközelítésnek, hogy nem kezeli a h hibával kapcsolatos elvárásainkat. Vagyis, nem biztosítja, hogy ha m és m' között maximum h hiba eltérés van, akkor $H[f(m)] = H[f(m')]$

A megoldás a fenti problémára a szűrés vagy filtráció.

Filtráció esetén m minta nem minden bázisát vesszük figyelembe a hasító táblázat felépítésekor, hanem annak csak egy kivonatát vagy tulajdonságát (feature). Más szóval az index felépítésekor az m mintákat valamilyen feature alapján csoportokba soroljuk

Egy (ℓ, h) feature-nek azt a tulajdonságot nevezzük, amellyel minden ℓ hosszúságú szakasz rendelkezik, ha legfeljebb h helyen térnek el egymástól. Ha két ℓ hosszúságú minta legfeljebb h helyen tér el egymástól, akkor az (ℓ, h) feautre-ük azonos, de ez a hozzárendelés fordított irányban nem igaz. Vagyis ha az (ℓ, h) feautre azonos lehet akkor is, ha egyébként nem teljesítik a h hibára vonatkozó követelményt. Egy (ℓ, h) feature fals pozitív rátája $p(r)$, várható értéke pedig E_p . A fals pozitivitás hatását figyelembe véve a keresés költsége: $\Theta(s + l * találatok_száma + slkE_p)$. Mint látható a képletben sajnos ismét megjelent slk amit a vizsgálódásaink elején szerettünk volna elkerülni. Fontos tehát E_p értékét alacsonyan tartani, amelyre több különböző filtrációs eljárást is kidolgoztak: word matches, discontiguous seeds, multiple seeds eljárások, melyeket részletesen bemutat Jeremy Buhler - CSE587 Short Read Class 1-3 jegyzete.

A konzultáció során kiderült, hogy az átadott adatok már átestek fenti előfeldolgozási eljáráson, így nekünk elegendő a minták összehasonlítását kidolgozni.

2.2 Megvalósítandó funkcionalitás

A fentiek alapján tehát el kell készíteni egy mintaillesztő programot, amely inputként a 2.3 szakaszban leírt bemeneti állományokat fogadja. Értelmezve a parancsállomány rekordjait, összehasonlítja az adatállományból startB-től kezdődően, lengthB darab mintát (rekesz B), startA-tól kezdődően, lengthA darab mintával (rekesz A).

Az összehasonlítás során figyelembe kell venni a parancs rekordban szereplő mismatch (maximális eltérés) értéket, kimenetként pedig olyan A-B összetartozó minta párok indexét kell szolgáltatni, melyeknél az eltérés maximum mismatch nagyságú.



A fenti feladat elvégzésénél lehetőség szerint minél jobban ki kell használni az x86 és x64, architektúrájú Intel és AMD CPU-k lehetőségeit (MMX, SSE, 3DNow! Utasításkészlet).

2.3 Input adatok

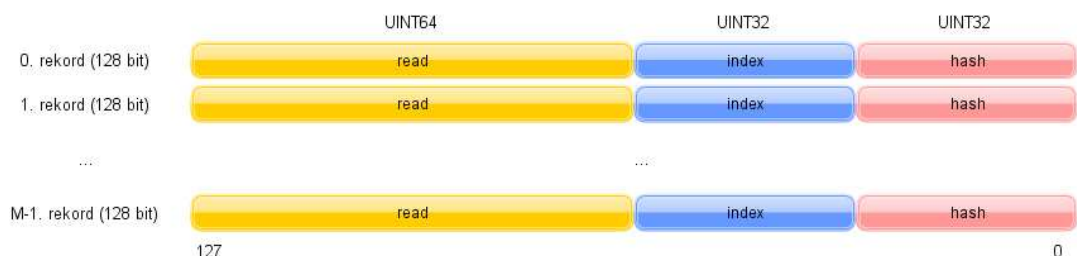
A konzultációk alkalmával rendelkezésünkre bocsátották a feladat elvégzéséhez szükséges állományokat:

- **f_cmd.bin** – 256.256 bájt ($N = 16.016$ rekord) hosszúságú parancsállomány, felépítése a következő:



2. ábra a parncsállomány felépítése

- **f_data.bin** – 104.855.632 bájt ($M = 6.553.477$ rekord) hosszúságú adatállomány, felépítése a következő:



3. ábra az adatállomány felépítése

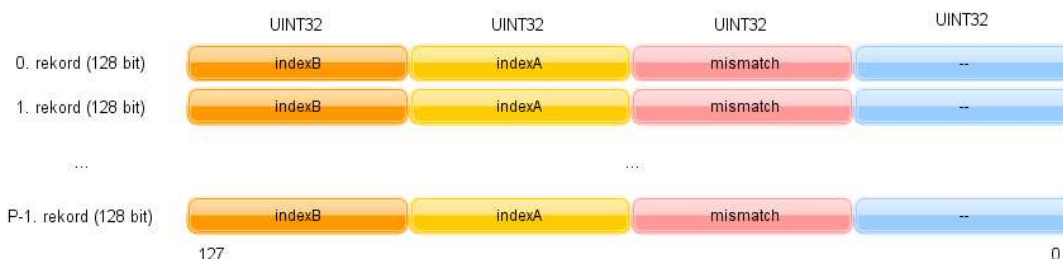
- **cuda_host.cu** – az előző állományok értelmezéséhez szükséges struktúrákat tartalmazó CUDA forrás állomány.

Kiegészítő információként még megtudtuk, hogy az adatállományban 25 bázis hosszúságú minták vannak, bázisonként két biten tárolva.



2.4 Output adatok

A fenti szöveges specifikáció alapján a program kimeneti fájl formátuma:



4. ábra a kimeneti állomány felépítése

3 Program dokumentáció

3.1 Általános leírás

A programkészítés fázisát jelentős irodalomkutatás előzte meg. Igen hasznosak voltak a feladat megoldásánál Agner Fog kód optimalizációval foglalkozó írásai, példa programjai, valamint az AMD és Intel processzorok különböző dokumentációi, melyeket az irodalomjegyzékben soroltunk fel.

Bár a fejlesztés AMD Phenom II processzoron történt, az itt alkalmazott optimalizációs eljárások jelentős része az Intel Core2Duo processzorokon is működik. Elsősorban 64 bites futtatásra készült, de 32 bites verzió is fordítható belőle. Ebben az esetben azonban jelentős teljesítmény csökkenést okozhat a 64 bites regiszterek hiánya és a csökkentett regiszter készlet (r8..r15, xmm8..xmm15 regiszterek csak 64 bites módban érhetőek el). Az előző megfontolásokat figyelembe véve, teljesítményorientált alkalmazások esetében mindenképpen szükségesnek látjuk 64 bites operációs rendszer és 64 bites kód használatát. Nem volt célunk 386 kompatibilis kód készítése, ezekre a régebbi típusú processzorokra máskepp kell optimalizálni az eltérő architektúra, utasítás készlet és végrehajtási idők miatt.

A programfejlesztés Visual Studio 2008 programcsomag felhasználásával történt, C nyelven. A sebesség kritikus szakaszok egy része assembly nyelven íródott, illetve a C kód hangolása a fordító által generált kód elemzésével történt.

Alkalmazott teljesítménynövelő (HPC) technikák

- (Algoritmusoptimalizálás – mindig az elsődleges, legtöbb javulással kecsegtető)
- OpenMP használata, párhuzamosításra, több processzormag kihasználására
- Loop Unroll (loop overhead csökkentés)
- Adatfüggőség (dependency), ütemezés (scheduling) ciklusmagon belül több összehasonlítás
- Intrinsic utasítások, ezek körültekintő megválasztása



- (Prefetch – cache opt.)
- Performance counter-ek használata a tesztelésre, data cache miss, branch missprediction (ami több információt nyújt, mint csak a futásidő)
- Adat, regiszter méret optimális megválasztása
- Megfelelő utasítások kiválasztása a latency figyelembe vételével
- Indirekt módon, a forrásprogram megváltoztatásával rávenni a fordítót kedvezőbb kód generálására
- utasítás sorrend megváltoztatása. (C nyelvben erre kevés lehetőség, assembly esetén viszont egy jól használható módszer).

Az optimalizálásnál figyelembe kell venni a szuperskalár architektúrát:

- azt, hogy 32 byte kódot hoz fel a processzor az L1 cache-ből dekódolásra
- képes egyszerre akár 3 utasítást is ütemezni
- képes nem sorrendi végrehajtásra
- képes regiszter átnevezésre

A programozási munka során, az egyik legtöbbet használt dokumentáció a 40546-es számú "Software Optimization Guide for AMD Family 10h Processors" volt. Ebben találtuk meg a szükséges végrehajtási késleltetési időket.

Ezen felül számos tanácsot tartalmaz nagy teljesítményű kódok írásához például:

- Load-Execute utasítások használata (ADD rax, QWORD PTR [foo]) az elkülönített betöltő, és végrehajtó utasítások helyett
- Adat mozgatás általános célú (GP), MMX és XMM regiszterek között
- A Population Count (popcount) utasítás alkalmazása

3.2 Program dokumentáció

A program forráskód szintű dokumentációja Doxygen dokumentációs rendszerrel készült, elérhető a [lemez mellékletéről](#).

A könnyebb átláthatóság miatt, itt csak a működés főbb pontjait szeretnénk bemutatni.

3.2.1 A főprogram működése

A program belépési pontja az SR.cpp állományban található. A `_tmain()` függvény értelmezi az átadott parancssori kapcsolókat és paramétereket (parancs, adat, kimeneti állományok elérési útvonala, verzió lekérés, verbose, `-i` kapcsoló). Ha nem kapott a `-i` kapcsoló paramétert (lásd: 4. szakasz) akkor a programnak meg kell állapítania, hogy a minták összehasonlítására melyik metódus a legalkalmasabb, figyelembe véve a rendszerben található CPU képességeit. Meghívja tehát `cpu_id()` függvényt, hogy megállapítsa a rendelkezésre álló processzor tulajdonságait. Valójában csak azt dönti el, hogy a processzor rendelkezik-e utasítás szintű popcnt támogatással, vagy sem. (SSE meglétét feltételezzük) Az alapértelmezett összehasonlító algoritmust mérési eredmények alapján határoztuk meg. `sr_bincmp_gpr_popcnt` – popcnt támogatás esetén, és `sr_bincmp_sse_lp_popcnt` popcnt támogatás hiányában.



A `_tmain()` függvény ezután meghívja a bemeneti állományok beolvasására szolgáló `ReadData()` és `ReadCmd()` függvényeket, majd törli a kimeneti puffert – `SRInit()` (erre az esetleges többszöri futtatás miatt van szükség) és elvégzi a minták összehasonlítását `SRTest()` függvények. Ez utóbbiakat Agner Fog `mon_main()` (performance counters) függvényén keresztül teszi, ha a `SR.h`-ban `#define PERFMON 1`. Az összehasonlítás elvégzése után megnyitja a kimeneti állományt, elmenti az eredményeket, és felszabadítja a futás közben lefoglalt memória területeket.

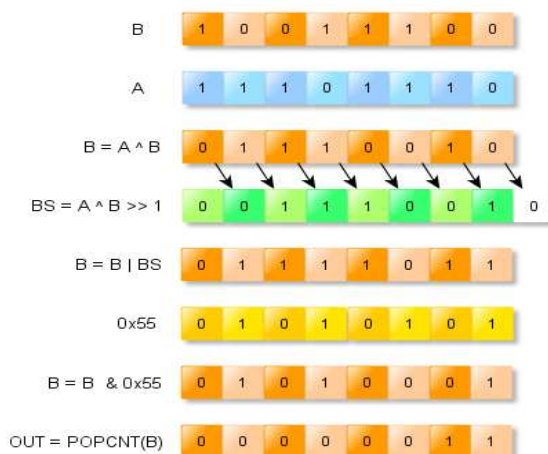
3.2.2 A minták összehasonlítása

Ahogy az előző szakaszban már említettük, a minták összehasonlítását az `SRTest()` függvény végzi a `*psr_bincmp` függvény pointer által megcímzett metódussal kosaranként. A `*psr_bincmp` mutató a főprogramban kap értéket `-i` kapcsoló értékétől függően. Így dől el, hogy melyik függvény kerül meghívásra az összehasonlítás elvégzéséhez. A `*psr_bincmp` paraméterben megkapja a parancs file egy sorát, azt hogy mely 2 kosárba lévő mintákat kell összehasonlítani. Ez a függvény foglalja le az eredményes összehasonlításoknak, találatoknak a memóriát.

3.2.3 A bázisok összehasonlítása (basediff)

A program működésének leginkább sebesség kritikus része a minták összehasonlítása. Erre 5. ábrán látható algoritmust alkalmaztuk.

A programban használt megvalósítás (lásd a következő szakaszban) annyiban tér el ettől, hogy a regisztereink 64, ill.128 bitesek. SSE verzióban egyszerre hasonlítjuk össze $(A0, A1)$ mintapárt, $(B0, B1)$ és $(B1, B0)$ minta párral, megkapva ezek bázis távolságát. A másik eltérés, hogy az algoritmus végén található `popcnt` utasítás nem része a gyakorlatban megvalósított `basediff` függvénynek, hanem attól külön került elhelyezésre.



5. ábra: a basediff algoritmus

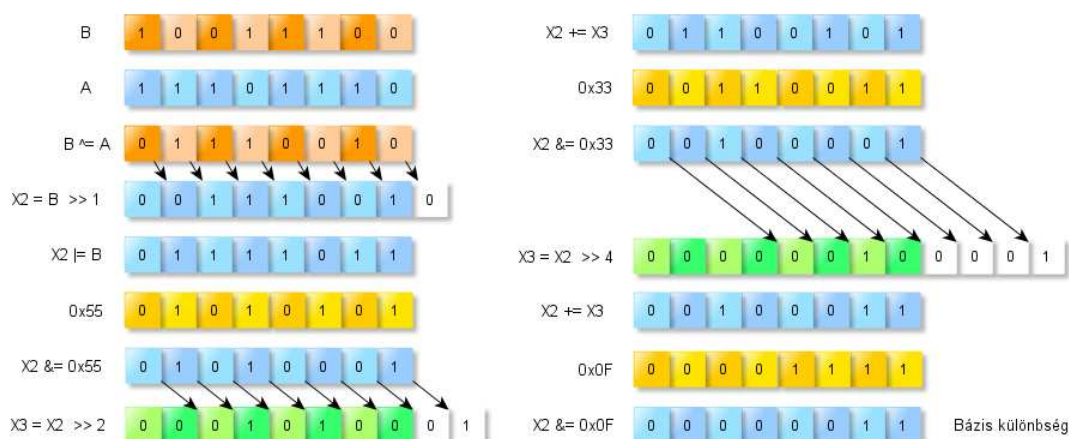
```
inline void basediff_sse2x11(__m128i a, __m128i& b, __m128i& bx)
{
    __m128i bs, bxs;
    b = _mm_xor_si128(b, a); // (A1, A0) ^ (B1, B0) > (A1^B1, A0^B0) = (resAB, res)
    bx = _mm_xor_si128(bx, a); // (A1, A0) ^ (B0, B1) > (A1^B0, A0^B1) = (resA, resB)
    bs = _mm_srli_epi64(b, 1); // bs = (A1^B1>>1, A0^B0>>1)
    bxs = _mm_srli_epi64(bx, 1); // bxs = (A1^B0>>1, A0^B1>>1)
    b = _mm_or_si128(b, bs); // b |= bs
    bx = _mm_or_si128(bx, bxs); // bx |= bxs
    b = _mm_and_si128(b, xmm55); // b &= 0x5555...
    bx = _mm_and_si128(bx, xmm55); // bx &= 0x5555...
}
```



3.2.4 A popcnt utasítás

A popcnt művelet csak a legújabb AMD (SSE4a), és intel Core i7 (SSE4.2) processzorokban érhető el, és 16, 32 és 64 bites általános célú regiszterekkel működik. Régebbi processzorokban a popcnt utasítást szoftveresen kell megvalósítani. Az előadások során megismerkedtünk egy hatékony population count algoritmussal, de az interneten több egyéb megvalósítás is található. Ezek egy részét popcount.h állományban gyűjtöttük össze.

A tényleges megvalósításban a legtöbb segítséget ismét egy AMD dokumentáció nyújtotta, egy régebbi 22007-es számú "AMD Athlon Processor x86 Code Optimization Guide". Ebben nem csak az elmélet részletes magyarázata található, hanem a tényleges assembly megvalósítás, általános célú és MMX regiszterekkel is. Az utóbbi könnyen módosítható XMM regiszterekre, mivel minden utasítás működik 128 bites regiszterekkel is. (Valójában 2x64 bites regiszterekkel, ha precízek akarunk lenni. Egy shift például nem a teljes 128 bitet tolja, hanem a két 64 bites részt, vagyis két helyen van kicsordulás). A 6. ábrán a bázis különbségek kiszámításához módosított popcnt algoritmus látható.



6. ábra basecmp és popcnt algoritmus

A fenti példa, 8 bites mintákon mutatja be az algoritmus működését. A gyakorlatban, nagyobb szószélességnél a PSADBW (Packed Sum of Absolute Differences) művelet használható a 8 bites részeredmények összegzésére megkapva 2 darab 64 bites pop count eredményét.



3.3 Fejlesztési tapasztalatok

A program készítése során napfényre került a fordítóprogram néhány hibája is, példaként szolgáljon az alábbi kódrészlet, és a hozzá tartozó megjegyzés.

```
uint64_t r;  
__m128i xmmB;  
  
r = _mm_cvtsi128_si64x(xmmB);           // MOVD  
xmmB = _mm_shuffle_epi32(xmmB, 0x4e);   // swap low <-> high  
rAB = _mm_cvtsi128_si64x(xmmB);         // MOVD  
r = __popcnt64;                         // Popcount  
  
// A fenti kódrészlet __popcnt64 utasításából a fordító  
// popcnt rcx, xmm1  
// utasítást generál, ami egy érvénytelen utasítás...
```

Egy másik hiba, bár az előzőnél kevésbé súlyos: a bevezetőben említett 40546-es dokumentum azt javasolja, hogy C, C++ nyelvű programokban, tömbkezelésnél kerüljük el a pointeres címezést, és helyette használjuk inkább a szokásos tömb jelöléseket. Viszont úgy tűnik, hogy ebben az esetben a Visual Studio 2008 több felesleges utasítást is generál. (szerencsére nem a teljesítmény kritikus részekben)

Az ilyen helyeken egy pointer hozzárendeléssel sikerült javítani a generált kódon (ami viszont rontotta az olvashatóságot, így végül nem került be a végleges programba)

4 Felhasználói útmutató

Az elkészült program parancssori felülettel rendelkezik, az SR.exe parancssorból indítható. Használatáról a szokásos -h, /h, -?, /? kapcsolók bármelyikével kaphatunk egy rövid áttekintést.

```
sr -h  
Short-Read V1.3 build May 9 2010 19:37:59 UNICODE  
Irta es rendezte: Szikra Istvan, Copyright 2010  
start with /h for help  
param h: (null)  
Usage:  
    SR.exe [-options] [cmd name [data name [out name]]]  
    SR.exe [-c<cmd filename>] [-d<data filename>] [-o<out name>] [-  
options]  
Options:  
    -h, /h, -?, /?: this help  
    -V: version  
    -v<i|c|s>: verbose output: cpuid, compare, save  
  
    -i<n>: disable autoselect compare version by cpuid, and specify  
version manually
```

Paraméterek nélkül indítva az aktuális könyvtárban keresi a 2.3-es szakaszban részletesen bemutatott input állományokat f_cmd.bin és f_data.bin néven. Ha megtalálta őket, akkor detektálja a rendszerben található processzor képességeit, és az



adott CPU-nak leginkább megfelelő metódussal elvégzi az összehasonlítást. Az összehasonlítás eredményét az aktuális könyvtárba menti el `f_out.bin` néven (lásd: 2.4-es szakasz). A program futása közben tájékoztat az rendszerben található processzor típusáról, képességeiről, a választott összehasonlítási metódusról. Rövid statisztikát készít a bemeneti és kimeneti adatokról, és a futási időről.

Ha el akarunk térni az alapértelmezett működéstől, akkor a `help`-ben (előző szakasz) leírt parancssori kapcsolókat és paramétereket használhatjuk, melyek közül talán csak `-i<n>` kapcsolót kell részletesen bemutatni, mellyel kézzel adhatjuk meg, hogy a program milyen összehasonlító metódust használjon. Ezek a következők:

0. általános célú regiszterekkel, `popcnt` utasítás nélküli
1. általános célú regiszterekkel, `popcnt` utasítást használó
2. SSE, xmm regisztereket használó, `popcnt` utasítás nélküli, adatokat röptében feldolgozó
3. SSE, xmm regisztereket, `popcnt` utasítást használó, adatokat röptében feldolgozó
4. SSE, xmm regisztereket használó, `popcnt` utasítás nélküli, adatokat előfeldolgozó
5. SSE, xmm regisztereket, `popcnt` utasítást használó, adatokat előfeldolgozó

(ezen kívül még néhány tesztelési verzió is használható)

Az automata algoritmus kiválasztó figyelembe veszi a futás során lekérdezett cpu lehetőségeket, és a fordítási platformot is (32/64 bit). A processzor `popcnt` utasítás meglétének vizsgálatára külön `cpuid` parancsot használó cpu információs modult építettünk a programba. Lehetőség van a modul által detektált részletek megtekintésére is a `-vi` kapcsolóval.

Lehetőség van az összehasonlítás folyamatának nyomon követésére a `-vc` kapcsolóval, mely hatására a parancs sorszámát, összehasonlítás során talált egyezések számát írja ki a program.

Az eredmények fájlba mentésekor a parancs soronkénti adatméretet tudjuk a `-vs` kapcsolóval kiírni.

Az utóbbi 3 kapcsoló tetszőleges kombinációban használható. Pl.: `-vics`

A file paraméterek kapcsolókkal is megadhatók, vagy kapcsoló nélkül sorrendi alapon, vagy akár keverten. Pl.:

- `sr cmd.bin data.bin`
- `sr -oout.bin cmd.bin`
- ...

További konfigurálásra fordítás időben `#define`-ok segítségével van lehetőség.



5 Teszt eredmények

A program tesztelése során két különböző konfigurációt használtunk a működés ellenőrzésére, és a teljesítmény mérésére.

5.1 Tesztkörnyezet AMD

Mainboard Model:

- M4A785TD-V EVO

Processor:

- Name AMD Phenom II X3 705
- Number of cores 3 (max 3)
- Number of threads 3 (max 3)
- Stock frequency 2500 MHz
- Instructions sets MMX (+), 3DNow! (+), SSE, SSE2, SSE3, SSE4A, x86-64, AMD-V

Memory:

- Memory type DDR3
- Size 2x2048 MBytes
- Max bandwidth PC3-10700H (667 MHz)

Software:

- Microsoft Windows Server 2003 Enterprise Edition Service Pack 2 (Build 3790)

További adatok a [lemez mellékletéről](#) érhetők el.

5.2 Tesztkörnyezet INTEL

Mainboard Model:

- Gigabyte EP45T-DS3R

Processor:

- Name Intel Core 2 Duo E8400
- Number of cores 2 (max 2)
- Number of threads 2 (max 2)
- Stock frequency 3000 MHz
- Instructions sets MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, EM64T, VT-x

Memory:

- Memory type DDR3
- Size 2x2048 MBytes
- Max bandwidth PC3-8500F (533 MHz)

Software:

- Microsoft Windows XP x64 Professional Service Pack 2 (Build 3790)

További adatok a [lemez mellékletéről](#) érhetők el.



5.3 Futtatási eredmények

A sebesség tesztek során a konzultációk során kapott bemeneti állományokat használva, különböző összehasonlítási metódusok mellett mértük a futási időket, 32 és 64 bites kódon (mindkét esetben 64 bites operációs rendszer alatt). A kapott eredményeket a következő táblázatban foglaltuk össze.

	-i 0	-i 1	-i 2	-i 3	-i 4	-i 5
AMD 64 bit	9.7s	2.95s	5.18s	3.7s	4s	3.6s
AMD 32 bit	29s	8.4s	7.9s	10.8s	9s	8.5s
INTEL 64 bit	8.89s	na.	4.56s	na.	4.34s	na.
INTEL 32 bit	30.25s	na.	8.29s	na.	7.85s	na.

A legjobb futási eredményeket AMD tesztkörnyezetben az általános célú regiszterekkel, popcnt utasítást használó metódussal értük el, míg az INTEL környezetben az SSE, xmm regisztereket használó, popcnt utasítás nélküli, adatokat előfeldolgozó metódus nyújtotta a legjobb eredményt.



6 Irodalomjegyzék

6.1 Bioinformatika:

1. **Local alignment of two-base encoded DNA sequence**
<http://www.biomedcentral.com/1471-2105/10/175>
2. **Application note di-base sequencing and the advantage of color space analysis**
http://marketing.appliedbiosystems.com/images/Product_Microsites/Solid_Knowledge_MS/pdf/SOLiD_Dibase_Sequencing_and_Color_Space_Analysis.pdf
3. **Jeremy Buhler - CSE587 Short Read Class 1-3**
<http://warlord.wustl.edu/Alignment/lec1.pdf>
<http://warlord.wustl.edu/Alignment/lec2.pdf>
<http://warlord.wustl.edu/Alignment/lec3.pdf>
4. **Genome Assembly with Short Reads**
<http://www.cbcb.umd.edu/research/SR-assembly.shtml>
5. **Bowtie - An ultrafast memory-efficient short read aligner**
<http://bowtie-bio.sourceforge.net/index.shtml>
6. **BWA Burrows-Wheeler Aligner**
<http://bio-bwa.sourceforge.net/index.shtml>
7. **Burrows–Wheeler transform**
http://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform
8. **BioPerl**
http://www.bioperl.org/wiki/Main_Page
9. **Uri Keich, Ming Li, Bin Ma, John Tromp – On Spaced Seeds for Similarity Search**
<http://www.bioinformatics.uwaterloo.ca/papers/04seeds.pdf>
10. **Dezoxiribonukleinsav**
<http://hu.wikipedia.org/wiki/Dezoxiribonukleinsav>

6.2 Optimalizálás, CPU referencia, algoritmus:

11. **248966 - Intel 64 and IA-32 Architectures Optimization Reference Manual**
<http://www.intel.com/Assets/PDF/manual/248966.pdf>
12. **253665 - Intel 64 and IA-32 Architectures Software Developer's Manual Vol1**
<http://www.intel.com/assets/PDF/manual/253665.pdf>
13. **253666 - Intel 64 and IA-32 Architectures Software Developer's Manual Vol2A**
<http://www.intel.com/assets/pdf/manual/253666.pdf>
14. **253667 - Intel 64 and IA-32 Architectures Software Developer's Manual Vol2B**
<http://www.intel.com/Assets/pdf/manual/253667.pdf>



15. **253668 - Intel 64 and IA-32 Architectures Software Developer's Manual Vol3A**
<http://www.intel.com/Assets/PDF/manual/253668.pdf>
16. **253669 - Intel 64 and IA-32 Architectures Software Developer's Manual Vol3B**
<http://www.intel.com/assets/pdf/manual/253669.pdf>
17. **318148 - Intel 64 Architecture x2APIC Specification**
<http://www.intel.com/Assets/pdf/manual/318148.pdf>
18. **D91561 - Intel SSE4 Programming Reference**
<http://software.intel.com/file/18187/>
19. **40546 - Software Optimization Guide for AMD Family 10h Processors**
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf
20. **24594 - AMD64 Architecture Programmer's Manual Volume 3- General-Purpose and System Instructions**
http://support.amd.com/us/Processor_TechDocs/24594.pdf
21. **22007 - AMD Athlon Processor x86 Code Optimization Guide**
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf
22. **Agner Fog - Software optimization resources**
<http://www.agner.org/optimize/#manuals>
23. **Frank de Groot – magic popcount (popcnt) command**
<http://popcnt.org/2007/09/magic-popcount-popcnt-command.html>
24. **Cacheline splits, aka Intel hell**
<http://x264dev.blogspot.com/2008/05/cacheline-splits-aka-intel-hell.html>

6.3 Ábrák és egyéb felhasznált anyagok:

25. **A DNS molekula ábrája**
http://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/DNA_chemical_structure.svg/350px-DNA_chemical_structure.svg.png



7 Lemez melléklet