CUDA bemutató

Adatbányászat és Webes Keresés Kutatócsoport

SZTAKI

2010

Tartalom

Tartalom

GPU-k és a CUDA

Előzmények

- grafikus kártyák: nagy párhuzamos számítási kapacitás
- eredetileg csak grafikus műveleteket tudtak végezni

GPU-k és a CUDA

Előzmények

- grafikus kártyák: nagy párhuzamos számítási kapacitás
- eredetileg csak grafikus műveleteket tudtak végezni

Compute Unified Device Architecture (CUDA)

- a „CUDA-enabled” NVIDIA GPU-kat lehet programozni
- használhatók a kártya multiprocesszorai és a memóriája
- Tesla sorozat: már nincs is kép kimenet

Execution model

- a CUDA kód végrehajtása párhuzamos thread-ekben történik
- a thread-ek block-okba vannak rendezve
- egy block thread-jei együtt dolgoznak
- az egyes block-ok egymástól függetlenül dolgoznak
- technikai részlet: valójában a thread-ek 32-esével futnak együtt (ez egy warp)

Execution model

- a CUDA kód végrehajtása párhuzamos thread-ekben történik
- a thread-ek block-okba vannak rendezve
- egy block thread-jei együtt dolgoznak
- az egyes block-ok egymástól függetlenül dolgoznak
- technikai részlet: valójában a thread-ek 32-esével futnak együtt (ez egy warp)

Példa: Kép szegmenseinek feldolgozása

- minden block egy külön szegmenst dolgoz fel
- egy block-on belül a thread-ek együtt, párhuzamosan számolják pl. a szegmens átlag színét

Memória fajták I.

global

- a GPU hagyományos értelemben vett memóriája
- ide/innen tud adatot másolni a host
- GB-os nagyságrend
- viszonylag lassú adatátvitel, lassú elérés a GPU-ról, nincs cache

Memória fajták I.

global

- a GPU hagyományos értelemben vett memóriája
- ide/innen tud adatot másolni a host
- GB-os nagyságrend
- viszonylag lassú adatátvitel, lassú elérés a GPU-ról, nincs cache

shared

- egy block thread-jeinek a közös memóriája
- itt kommunikálnak egymással a thread-ek, valamint itt érdemes tárolni a sokszor elért adatot
- 16KB (/block)
- nagyon gyors elérés

Memória fajták II.

constant

- kicsi, gyors, cache-elt elérésű memória
- csak a host tudja módosítani

Memória fajták II.

constant

- kicsi, gyors, cache-elt elérésű memória
- csak a host tudja módosítani

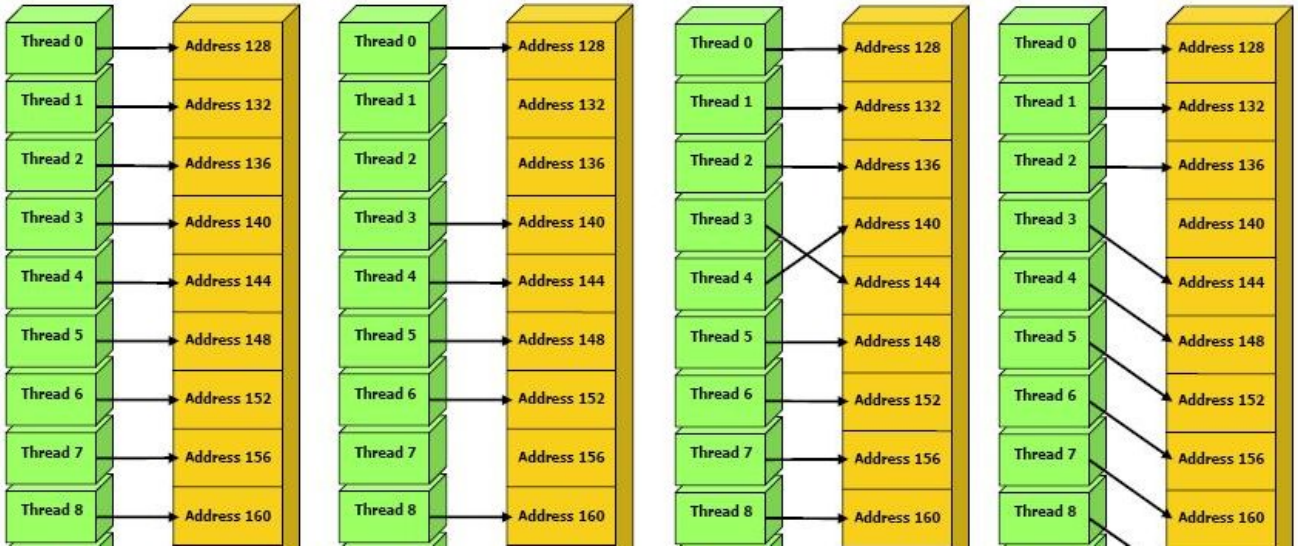
regiszterek

- minden block rendelkezésére áll néhány regiszter, amiket a thread-ek között oszt szét
- általában csak ciklus- és temp változókat tárol

Memória elérési trükkök, szabályok

- a globális memória elérésének sok szabálya van, amik betartása felgyorsítja azt (ld. ábra)
- globális memóriaterületeket texture-nek lehet kinevezni cache-elt elérés, viszont nem írható
- non-pageable host memória: gyorsabb másolás

Két-két példa coalesced és non-coalesced elérésre:



A CUDA programok nyelve

a CUDA kód lényegében C kód, néhány kiegészítéssel

szintaktikai kiegészítések

- új függvény qualifierek
- gpu-s függvények hívása

A CUDA programok nyelve

a CUDA kód lényegében C kód, néhány kiegészítéssel

szintaktikai kiegészítések

- új függvény qualifierek
- gpu-s függvények hívása

új host függvények

pl. gpu memória foglalás, másolás, stb.

A CUDA programok nyelve

a CUDA kód lényegében C kód, néhány kiegészítéssel

szintaktikai kiegészítések

- új függvény qualifierek
- gpu-s függvények hívása

új host függvények

pl. gpu memória foglalás, másolás, stb.

GPU-specifikus függvények, változók

- threadIdx, blockIdx, stb.
- thread-ek szinkronizálása
- gyors matematikai függvények (exp, sin, stb.)

CUDA kód futtatása

a GPU memóriája közvetlenül nem elérhető, így egy CUDA programrészlet futtatása általában így néz ki:

1. GPU memória allokálás
2. adat másolása a host-ról az allokált memóriába
3. GPU függvény hívása, ami feldolgozza az adatot
4. az eredmény másolása a host-ra
5. (GPU memória felszabadítás)

Egy nagyon egyszerű példa

```
__global__ void vecmult(float *v, float m) {
  int i = threadIdx.x;
  v[i] = v[i] * m;
}

int main() {
  int N=100;
  float *v_h;
  ... //v_h allokálás, inicializálás
  float *v_d;
  cudaMalloc((void**)&v_d, N*sizeof(float));
  cudaMemcpy(v_d, v_h, N*sizeof(float), cudaMemcpyHostToDevice);
  vecmult<<<1,N>>>(v_d, m); //kernel hívás: 1 block, N thread
  cudaMemcpy(v_h, v_d, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  return 0;
}
```

Egy nagyon egyszerű példa

```
__global__ void vecmult(float *v, float m) {
  int i = threadIdx.x;
  v[i] = v[i] * m;
}

int main() {
  int N=100;
  float *v_h;
  ... //v_h allokálás, inicializálás
  float *v_d;
  cudaMalloc((void**)&v_d, N*sizeof(float));
  cudaMemcpy(v_d, v_h, N*sizeof(float), cudaMemcpyHostToDevice);
  vecmult<<<1,N>>>(v_d, m); //kernel hívás: 1 block, N thread
  cudaMemcpy(v_h, v_d, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  return 0;
}
```

Egy nagyon egyszerű példa

```
__global__ void vecmult(float *v, float m) {
  int i = threadIdx.x;
  v[i] = v[i] * m;
}

int main() {
  int N=100;
  float *v_h;
  ... //v_h allokálás, inicializálás
  float *v_d;
  cudaMalloc((void**)&v_d, N*sizeof(float));
  cudaMemcpy(v_d, v_h, N*sizeof(float), cudaMemcpyHostToDevice);
  vecmult<<<1,N>>>(v_d, m); //kernel hívás: 1 block, N thread
  cudaMemcpy(v_h, v_d, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  return 0;
}
```

Egy nagyon egyszerű példa

```
__global__ void vecmult(float *v, float m) {
  int i = threadIdx.x;
  v[i] = v[i] * m;
}

int main() {
  int N=100;
  float *v_h;
  ... //v_h allokálás, inicializálás
  float *v_d;
  cudaMalloc((void**)&v_d, N*sizeof(float));
  cudaMemcpy(v_d, v_h, N*sizeof(float), cudaMemcpyHostToDevice);
  vecmult<<<1,N>>>(v_d, m); //kernel hívás: 1 block, N thread
  cudaMemcpy(v_h, v_d, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  return 0;
}
```

Egy nagyon egyszerű példa

```
__global__ void vecmult(float *v, float m) {
  int i = threadIdx.x;
  v[i] = v[i] * m;
}

int main() {
  int N=100;
  float *v_h;
  ... //v_h allokálás, inicializálás
  float *v_d;
  cudaMalloc((void**)&v_d, N*sizeof(float));
  cudaMemcpy(v_d, v_h, N*sizeof(float), cudaMemcpyHostToDevice);
  vecmult<<<1,N>>>(v_d, m); //kernel hívás: 1 block, N thread
  cudaMemcpy(v_h, v_d, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  return 0;
}
```

Kernel függvények

- kernel: a GPU-n futó függvény
- mindig `__global__ void` típusú
- `kernelnev<<<blocks,threads>>>(parameterek)`
- `blocks`, `threads` lehetnek int-ek, vagy `dim3` típusúak, ahol `dim3.x`,`dim3.y`,`dim3.z` adják meg a dimenziókat
- kernel kódban a `dim3` típusú `threadIdx`, `blockDim` és `gridDim` változók elérhetők
- a kernel hívás aszinkron (rögtön folytatódik a host kód végrehajtása)

Kernel függvények

- kernel: a GPU-n futó függvény
- mindig `__global__ void` típusú
- `kernelnev<<<blocks,threads>>>(parameterek)`
- `blocks`, `threads` lehetnek int-ek, vagy `dim3` típusúak, ahol `dim3.x`,`dim3.y`,`dim3.z` adják meg a dimenziókat
- kernel kódban a `dim3` típusú `threadIdx`, `blockDim` és `gridDim` változók elérhetők
- a kernel hívás aszinkron (rögtön folytatódik a host kód végrehajtása)
- kernel csak a host-ról hívható
- kernelben nem lehet rekurzió
- nem elérhetők a host függvények, host memória, stb.

shared memória kezelés, szinkronizálás

- shared változó (vagy tömb) a `__shared__` qualifier-rel deklarálható
- shared változók egy block-on belül minden thread számára elérhetők
- egy block thread-jei a `__syncthreads()` paranccsal szinkronizálhatók

shared memória kezelés, szinkronizálás

- shared változó (vagy tömb) a `__shared__` qualifier-rel deklarálható
- shared változók egy block-on belül minden thread számára elérhetők
- egy block thread-jei a `__syncthreads()` paranccsal szinkronizálhatók

Példák

- `__shared__ int var; //változó`
- `__shared__ int arr[10]; //tömb`

shared memória példa

```
__global__ void kernelfv(int N, int *array) {
  const int tid=threadIdx.x;
  __shared__ int sharedarray[MAXN];

  if (tid < N)
    sharedarray[tid]=array[tid];


  if (tid < N-1) {




    sharedarray[tid+1]+=sharedarray[tid];
  }

  ...
}
```

shared memória példa

```
__global__ void kernelfv(int N, int *array) {
  const int tid=threadIdx.x;
  __shared__ int sharedarray[MAXN];

  if (tid < N)
    sharedarray[tid]=array[tid];
  __syncthreads();

  if (tid < N-1) {




    sharedarray[tid+1]+=sharedarray[tid];
  }
  __syncthreads();
  ...
}
```

shared memória példa

```
__global__ void kernelfv(int N, int *array) {
  const int tid=threadIdx.x;
  __shared__ int sharedarray[MAXN];

  if (tid < N)
    sharedarray[tid]=array[tid];
  __syncthreads();

  if (tid < N-1) {
    int tmp=sharedarray[tid];



    sharedarray[tid+1]+=tmp;
  }
  __syncthreads();
  ...
}
```

shared memória példa

```
__global__ void kernelfv(int N, int *array) {
  const int tid=threadIdx.x;
  __shared__ int sharedarray[MAXN];

  if (tid < N)
    sharedarray[tid]=array[tid];
  __syncthreads();

  if (tid < N-1) {
    int tmp=sharedarray[tid];

  __syncthreads();

    sharedarray[tid+1]+=tmp;
  }
  __syncthreads();
  ...
}
```

shared memória példa

```
__global__ void kernelfv(int N, int *array) {
  const int tid=threadIdx.x;
  __shared__ int sharedarray[MAXN];

  if (tid < N)
    sharedarray[tid]=array[tid];
  __syncthreads();

  if (tid < N-1) {
    int tmp=sharedarray[tid];
  }
  __syncthreads();
  if (tid < N-1) {
    sharedarray[tid+1]+=tmp;
  }
  __syncthreads();
  ...
}
```

Tartalom

Tartalom

Bevezetõ

SVD: Singular Value Decomposition

A feladat

Egy $n \times n$ méretû M szimmetrikus, ritka mátrixnak keressük a k legnagyobb szinguláris értékét és a hozzájuk tartozó szinguláris vektorokat, ahol $k \ll n$.

Bevezetõ

SVD: Singular Value Decomposition

A feladat

Egy $n \times n$ méretû M szimmetrikus, ritka mátrixnak keressük a k legnagyobb szinguláris értékét és a hozzájuk tartozó szinguláris vektorokat, ahol $k \ll n$.

Alkalmazások

- gráfok spektrál klaszterezése
- mátrixok kis rangú közelítése

Algoritmus

Algoritmus

Ezt a feladatot megoldja a Lánczos-módszer, ami implementálva van a LAPACK programcsomagban.
Az algoritmus nagyrészt csak vektormûveleteket (lineáris kombináció, skalárszorzat, stb.) és mátrix-vektor szorzásokat végez. Ezek nyilván jól párhuzamosíthatók.

Algoritmus

Algoritmus

Ezt a feladatot megoldja a Lánczos-módszer, ami implementálva van a LAPACK programcsomagban.
Az algoritmus nagyrészt csak vektormûveleteket (lineáris kombináció, skalárszorzat, stb.) és mátrix-vektor szorzásokat végez. Ezek nyilván jól párhuzamosíthatók.

CUDA módosítások

1. vektormûveletek átírása CUDA-ra
2. ritka mátrix-vektor szorzás kernel
3. memóriakezelés

CUDA implementáció, CUBLAS

CUBLAS

A CUBLAS a BLAS lineáris algebra programcsomag CUDA implementációja. A mátrix-vektor szorzáson kívül minden megvan benne.

CUDA implementáció, CUBLAS

CUBLAS

A CUBLAS a BLAS lineáris algebra programcsomag CUDA implementációja. A mátrix-vektor szorzáson kívül minden megvan benne.

Mátrix-vektor szorzás

1. minden block a szorzat egy szeletét számolja ki
2. az egyes block-ok soronként számolják az eredményt
3. a k. thread az adott sor $k, 2k, \ldots$-adik elemeivel számol
4. a sor végére érve a thread-ek összesítik az eredményt (scan: ld. később)

Memória gondok

- a használt GPU memória 2 GB, ennél nagyobb mátrixokra is szerettünk volna futtatni
- megoldás: concurrent copy and execution
- a GPU egyszerre számol az adat egy szeletével, és másolja a GPU memóriába a következő szeletet

Memória gondok

- a használt GPU memória 2 GB, ennél nagyobb mátrixokra is szerettünk volna futtatni
- megoldás: concurrent copy and execution
- a GPU egyszerre számol az adat egy szeletével, és másolja a GPU memóriába a következő szeletet
- minden mátrix szorzásnál az egész adatot meg kell mozgatni, ez túl lassú (mert túl sok a mátrix szorzás az algoritmusban)

Eredmények

Méret, elemek	CPU	CUDA
10.000×10.000, 1.000.000	65 sec	5 sec
10.000×10.000, 4.000.000	191 sec	8 sec
20.000×20.000, 1.600.000	88 sec	28 sec
20.000×20.000, 4.000.000	207 sec	35 sec
20.000×20.000, 16.000.000	810 sec	53 sec

Tartalom

Bevezetõ

A feladat

Egy képet szeretnénk feldarabolni nagyjából k részre úgy, hogy az egyes részek „összefüggők” legyenek.

Bevezetõ

A feladat

Egy képet szeretnénk feldarabolni nagyjából k részre úgy, hogy az egyes részek „összefüggők” legyenek.

Naiv algoritmus

Harish et al. Fast Minimum Spanning Tree for Large Graphs on the GPU

- iteratív minimális feszítőfa algoritmus
- kezdetben minden pont egy fa, minden iterációban szomszédos fákat egyesít
- alkalmazható (kezdetleges) kép szegmentálásra
- scan és sort műveletekből épül fel

Naiv algoritmus

Harish et al. Fast Minimum Spanning Tree for Large Graphs on the GPU

- iteratív minimális feszítőfa algoritmus
- kezdetben minden pont egy fa, minden iterációban szomszédos fákat egyesít
- alkalmazható (kezdetleges) kép szegmentálásra
- scan és sort műveletekből épül fel

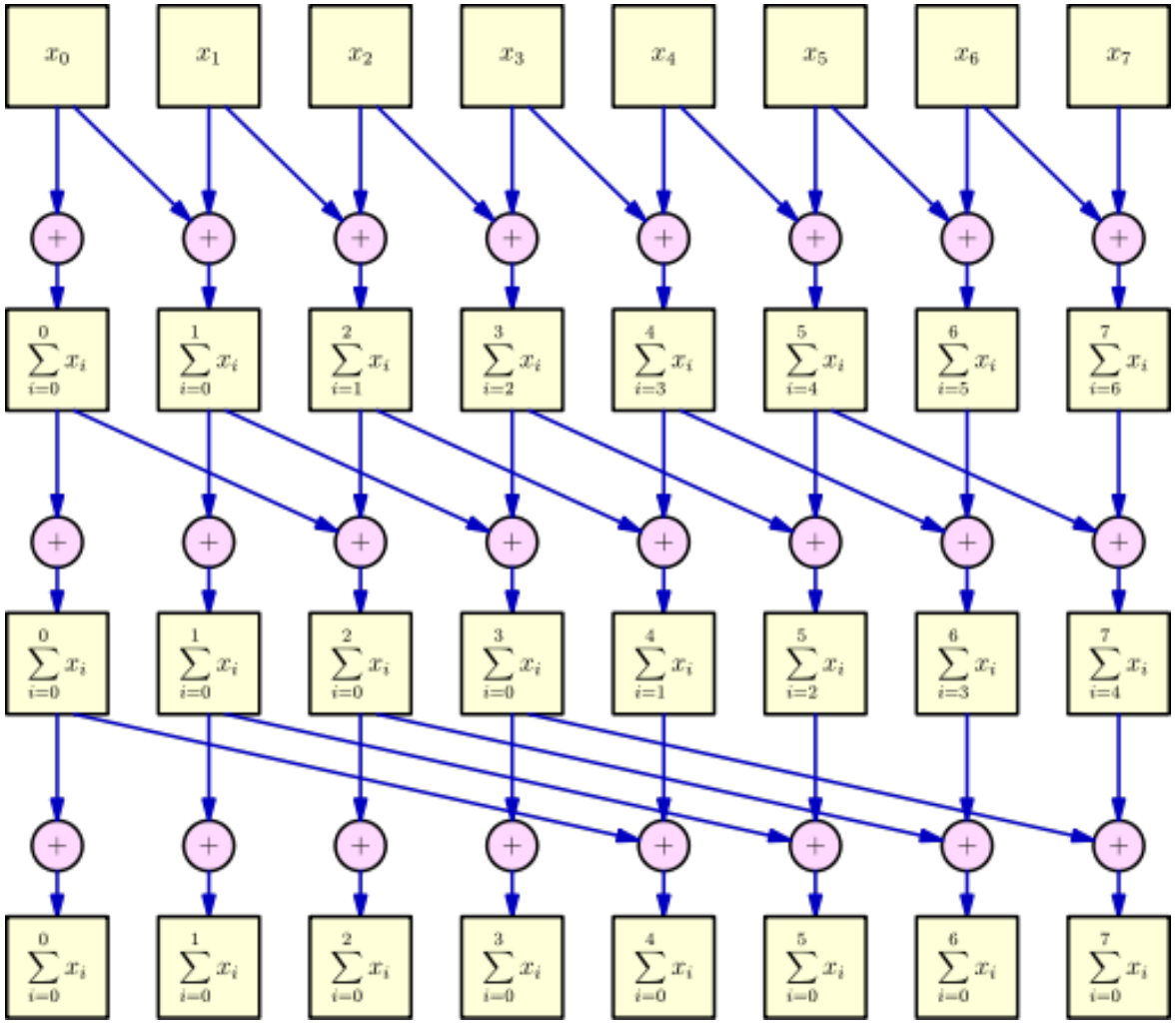
CUDPP Library

A CUDPP a sort és scan műveletek CUDA implementációját tartalmazó library

Scan

- prefix összeg: $prfsum[i] = \sum_{j=0}^{i} v[i]$
- a scan a prefix összeg általánosítása: $+$ helyett φ asszoc. művelet
- $scan[i] = \varphi\left(scan[i-1], v[i]\right)$
- ebben az algoritmusban csak $+$ és min -scan
- nagyon sokszor alkalmazható
- jól párhuzamosítható
- segmented scan: egy vektort szeletenként scan-elünk (de ezt egy scan művelet végzi el)

Párhuzamos scan



Az algoritmus részletei

Egy iteráció lépései:

1. segmented minscan az (élsúly, végpont) párokra
   $\Rightarrow$ minden pont legközelebbi, legkisebb címkéjű szomszédja
   $\Rightarrow$ successor graph $\mathcal{S}$

Az algoritmus részletei

Egy iteráció lépései:

1. segmented minscan az (élsúly, végpont) párokra
   $\Rightarrow$ minden pont legközelebbi, legkisebb címkéjű szomszédja
   $\Rightarrow$ successor graph $\mathcal{S}$
2. $\mathcal{S}$-ben 2 hosszú körök lehetnek, ezeket eltávolítjuk
3. $\mathcal{S}$-ben pointer ugrással mindenki a reprezántánsára mutat

Az algoritmus részletei

Egy iteráció lépései:

1. segmented minscan az (élsúly, végpont) párokra
   $\Rightarrow$ minden pont legközelebbi, legkisebb címkéjű szomszédja
   $\Rightarrow$ successor graph $\mathcal{S}$
2. $\mathcal{S}$-ben 2 hosszú körök lehetnek, ezeket eltávolítjuk
3. $\mathcal{S}$-ben pointer ugrással mindenki a reprezántánsára mutat
4. reprezentáns szerint rendezünk, flag-ekkel bejelöljük a határokat
5. +scan-eljük a flag-eket $\Rightarrow$ új címkék

Az algoritmus részletei

Egy iteráció lépései:

1. segmented minscan az (élsúly, végpont) párokra
   $\Rightarrow$ minden pont legközelebbi, legkisebb címkéjű szomszédja
   $\Rightarrow$ successor graph $\mathcal{S}$
2. $\mathcal{S}$-ben 2 hosszú körök lehetnek, ezeket eltávolítjuk
3. $\mathcal{S}$-ben pointer ugrással mindenki a reprezántánsára mutat
4. reprezentáns szerint rendezünk, flag-ekkel bejelöljük a határokat
5. +scan-eljük a flag-eket $\Rightarrow$ új címkék
6. (kezdőpont, végpont, élsúly) hármasok rendezésével megkapjuk az új gráf éleit

Végső változat, eredmények

- az új gráf éleinek létrehozásakor nem a minimális súlyt választjuk, hanem összeadjuk a megfelelő élek súlyát $\Rightarrow$ átlag súly
- ez szintén megy scan-nel
- a szegmensek belsejében is sok minden számolható scan-nel (pl. átlag szín)

egy kép 100 szegmensre vágása kb. 3-4 sec.

Tartalom

Bevezetõ

Gaussian Mixture Model

- azt feltételezzük, hogy a klaszterezendő pontok K db. d-dim. Gauss eloszlásból származnak
- p_i valószínűséggel kiválasztjuk az i-edik Gauss-t, majd ebből sorsolunk egy pontot az $\mathcal{N}_i(\mu_i, \sigma_i)$ eloszlás szerint
- a modell paraméterei: p_i, μ_i, σ_i

Bevezetõ

Gaussian Mixture Model

- azt feltételezzük, hogy a klaszterezendő pontok K db. d-dim. Gauss eloszlásból származnak
- p_i valószínűséggel kiválasztjuk az i-edik Gauss-t, majd ebből sorsolunk egy pontot az $\mathcal{N}_i(\mu_i, \sigma_i)$ eloszlás szerint
- a modell paraméterei: p_i, μ_i, σ_i

Alkalmazások, előnyök

- „finom” klaszterezés (csak valószínűségeket mond)
- a k-means a GMM egy spec. esete ($\sigma_i \sim 0$)
- átfedő klaszterek detektálása, stb.

EM algoritmus, párhuzamosítás

EM algoritmus

iteratív algoritmus

1. kiszámolja a $p_{nk} = \mathbb{P}(n.\ \text{pont a}\ k.\ \text{klaszterbe tartozik})$ valószínűségeket
2. p_{nk}-k alapján frissíti a modell paramétereit

a likelihood-fv. lok. maximumába konvergál

EM algoritmus, párhuzamosítás

EM algoritmus

iteratív algoritmus

1. kiszámolja a $p_{nk} = \mathbb{P}(n.\ \text{pont a}\ k.\ \text{klaszterbe tartozik})$ valószínűségeket
2. p_{nk}-k alapján frissíti a modell paramétereit

a likelihood-fv. lok. maximumába konvergál

Párhuzamosítás

- a klaszterek paraméterei egymástól függetlenül frissíthetők
- egy klaszteren belül is sok számolás kell
- $\Rightarrow$ minden blokk egy klasztert számol
- ez csak addig működik, amíg $N * K$ elég kicsi (mert így az összes p_{nk}-nak egyszerre rendelkezésre kell állni)

Részletek - algoritmus, numerikus trükkök

Memóriagazdaságos párhuzamosítás

- az algoritmus nagy része sok tagú összegek számolásából áll
- az 1 block/klaszter munkamegosztás helyett egymás után dolgozzuk fel a klasztereket
- az összeg egy-egy részét számolják a block-ok
- a végén a block-ok részeredményeit egy újabb kernel összeadja

Részletek - algoritmus, numerikus trükkök

Memóriagazdaságos párhuzamosítás

- az algoritmus nagy része sok tagú összegek számolásából áll
- az 1 block/klaszter munkamegosztás helyett egymás után dolgozzuk fel a klasztereket
- az összeg egy-egy részét számolják a block-ok
- a végén a block-ok részeredményeit egy újabb kernel összeadja

Numerikus trükkök

- normális eo. sűrűségfv.-eket kell számolni
- ez sok alulcsorduláshoz vezet
- $\Rightarrow$ logaritmusokkal kell számolni
- összegzés: $\log\left(\sum e^{z_i}\right) = z_{max} + \log\left(\sum e^{z_i - z_{max}}\right)$

Részletek - memóriakezelés

Újraszámolás

- az eredeti EM algoritmusban ki kell számolni az összes p_{nk} valószínűséget, majd frissíteni a modellt
- GPU-n ezek tárolása túl sok memória
- nem tároljuk, hanem mindig újra kiszámoljuk; ez elhanyagolható pluszmunka

Részletek - memóriakezelés

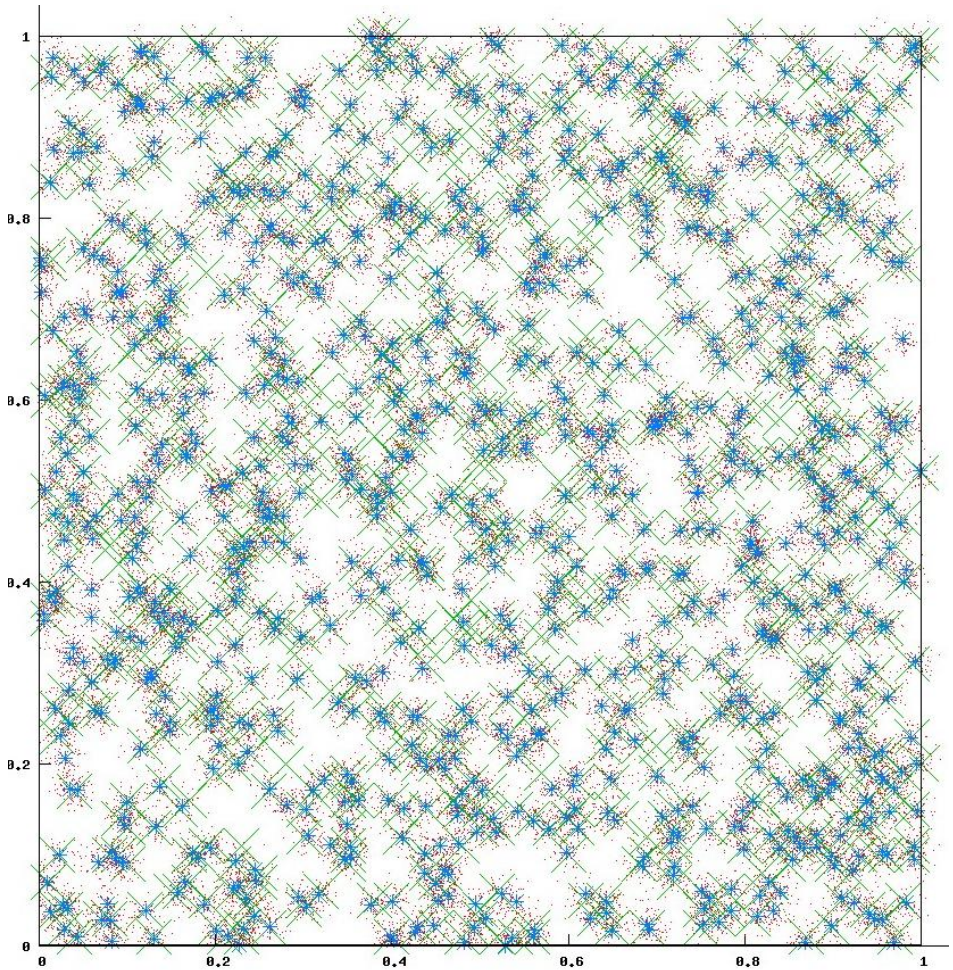
Újraszámolás

- az eredeti EM algoritmusban ki kell számolni az összes p_{nk} valószínűséget, majd frissíteni a modellt
- GPU-n ezek tárolása túl sok memória
- nem tároljuk, hanem mindig újra kiszámoljuk; ez elhanyagolható pluszmunka

Nagy adat kezelése

- az adat általában túl nagy a GPU memóriához
- az SVD-nél alkalmazott trükk itt működik!
- az adatot feldaraboljuk, és amíg a GPU számol egy szeleten, addig párhuzamosan másoljuk a következőt
- itt elég sok a számítás ahhoz, hogy ez gyors legyen

Eredmények



nagy adaton kb. 70-szeres gyorsulás!