Lineáris egyenletrendszerek

Lineáris egyenletrendszer

- Lineáris egyenlet
  - algebrai egyenlet
  - konstansok és első fokú ismeretlenek
  - pl.: egyenes egyenlete $y = ax + b$

- Lineáris egyenletrendszer
  - lineáris egyenletek csoportja
  - ugyan azon a változó halmazon

Jakobi iteráció

- Lineáris egyenletrendszer

$$\underline{\underline{B}}\,\underline{x} = \underline{b}$$

$$\underline{x} + \underline{\underline{B}}\,\underline{x} = \underline{x} + \underline{b}$$

$$\underline{x} = \underbrace{\left(\underline{\underline{1}} - \underline{\underline{B}}\right)}_{\underline{\underline{A}}}\underline{x} + \underline{b}$$

$$\longrightarrow \quad \boxed{\underline{x} = \underline{\underline{A}}\,\underline{x} + \underline{b}}$$

Jakobi iteráció

- Mikor oldható meg?

$$\underline{x} = \underline{\underline{A}}\,\underline{x} + \underline{b}$$

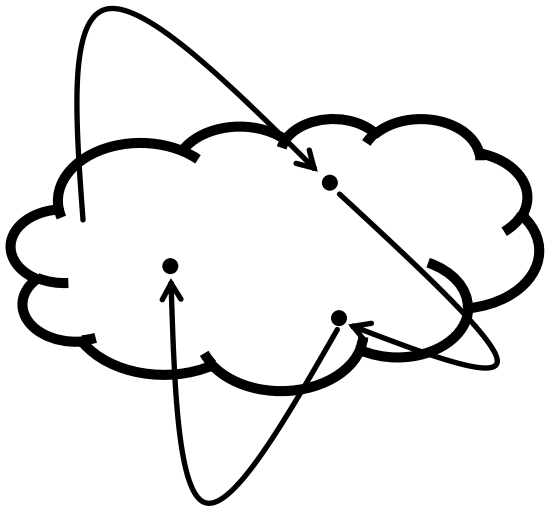
  - Ha konvergens

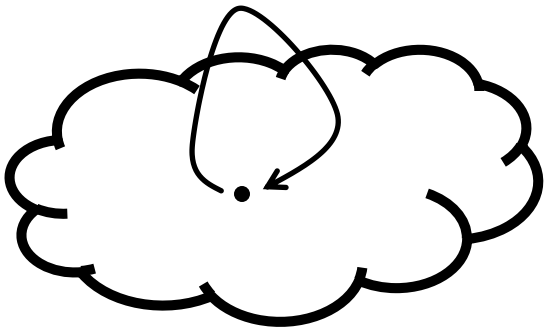
$$\underline{x}_{n+1} = \underline{\underline{A}}\,\underline{x}_n + \underline{b}$$

$$\lim x_{n+1} \to x^*$$

$$\lim x_n \to x^*$$

$$\underline{x}^* = \underline{\underline{A}}\,\underline{x}^* + \underline{b}$$





Fix pont

Jakobi iteráció

- Mikor konvergens?
  - Ha A kompatibilis x valamely normájával.
  - Norma
    - Egy vektorteren értelmezett leképzés, amely a nullvektor kivételével a tér minden vektorához egy pozitív számot rendel.

$$d(x) >= 0$$

$$d(x) = 0 \quad \text{a.cs.a., ha} \quad x = 0$$

$$d(x+y) <= d(x) + d(y)$$

$$d(\lambda x) = |\lambda| d(x)$$

Jakobi iteráció

- Normák
  - p-norma (Hölder-norma)

$$\|x\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

  - 1-norma

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

  - 2-norma

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

  - Végtelen norma

$$\|x\|_\infty = \max_{i=1}^{n} |x_i|$$

Jakobi iteráció

- Mátrixnormák
  - A vektornormák mátrixnormákat indukálnak.

$$\|A\|_M = \sup_{x \neq 0} \frac{\|Ax\|_v}{\|x\|_v} = \sup_{\|x\|_{v=1}} \|Ax\|_v$$

    - Linearitás miatt elég az 1 normájú vektorokat tekinteni.
    - Kompakt halmaz, így a folytonos $\|Ax\|_v$ függvény felveszi a maximumát.
  - A mátrixnormárakra teljesül

$$\|A \cdot B\| \leq \|A\| \cdot \|B\|$$

$$\|Ax\|_v \leq \|A\|_M \cdot \|x\|_v$$

Jakobi iteráció

- Iterációs megoldás

$$\underline{x}_{n+1} = \underline{\underline{A}}\,\underline{x}_n + \underline{b}$$

$$\underline{x}_n = \underline{\underline{A}}\,\underline{x}_{n-1} + \underline{b}$$

  - Bizonyítás

$$\left(\underline{x}_{n+1} - \underline{x}_n\right) = \underline{\underline{A}}\left(\underline{x}_n - \underline{x}_{n-1}\right)$$

$$\left\|\underline{x}_{n+1} - \underline{x}_n\right\| = \left\|\underline{\underline{A}}\left(\underline{x}_n - \underline{x}_{n-1}\right)\right\| \le \underbrace{\left\|\underline{\underline{A}}\right\|}_{\lambda} \cdot \left\|\underline{x}_n - \underline{x}_{n-1}\right\| \qquad \left\|A\right\| < 1$$

$$0 \le \underbrace{\left\|\underline{x}_{n+1} - \underline{x}_n\right\|}_{\downarrow\, 0} \le \lambda \left\|\underline{x}_n - \underline{x}_{n-1}\right\| \le \lambda^2 \left\|\underline{x}_{n-1} - \underline{x}_{n-2}\right\| \le \cdots \le \underbrace{\lambda^n}_{\downarrow\, 0} \left\|\underline{x}_1 - \underline{x}_0\right\|$$

    - Rendőr szabály!

$$\boxed{\underline{0}, \underline{b}, \underline{\underline{A}}\,\underline{b}, \underline{\underline{A}}^2\,\underline{b} \ldots \le \lambda^n \left\|\underline{b}\right\|}$$

Jakobi iteráció

- Iterációs megoldás

```
void jakobi(){
  ...
  int inputBuffer = 0;
  const int iterations = 20;
  for(int i = 0; i < iterations; ++i){
    mulMatrixVector(n, n, x[(inputBuffer + 1) % 2], A, x[inputBuffer], b);
    inputBuffer = (inputBuffer + 1) % 2;
  }
  ...
}
```

Jakobi iteráció

- Iterációs megoldás

$$\underline{\underline{A}} = 0.5 \cdot \underline{\underline{I}} \qquad \underline{b} = [1] \qquad \underline{x} = [0]$$

```
Jakobi x: [1, 1, 1, 1, 1, 1, 1, 1]
Jakobi x: [1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5]
Jakobi x: [1.75, 1.75, 1.75, 1.75, 1.75, 1.75, 1.75, 1.75]
Jakobi x: [1.875, 1.875, 1.875, 1.875, 1.875, 1.875, 1.875, 1.875]
Jakobi x: [1.9375, 1.9375, 1.9375, 1.9375, 1.9375, 1.9375, 1.9375, 1.9375]
Jakobi x: [1.96875, 1.96875, 1.96875, 1.96875, 1.96875, 1.96875, 1.96875, 1.96875]
Jakobi x: [1.98438, 1.98438, 1.98438, 1.98438, 1.98438, 1.98438, 1.98438, 1.98438]
Jakobi x: [1.99219, 1.99219, 1.99219, 1.99219, 1.99219, 1.99219, 1.99219, 1.99219]
Jakobi x: [1.99609, 1.99609, 1.99609, 1.99609, 1.99609, 1.99609, 1.99609, 1.99609]
Jakobi x: [1.99805, 1.99805, 1.99805, 1.99805, 1.99805, 1.99805, 1.99805, 1.99805]
Jakobi x: [1.99902, 1.99902, 1.99902, 1.99902, 1.99902, 1.99902, 1.99902, 1.99902]
Jakobi x: [1.99951, 1.99951, 1.99951, 1.99951, 1.99951, 1.99951, 1.99951, 1.99951]
Jakobi x: [1.99976, 1.99976, 1.99976, 1.99976, 1.99976, 1.99976, 1.99976, 1.99976]
Jakobi x: [1.99988, 1.99988, 1.99988, 1.99988, 1.99988, 1.99988, 1.99988, 1.99988]
Jakobi x: [1.99994, 1.99994, 1.99994, 1.99994, 1.99994, 1.99994, 1.99994, 1.99994]
Jakobi x: [1.99997, 1.99997, 1.99997, 1.99997, 1.99997, 1.99997, 1.99997, 1.99997]
Jakobi x: [1.99998, 1.99998, 1.99998, 1.99998, 1.99998, 1.99998, 1.99998, 1.99998]
Jakobi x: [1.99999, 1.99999, 1.99999, 1.99999, 1.99999, 1.99999, 1.99999, 1.99999]
Jakobi x: [2, 2, 2, 2, 2, 2, 2, 2]
Jakobi x: [2, 2, 2, 2, 2, 2, 2, 2]
```

Mátrix vektor szorzás

- CPU implementáció

```
void scalarMV(int n, int m,
              float* y, const float* A,
              const float* x, const float* b){
  for(int i=0; i<n; ++i){
    float yi = b[i];
    for(int j=0; j<m; ++j){
      yi += A[i * m + j] * x[j];
    }
    y[i] = yi;
  }
}
```

Mátrix vektor szorzás

- Hogyan párhuzamosítható?
  - Eredmény szálakhoz rendelése
    - Gather típus: minden szál összegzi a bemenet minden elemének hozzájárulását
  - Bemenet szálakhoz rendelése
    - Scatter típus: minden szál kiszámítja a bement egy elemének hozzájárulását a kimenet minden eleméhez
    - Szinkronizáció szükséges!

Mátrix vektor szorzás II.

- Gather típusú megoldás
  - Az eredmény egy N elemű vektor
  - A munka méret N
  - Minden szál kiszámítja a mátrix egy sora és bemeneti vektor alapján az eredmény vektor egy elemét.
  - Ellenőrizni kell a túlcímzést!

Mátrix vektor szorzás II.

- Host program

```
void simpleMV(int n, int m, float* y, const float* A, const float* x, const float* b){
  cl_kernel simpleMVKernel = createKernel(program, "simpleMV");

  cl_mem yGPU = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float)*m, NULL, NULL);
  cl_mem AGPU = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*m*n, NULL, NULL);
  clEnqueueWriteBuffer(commands, AGPU, CL_FALSE, 0, sizeof(float)*m*n, A, 0, NULL, NULL);
  cl_mem xGPU = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * n, NULL, NULL);
  clEnqueueWriteBuffer(commands, xGPU, CL_FALSE, 0, sizeof(float)*n, x, 0, NULL, NULL);
  cl_mem bGPU = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * m, NULL, NULL);
  clEnqueueWriteBuffer(commands, bGPU, CL_FALSE, 0, sizeof(float)*m, b, 0, NULL, NULL);

  clSetKernelArg(simpleMVKernel, 0, sizeof(int), &n);
  clSetKernelArg(simpleMVKernel, 1, sizeof(int), &m);
  clSetKernelArg(simpleMVKernel, 2, sizeof(cl_mem), &yGPU);
  clSetKernelArg(simpleMVKernel, 3, sizeof(cl_mem), &AGPU);
  clSetKernelArg(simpleMVKernel, 4, sizeof(cl_mem), &xGPU);
  clSetKernelArg(simpleMVKernel, 5, sizeof(cl_mem), &bGPU);

  clEnqueueBarrier(commands);
  // ...
```

Mátrix vektor szorzás II.

- Host program

```
  // ...
  size_t workSize = m;
  clEnqueueNDRangeKernel(commands, simpleMVKernel,
                         1, NULL, &workSize, NULL,
                         0, NULL, NULL);

  clFinish(commands);

  clEnqueueReadBuffer(commands, yGPU, CL_TRUE, 0, sizeof(float) * m,
                                          y, 0, NULL, NULL);

  clReleaseMemObject(yGPU);
  clReleaseMemObject(AGPU);
  clReleaseMemObject(xGPU);
  clReleaseMemObject(bGPU);
  clReleaseKernel(simpleMVKernel);
}
```

Mátrix vektor szorzás II.

- OpenCL kernel

```
 __kernel
void simpleMV(const int n, const int m, __global float* y,
              __global float* A, __global float* x,
              __global float* b){

  int i = get_global_id(0);
  if(i < n){
    float yi = b[i];
    for(int j = 0; j < m; ++j){
      yi += A[j + i * m ] * x[j];
    }
    y[i] = yi;
  }

}
```

Soros számítás!

Mátrix vektor szorzás III.

- A skaláris szorzás párhuzamosítása bonyolult
- Az összegzés triviálisan párhuzamosítható!
  - Klasszikus redukciós megoldás
  - Munkacsoportonként dolgozunk fel egy-egy oszlopot
  - Minden szál elvégzi az elemi szorzást
    - Az eredményt a lokális memóriában gyűjtjük
  - Redukciós lépések
    - Minden lépésben felezzük a szálak számát
    - A még futó szálak összegzik a leállított szálak részösszegeit
    - Az utolsó szál kiírja az eredményt a globális memóriába

Mátrix vektor szorzás III.

- Feltételezések
  - N*M-es mátrix esetén
    - M szál indítható munkacsoportonként
    - N munkacsoport indítható
    - A lokális memória legalább M méretű
    - $M=2^k$ a redukcióhoz

Mátrix vektor szorzás III.

- Host program

```
#define M 32
void reduceMV(int n, float* y, const float* A, const float* x, const float* b){
  // ...
  size_t workSize = M * n;
  size_t workGroupSize = M;
  CL_SAFE_CALL( clEnqueueNDRangeKernel(commands, reduceMVKernel,
                                       1, NULL, &workSize, &workGroupSize,
                                       0, NULL, NULL) );
  // ...
}
```

Mátrix vektor szorzás III.

- OpenCL kernel

```
 #define M 32
__kernel
void reduceMV(const int n, __global float* y, __global float* A,
                           __global float* x, __global float* b){
  int i = get_group_id(0);
  int j = get_local_id(0);

  __local float Q[M];

  Q[j] = A[i * M + j] * x[j];

  for(int stride = M / 2; stride > 0; stride >>= 1){
    barrier(CLK_LOCAL_MEM_FENCE);
    if(j + stride < M){
      Q[j] += Q[j + stride];
    }
  }

  if(j == 0){
    y[i] = Q[0] + b[i];
  }
}
```

Mátrix vektor szorzás IV.

- Megoldási lehetőség
  - Az egyszerűség kedvéért csak egy munkacsoport
  - Daraboljuk a kimenetet *T* hosszú darabokra
    - A munkacsoport egyszerre egy szegmensen dolgozik
  - Daraboljuk fel a bemenetet *Z* hosszú darabokra
    - A skaláris szorzatok összegét a részösszegekből számítjuk
  - A lokális memóriában tároljuk részösszegeket
    - Q[T*Z] méretű lokális tömbben
  - Az eredmény *T* hosszú darabját redukcióval kapjuk

Mátrix vektor szorzás IV.

- Host program

```
#define T 8
#define Z 2

void largeMV(int n, int m, float* y,
             const float* A, const float* x, const float* b){

  // ...
  size_t workSize = T * Z;
  size_t workGroupSize = T * Z;
  clEnqueueNDRangeKernel(commands, largeMVKernel,
                         1, NULL, &workSize, &workGroupSize,
                         0, NULL, NULL);
  // ...
}
```

Mátrix vektor szorzás IV.

- OpenCL kernel

```
#define T 8
#define Z 2
__kernel void largeMV(const int n, const int m, __global float* y,
                      __global float* A, __global float* x, __global float* b){
  __local float Q[T * Z];

  int t = get_local_id(0) / Z;
  int z = get_local_id(0) % Z;

  for(int i = t; i < n; i += T){

  // ... ciklus mag a következő oldalon

    if(z == 0){
      y[i] = Q[t * Z + 0] + b[i];
    }
  }
}
```

Mátrix vektor szorzás IV.

- OpenCL kernel

```
// ciklus mag
Q[t * Z + z] = 0.0f;
for(int j = z; j < m; j+=Z){
  Q[t * Z + z] += A[j + i * m] * x[j];
}

for(int stride = Z / 2; stride > 0; stride >>= 1){
  barrier(CLK_LOCAL_MEM_FENCE);
  if(z + stride < Z){
   Q[t * Z + z] += Q[t * Z + z + stride];
  }
}
```

Mátrix vektor szorzás V.

- Ritka mátrixok
  - Sok nulla elem
  - Tömörítés és a tömörített reprezentáción számítás

- Compressed Sparse Row

$$\begin{bmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Value: $\begin{bmatrix} a & b & c & d & e & f \end{bmatrix}$

Column: $\begin{bmatrix} 0 & 2 & 0 & 1 & 2 & 2 \end{bmatrix}$

Row Ptr: $\begin{bmatrix} 0 & 2 & 5 \end{bmatrix}$

Mátrix vektor szorzás V.

$$\begin{bmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Value: $\begin{bmatrix} a & b & c & d & e & f \end{bmatrix}$

Column: $\begin{bmatrix} 0 & 2 & 0 & 1 & 2 & 2 \end{bmatrix}$

Row Ptr: $\begin{bmatrix} 0 & 2 & 5 \end{bmatrix}$

Value + Row Ptr: $\left[a \quad b \,\middle|\, c \quad d \quad e \,\middle|\, f \right]$

Vector + Column: $\left[x \quad z \,\middle|\, x \quad y \quad z \,\middle|\, z \right]$

Elemenkénti szorzat: $\left[ax \quad bz \,\middle|\, cx \quad dy \quad ez \,\middle|\, fz \right]$

Inclusive szegmentált scan: $\left[O_1 \,\middle|\, O_2 \,\middle|\, O_3 \right]$

Mátrix vektor szorzás V.

- Szegmentált scan
  - Feltételes scan
  - A feltétel egy külön tömbben

Inclusive scan:

$$\begin{pmatrix}1 & 2 & 3 & 4 & 5 & 6 & 7 & 8\end{pmatrix} \rightarrow \begin{pmatrix}1 & 3 & 6 & 10 & 15 & 21 & 28 & 36\end{pmatrix}$$

Head tömb

$$\begin{pmatrix}1 & 0 & 1 & 0 & 0 & 1 & 0 & 0\end{pmatrix}$$

Inclusive segmented scan:

$$\begin{pmatrix}1 & 2 \mid & 3 & 4 & 5 \mid & 6 & 7 & 8\end{pmatrix} \rightarrow \begin{pmatrix}1 & 3 \mid & 3 & 7 & 12 \mid & 6 & 13 & 21\end{pmatrix}$$

Gauss-Jordan elimináció

- Gauss elimináció
  - Visszavezetjük az egyenletrendszert háromszög mátrixra
  - Visszahelyettesítéses megoldás
- Gauss-Jordan elimináció
  - Csak a főátlóban lehet nemnulla elem

Gauss-Jordan elimináció

- Megengedett műveletek
  - Két egyenlet felcserélése
  - Egyenlet skalárral szorzása
  - Egy egyenlethez egy másik skalárszorosának hozzáadása

Gauss-Jordan eliminació

- Példa

$$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned} \xrightarrow[L_3 + L_1 \to L_3]{L_2 + \frac{3}{2}L_1 \to L_2} \begin{aligned} 2x + y - z &= 8 \\ \frac{1}{2}y + \frac{1}{2}z &= 1 \\ 2y + z &= 5 \end{aligned}$$

$$\begin{aligned} 2x + y - z &= 8 \\ \frac{1}{2}y + \frac{1}{2}z &= 1 \\ 2y + z &= 5 \end{aligned} \xrightarrow{L_3 - 4L_2 \to L_3} \begin{aligned} 2x + y - z &= 8 \\ \frac{1}{2}y + \frac{1}{2}z &= 1 \\ -z &= 1 \end{aligned}$$

$$z = -1 \longrightarrow y = 3 \longrightarrow x = 2$$

Gauss-Jordan eliminació

- Példa

$$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned} \longrightarrow \left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$$

$$\left[\begin{array}{ccc|c} 1 & \frac{1}{3} & \frac{-2}{3} & \frac{11}{3} \\ 0 & 1 & \frac{2}{5} & \frac{13}{5} \\ 0 & 0 & 1 & -1 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

Gauss-Jordan eliminació

- Mátrix inverze

$$\left[\underline{\underline{A}}\,\underline{\underline{I}}\right] \Rightarrow \underline{\underline{A}}^{-1}\left[\underline{\underline{A}}\,\underline{\underline{I}}\right] \Rightarrow \left[\underline{\underline{I}}\,\underline{\underline{A}}^{-1}\right]$$

$$\underline{\underline{A}} = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \longrightarrow \left[\underline{\underline{A}}\,\underline{\underline{I}}\right] = \left[\begin{array}{ccc|ccc} 2 & -1 & 0 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \end{array}\right]$$

$$\left[\underline{\underline{I}}\,\underline{\underline{A}}^{-1}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & 1 & 0 & \frac{1}{2} & 1 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{array}\right]$$

Gauss-Jordan eliminació

- Algoritmus

```
for k := 1 .. n-1 do
 for i := k+1 .. n do
     l := a_ik/a_kk
     b_i := b_i – l * b_k
     for j := k .. n do
        a_ij := a_ij – l * a_kj
     end for
 end for
end for
```

Gauss-Jordan eliminació

- Host program

```
void gaussian(){
 int n = 6;
 int m = 3;
 float A[] = {  2, -1,  0,  1, 0, 0,
               -1,  2, -1,  0, 1, 0,
                0, -1,  2,  0, 0, 1};

 cl_kernel gaussianKernel = createKernel(program, "gaussian");

 cl_mem AGPU = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*n*m, NULL, NULL);
 clEnqueueWriteBuffer(commands, AGPU, CL_TRUE, 0, sizeof(float)*n*m, A, 0, NULL, NULL);

 clSetKernelArg(gaussianKernel, 0, sizeof(int), &n);
 clSetKernelArg(gaussianKernel, 1, sizeof(int), &m);
 clSetKernelArg(gaussianKernel, 2, sizeof(cl_mem), &AGPU);

 clEnqueueBarrier(commands);

// ...
```

Gauss-Jordan eliminació

- Host program

```
 // ...
 size_t workSize = m;
 size_t workGroupSize = m;
 clEnqueueNDRangeKernel(commands, gaussianKernel,
                        1, NULL, &workSize, &workGroupSize,
                        0, NULL, NULL);
 clFinish(commands);

 clEnqueueReadBuffer(commands, AGPU, CL_TRUE, 0, sizeof(float)*n*m,
                     A, 0, NULL, NULL);

  clReleaseMemObject(AGPU);
  clReleaseKernel(gaussianKernel);
}
```

Gauss-Jordan eliminació

- OpenCL kernel

```
__kernel
void gaussian(const int n, const int m, __global float* A){
  int id = get_local_id(0);
  for(int ma = 0; ma < m; ++ma){
    float pp = A[ma + ma * n];
    float coeff = A[ma + id * n] / pp;
    barrier(CLK_GLOBAL_MEM_FENCE);
    if(id != ma){
      for(int na = 0; na < n; ++na){
       A[na+id*n] = A[na+id*n] - coeff * A[na+n*ma];
      }
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
  }
// ...
```

Gauss-Jordan eliminació

- OpenCL kernel

```
  // ...
  float coeff = A[id + id * n];
  for(int na = 0; na < n; ++na){
    A[na + id * n] = A[na + id * n] / coeff;
  }
}
```

Gauss-Jordan eliminácó

- Példák
  - $$\begin{bmatrix} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{bmatrix} \rightarrow$$

```
 1,  0,  0,   2
 0,  1,  0,   3
-0, -0,  1,  -1
```

  - $$\begin{bmatrix} 2 & -1 & 0 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \end{bmatrix} \rightarrow$$

```
1, 2.23e-08, 9.93e-09,  0.75, 0.5, 0.25
0,        1, 1.32e-08,   0.5,   1,  0.5
0, 2.23e-08,        1,  0.25, 0.5, 0.75
```