Elementy GPGPU

Przemysław Kiciak

Wprowadzenie

GPU (*graphics processing unit*) to „karta graficzna" wyposażona w pamięć RAM i zestaw procesorów specjalnie zaprojektowanych do wytwarzania obrazów. Natura wykonywanych przy tym obliczeń umożliwia masywną równoległość. Procesory (rdzenie obliczeniowe) GPU są prostsze niż CPU, ale ich liczba w jednym urządzeniu jest o 2–3 rzędy wielkości większa niż liczba rdzeni CPU. Dlatego moc obliczeniowa GPU też jest o ok. 2 rzędy wielkości większa.

GPGPU (*general purpose GPU programming*) jest to zastosowanie GPU do obliczeń niekoniecznie związanych z grafiką. Celem jest pełne wykorzystanie mocy obliczeniowej GPU.

Istnieje kilka „wysokopoziomowych" języków programowania GPU; najbardziej znane to GLSL i HLSL („graficzne") oraz OpenCL i CUDA („ogólnego stosowania").

Język GLSL (*OpenGL shading language*) powstał jako uzupełnienie standardu OpenGL — pisze się w nim tzw. **szadery**, czyli programy, które są wmontowywane w tzw. **potok przetwarzania grafiki** (*rendering pipeline*). Zadaniem jego kolejnych etapów jest przetworzenie danych geometrycznych i innych na piksele obrazu, zgodnie z opisem podanym w specyfikacjach standardów OpenGL lub Vulkan.

Standardy te umożliwiają wykorzystywanie także tzw. **szaderów obliczeniowych** (*compute shaders*), działających poza potokiem przetwarzania grafiki. Szadery te mogą wykonywać dowolne obliczenia, których wyniki mogą być użyte później do wykonywania obrazów lub przesłane z RAM GPU do RAM CPU.

Aplikacja OpenGL-a musi utworzyć tzw. **kontekst OpenGL-a**, a następnie tzw. **programy szaderów** i **bufory**, w których będą umieszczone dane, wyniki pośrednie i wyniki końcowe obliczeń. Program szaderów powstaje z tekstów źródłowych w GLSL-u, które należy (przy użyciu procedur obecnych w bibliotece OpenGL) skompilować i złączyć.

Treść szadera obliczeniowego opisuje działanie jednego **wątku**; wywołanie szadera za pomocą procedury `glDispatchCompute` rozpoczyna równoległe wykonywanie **grupy roboczej** wątków.

Lokalna grupa robocza jest jedno-, dwu- lub trójwymiarową tablicą wątków, której wymiary są zadeklarowane w treści szadera (i nie można ich zmienić bez ponownej kompilacji szadera).

Globalna grupa robocza jest jedno-, dwu- lub trójwymiarową tablicą lokalnych grup roboczych. Jej wymiary są określone przez parametry procedury `glDispatchCompute` (i za każdym razem mogą być inne).

Każdy wątek otrzymuje informacje o wymiarach grupy globalnej i o swoim położeniu w grupie roboczej (lokalnej i globalnej) w **zmiennych wbudowanych**, które są trójkami indeksów do wspomnianych tablic.

Podstawowym problemem w programowaniu GPU jest wymaganie tzw. **jednolitości obliczeń**. Poszczególne rdzenie GPU wykonują **wątki obliczeniowe** zorganizowane w **grupy robocze**. Wszystkie procesory działające w grupie albo wykonują jednocześnie tę samą instrukcję albo czekają — co umożliwia realizację instrukcji warunkowych, ale im więcej rdzeni czeka, tym mniej efektywnie jest wykorzystywana GPU.

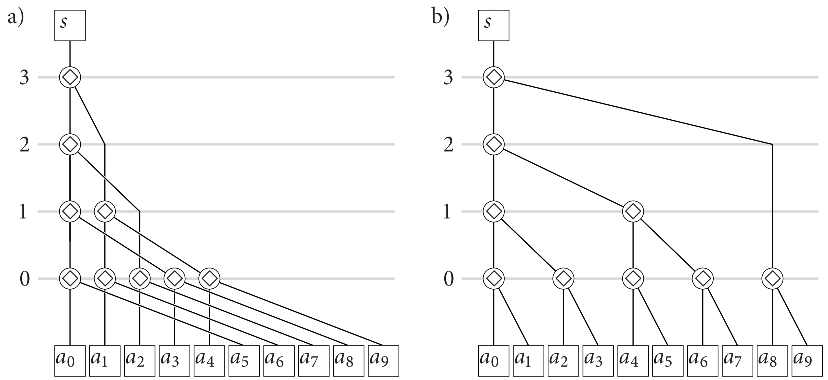
Zadaniem sterownika GPU jest przydzielanie wątków poszczególnym rdzeniom GPU, a także zapewnianie synchronizacji obliczeń.

Najważniejsze zmienne, do których szader obliczeniowy ma dostęp, to

- **zmienne jednolite** (*uniform variables*) — wartości nadaje im aplikacja działająca na CPU, wszystkie wątki obliczeniowe „widzą” w danej chwili tę samą wartość każdej takiej zmiennej,
- **zmienne w blokach magazynowych** (*shader storage blocks*), które mogą być zarówno czytane, jak i pisane przez szadery. Należy dbać o to, aby poszczególne wątki przypisywały wartości różnym zmiennym (np. różnym elementom tablicy),
- **zmienne wbudowane** (*built-in variables*), które opisują wielkości grup roboczych (tzw. lokalnej i globalnej) i numer lokalnej grupy roboczej oraz numer wątku w grupie,
- **zmienne współdzielone** (*shared variables*), za pomocą których wątki szadera mogą przekazywać sobie informacje w obrębie lokalnej grupy roboczej.

Działania parami

Działania dwuargumentowe często są iterowane, gdy trzeba obliczyć np. sumę wielu składników lub iloczyn wielu czynników. Aby takie obliczenia mogły być zrównoleglone, realizowane działanie „◇” musi być co najmniej **łączne**; *jeśli* jest także **przemienne**, to możliwe są różne algorytmy.



Obliczenie sumy lub iloczynu elementów ciągu o długości n można wykonać w $\lceil \log_2 n \rceil$ krokach. Pierwszy algorytm jest realizowany przez szader

GLSL

```
#version 450 core

#define GROUP_SIZE 64
layout (local_size_x = GROUP_SIZE) in;

uniform uint n;

void AddTwoTerms ( uint i, uint j ); /* działanie łączne i przemienne */

void main ( void )
{
  uint i, j;

  i = uint ( gl_GlobalInvocationID.x );
  if ( (j = i+(n+1)/2) < n )
    AddTwoTerms ( i, j );
} /*main*/
```

Procedura `AddTwoTerms` realizuje działanie; jeśli ma to być dodawanie liczb całkowitych umieszczonych w tablicy, to może ona wyglądać tak:

GLSL

```
layout(std430,binding=0) buffer Data { int a[]; } data;

uniform int n0;

void AddTwoTerms ( int i, int j )
{
  data.a[n0+i] += data.a[n0+j];
} /*AddTwoTerms*/
```

Jak widać, „psujemy” zawartość tablicy. Końcowy wynik, tj. suma liczb danych początkowo na pozycjach od `n0` do `n0+n-1` ma się znaleźć na pozycji `n0`.

Szader ten jest wywoływany przez procedurę

C

```
static GLuint program_id, GLuint uloc[2];
static GLint lgsize[3];

void GPUSumUp ( GLuint n, GLuint n0, GLuint databuf )
{
  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, databuf );
  glUniform1ui ( uloc[1], n0 );
  while ( n > 1 ) {
    glUniform1ui ( uloc[0], n );
    glDispatchCompute ( (n/2+lgsize[0]-1)/lgsize[0], 1, 1 );
    n = (n+1)/2;
    glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
  }
  ExitIfGLError ( "GPUSumUp" );
} /*GPUSumUp*/
```


Drugi algorytm sumowania parami (który zakłada tylko łączność działania, w związku z czym nie zmienia kolejności argumentów) jest realizowany przez szader

GLSL

```
#version 450 core

#define GROUP_SIZE 64
layout (local_size_x = GROUP_SIZE) in;

uniform uint n, q;

void AddTwoTerms ( uint i, uint j );  /* dowolne działanie łączne */

void main ( void )
{
  uint i, j;

  i = uint ( (q+q)+lg_GlobalInvocationID.x );
  if ( (j = i+q) < n )
    AddTwoTerms ( i, j );
} /*main*/
```

W zmiennej `q` jest odstęp w tablicy elementów, które mają być dodane — w kolejnych krokach to są kolejne całkowite potęgi liczby 2.

Działająca na CPU procedura sumowania realizująca drugi algorytm jest taka:

C

```
static GLuint program_id, GLuint uloc[3];
static GLint lgsize[3];

void GPUAltSumUp ( GLuint n, GLuint n0, GLuint databuf )
{
  GLuint q;

  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, databuf );
  glUniform1ui ( uloc[0], n );    /* uniform n0 = n0;*/
  glUniform1ui ( uloc[1], n0 );   /* uniform n = n; */
  for ( q = 1;  n > 1;  q += q, n = (n+1)/2 ) {
    glUniform1ui ( uloc[2], q );  /* uniform q = q; */
    glDispatchCompute ( (n/2+lgsize[0]-1)/lgsize[0], 1, 1 );
    glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
  }
} /*GPUAltSumUp*/
```

Uwagi: Z powodu „psucia” początkowej zawartości tablicy to są algorytmy o sporej złożoności pamięciowej, którą można zamienić na czas — jeśli oryginalne dane są później potrzebne, to należy albo je skopiować i zepsuć kopię, albo zepsuć oryginał, odczytać wynik obliczeń i ponownie wygenerować dane.

Dodawanie i mnożenie liczb zmiennopozycyjnych (lub macierzy o współczynnikach zmiennopozycyjnych) jest łączne z dokładnością do błędów zaokrągleń — czyli *nie jest* łączne, ale się stara. Obliczony wynik będzie sumą lub iloczynem danych zaburzonych na poziomie błędów zaokrągleń, co oznacza numeryczną poprawność.

Algorytm numerycznie poprawny jest tym lepszy im mniejsze są stałe kumulacji. W przypadku sumowania po kolei n liczb one są rzędu n, a algorytmy sumowania parami mają stałe rzędu $\log n$ — czyli są lepsze.

Wybór mniejszego lub większego elementu jest też działaniem łącznym i przemiennym, zatem można w czasie rzędu $\log n$ **znaleźć najmniejszy lub największy element ciągu**. Zobaczmy, jak oba skrajne elementy znaleźć jednocześnie.

W pierwszym kroku algorytmu użyjemy procedury zwanej **komparatorem**, która porówna dwa elementy na wskazanych miejscach i jeśli drugi jest mniejszy niż pierwszy, to je przestawi. W ten sposób w przetworzonej przez komparator parze pierwszy element jest mniejszy lub równy drugiemu.

Porządkujemy w ten sposób pary elementów sąsiednich. Dodatkowo, jeśli liczba n jest nieparzysta, to ostatni element ciągu (ten bez pary) trzeba porównać z elementami w pierwszej parze i ewentualnie zastąpić nim jeden z nich.

W kolejnych krokach działamy na parach par, znajdując w nich elementy najmniejsze i największe. W każdym kroku liczba par zawierających istotne dane maleje dwukrotnie.

Procedury pomocnicze:

GLSL

```
void CompSwap ( uint i, uint j )  /* komparator */
{
  int x;

  if ( data.a[i += n0] > data.a[j += n0] )
    { x = data.a[i];  data.a[i] = data.a[j];  data.a[j] = x; }
} /*CompSwap*/

void ChooseMin ( uint i, uint j )
{
  if ( data.a[i += n0] > data.a[j += n0] )
    data.a[i] = data.a[j];
} /*ChooseMin*/

void ChooseMax ( uint i, uint j )
{
  if ( data.a[i += n0] < data.a[j += n0] )
    data.a[i] = data.a[j];
} /*ChooseMax*/
```


Teraz procedura `main` szadera:

GLSL

```
uniform uint n;
uniform bool s;

void main ( void )
{
  uint i, j;

  i = uint ( gl_GlobalInvocationID.x );  i += i;
  if ( s ) {                          /* porządkowanie par */
    if ( (j = i+1) < n )
      CompSwap ( i, j );
    if ( i == 0 && (n & 0x01) != 0 ) {  /* n nieparzyste */
      ChooseMin ( 0, n-1 );
      ChooseMax ( 1, n-1 );
    }
  }
  else if ( (j = i + 2*((n+3)/4)) < n ) {
    ChooseMin ( i, j );
    ChooseMax ( i+1, j+1 );
  }
} /*main*/
```

Procedura na CPU wywołująca szader w kolejnych krokach:

C
```c
static GLuint program_id, uloc[3];
static GLint  lgsize[3];

void GPUFindMinMax ( GLuint n, GLuint n0, GLuint databuf )
{
  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, databuf );
  glUniform1ui ( uloc[0], n );
  glUniform1ui ( uloc[1], n0 );
  glUniform1ui ( uloc[2], true );   /* uniform s = true; */
  glDispatchCompute ( (n/2+lgsize[0]-1)/lgsize[0], 1, 1 );
  glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
  glUniform1ui ( uloc[2], false );  /* uniform s = false; */
  for ( n &= ~0x01; n > 2; n = 2*((n+3)/4) ) {
    glUniform1ui ( uloc[0], n );
    glDispatchCompute ( (n/4+lgsize[0]-1)/lgsize[0], 1, 1 );
    glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
  }
  ExitIfGLError ( "GPUFindMinMax" );
} /*GPUFindMinMax*/
```

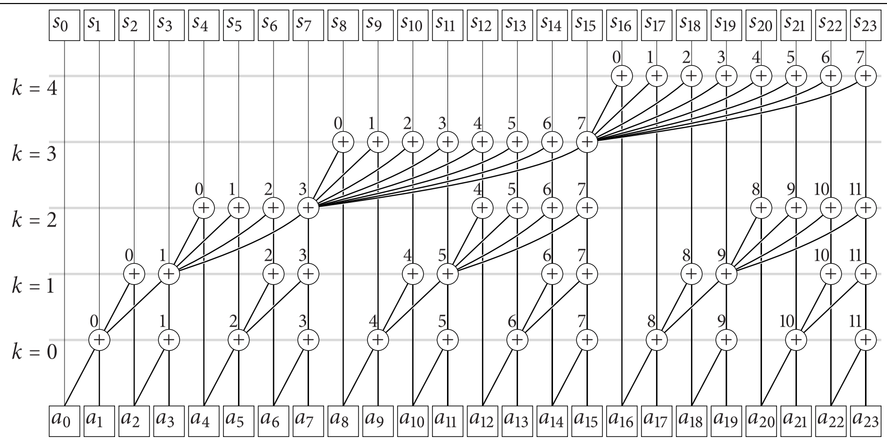

Obliczanie sum prefiksowych

Dany jest ciąg $a_0, \dots, a_{n-1}$ i działanie łączne w zbiorze, którego to są elementy — dla ustalenia uwagi dodawanie. Należy obliczyć **ciąg sum prefiksowych** $s_0, \dots, s_{n-1}$, którego elementy są zdefiniowane wzorem

$$s_i = \sum_{j=0}^{i} a_j.$$

To zadanie również można wykonać w $\lceil \log_2 n \rceil$ krokach, mając do dyspozycji $\lfloor n/2 \rfloor$ procesorów (albo raczej wątków) działających równolegle.

Ciąg dany w tablicy zostanie zastąpiony przez wynik.



W k-tym kroku kolejne wątki obliczają kolejne sumy podciągów. Wątek i-ty ma obliczyć sumę elementów o indeksach

$$i_a = 2^{k+1}\lfloor i/2^k \rfloor + 2^k - 1, \quad i_b = i_a + (i \bmod 2^k) + 1$$

i zapamiętać ją na miejscu i_b.

Procedura na GPU realizująca krok obliczania sum prefiksowych nie ma nazwy `main`, bo zazwyczaj jest ona częścią dużo większego szadera, w którym pełni rolę pomocniczą (rozwiązuje podzadanie).

GLSL
```glsl
layout (std430,binding=0) buffer Data { int a[]; } data;
uniform uint prN0, prN, prStep;

void iPrefixSum ( uint i )
{
  uint ii, m0, m1, ia, ib;

  ii = i+i;  m0 = 0x01 << prStep;  m1 = m0-1;
  ia = (ii & ~m0) | m1;
  if ( (ib = ia + (i & m1) + 1) < prN )
    data.a[prN0 + ib] += data.a[prN0 + ia];
} /*iPrefixSum*/
```

Z tego samego powodu poniższy podprogram na CPU obliczający sumy prefiksowe nie wywołuje procedury `glUseProgram`, natomiast nadaje zmiennej jednolitej `stage` wartość powodującą wywołanie procedury wykonującej krok sumowania, a potem wywołuje szader w pętli.

C

```
static void iPrefixSum ( GLuint N0, GLuint N )
{
  unsigned int k, m, d;

  glUniform1i ( uvloc[0], 0 );     /* uniform stage = 0; */
  glUniform1ui ( uvloc[1], N0 );   /* uniform prN0 = N0; */
  glUniform1ui ( uvloc[2], N );    /* uniform prN = N; */
  d = (N+2*lgsize[0]-1)/(2*lgsize[0]);
  for ( k = 0, m = N-1;  m > 0;  k++, m >>= 1 ) {
    glUniform1ui ( uvloc[3], k );  /* uniform prStep = k; */
    glDispatchCompute ( d, 1, 1 );
    glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
  }
  ExitIfGLError ( "iPrefixSum" );
} /*iPrefixSum*/
```


Sortowanie

Kolejne zadanie, które chciałoby się rozwiązać algorytmem równoległym, to **sortowanie**: dany jest ciąg n elementów zbioru, w którym jest określony porządek liniowy. Elementy należy poprzestawiać tak, aby powstał ciąg niemalejący.

Mamy do dyspozycji komparator; trzeba go wywołać dla odpowiednich par elementów tak, aby na końcu otrzymać poprawny wynik sortowania niezależnie od początkowej permutacji danych elementów. Komparatory zorganizujemy w **sieć sortującą** — w kolejnych krokach $\lfloor n/2 \rfloor$ komparatorów będzie jednocześnie porządkować $\lfloor n/2 \rfloor$ par elementów.

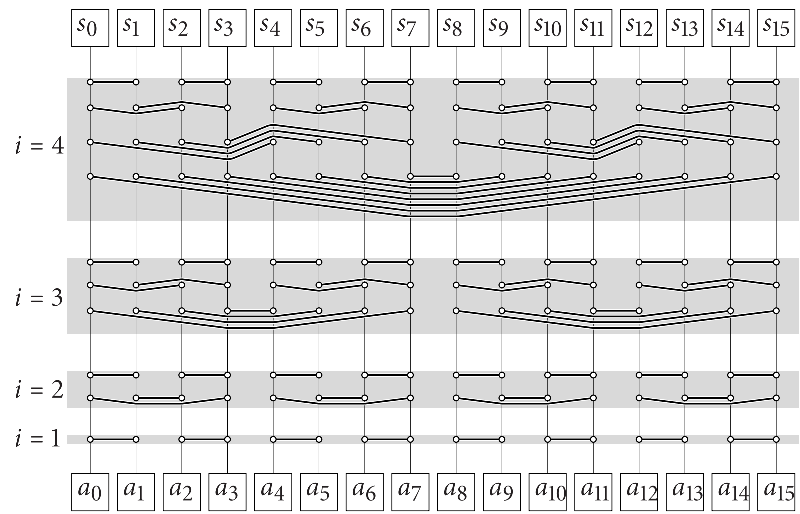
Zastosowana strategia to **sortowanie przez scalanie**. W k-tym etapie (licząc od $k = 1$) mamy posortowane kolejne podciągi o długości 2^{k-1}. Kolejne pary takich podciągów scalimy w podciągi posortowane o długości 2^k. Jeśli liczba n nie jest całkowitą potęgą dwójki, to ostatni podciąg może być krótszy. Wynik otrzymamy po $\lceil \log_2 n \rceil$ etapach scalania.

Ciąg bitoniczny jest połączeniem dwóch ciągów monotonicznych — nierosnącego i niemalejącego, o dowolnych długościach (któryś z nich może być nawet pusty).

Okazuje się (dowód w książce Cormena i in.), że jeśli podzielimy ciąg bitoniczny o parzystej długości na połowy, a potem zastosujemy komparator do par elementów ciągu bitonicznego — i-ty w pierwszej połowie z i-tym w drugiej dla każdego i, to połowy otrzymanego ciągu będą ciągami bitonicznymi, przy czym wszystkie elementy pierwszej połowy będą mniejsze lub równe elementom drugiej połowy.

Zatem, mając ciąg bitoniczny o długości 2^k, dla $k > 0$, możemy go zamienić na dwa ciągi bitoniczne o długości 2^{k-1} i dalej, indukcyjnie, na 2^k ciągów bitonicznych o długości 1 — to razem da ciąg posortowany. Takie sortowanie ciągu bitonicznego, wykonalne w k krokach, nazywa się **czyszczeniem**.

Jeśli teraz mamy scalić dwa sąsiednie ciągi posortowane o długości 2^{k-1}, to zauważamy, że po odwróceniu kolejności elementów drugiego ciągu powstaje ciąg bitoniczny o długości 2^k. Wystarczy go tylko wyczyścić.



Sortowanie ciągu o długości n składa się z $\lceil \log_2 n \rceil$ etapów, k-ty etap składa się z k kroków, a więc całość wymaga wykonania $\frac{1}{2}\lceil \log_2 n \rceil \big(\lceil \log_2 n \rceil + 1 \big)$ kroków.

Procedura `main` szadera sieci sortującej:

GLSL

```
#version 450 core

#define GROUP_SIZE 64
layout (local_size_x = GROUP_SIZE) in;

uniform uint n, h;
uniform bool reverse;

void CompSwap ( uint i, uint j );

void main ( void )
{
  uint iid, i, j, l, h2;

  iid = uint ( gl_GlobalInvocationID.x );
  h2 = h >> 1;  l = iid % h2;  iid /= h2;
  i = iid*h+l;  j = reverse ? (iid+1)*h-l-1 : i+h2;
  if ( j < n )
    CompSwap ( i, j );
} /*main*/
```


Procedura sortowania w C niestety na jednym slajdzie się nie mieści.

C

```
void GPUNetSort ( GLuint n, GLuint n0, GLuint dbuf )
{
  GLuint steps, nn, h, h2, h4, k, kk, gsize, i;

  if ( n < 2 )
    return;
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, dbuf );
  glUseProgram ( program_id );
  for ( nn = n-1, steps = 0;  nn;  nn >>= 1, steps ++ )
    ;
  nn = 1 << steps;
  glUniform1ui ( uloc[0], n );
  glUniform1ui ( uloc[1], n0 );
  glUniform1ui ( uloc[2], false );    /* uniform reverse = false; */
  glUniform1ui ( uloc[3], 2 );        /* uniform h = 2; */
  glDispatchCompute ( gsize = (nn/2+lgsize[0]-1)/lgsize[0], 1, 1 );
  glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
```



```
  for ( i = 1, h2 = 2, h = 4, k = nn >> 2;
        i < steps;
        i++, h2 = h, h <<= 1, k >>= 1 ) {
    glUniform1ui ( uloc[2], true );   /* uniform reverse = true; */
    glUniform1ui ( uloc[3], h );
    glDispatchCompute ( gsize, 1, 1 );
    glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
    glUniform1ui ( uloc[2], false );  /* uniform reverse = false; */
    for ( h2 = h >> 1, h4 = h2 >> 1, kk = k << 1;
          h2 > 1;
          h2 = h4, kk <<= 1, h4 >>= 1 ) {
      glUniform1ui ( uloc[3], h2 );   /* uniform h = h2; */
      glDispatchCompute ( gsize, 1, 1 );
      glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
    }
  }
  ExitIfGLError ( "GPUNetSort" );
} /*GPUNetSort*/
```


Mnożenie macierzy rzadkiej przez wektor

Macierz rzadka $m \times n$ jest to (duża) macierz, której większość współczynników jest równa 0. Liczba niezerowych współczynników może być rzędu $O(m+n)$ lub może to być kilka lub kilkanaście procent liczby mn.

Tu zajmuję się **rzadkimi macierzami nieregularnymi**, których niezerowe współczynniki są rozmieszczone w dowolnych miejscach. Użyję oszczędnej reprezentacji takiej macierzy w postaci wykazu miejsc, w których są niezerowe współczynniki.

Mając daną macierz rzadką A i wektor $\boldsymbol{x} \in \mathbb{R}^n$, chcę obliczyć wektor $\boldsymbol{y} = A\boldsymbol{x}$. Wiersze i kolumny numeruję od 0. Prowadząc obliczenia według wzoru

$$y_i = \sum_{i=0}^{n-1} a_{ij} x_j,$$

GPU obliczy i zsumuje tylko te składniki, w których $a_{ij} \neq 0$.

Reprezentacja macierzy jest umieszczona w trzech tablicach: tablica `r` o długości $m+1$ zawiera liczby całkowite $r_0, \ldots, r_m$, przy czym $r_0 = 0$, a dla $i = 0, \ldots, m-1$ różnica $r_{i+1} - r_i$ jest liczbą niezerowych współczynników w wierszu i-tym.
Liczba $N = r_m$ jest liczbą wszystkich niezerowych współczynników macierzy A.

W tablicy `c` jest N liczb całkowitych, $c_0, \ldots, c_{N-1}$. Liczby $c_{r_i}, c_{r_i+1}, \ldots, c_{r_{i+1}-1}$ są numerami tych kolumn, w których i-ty wiersz zawiera niezerowe współczynniki.

Same współczynniki (liczby rzeczywiste) są podane w tablicy `a` o długości N.
Na pozycjach $r_i, r_i+1, \ldots, r_{i+1}-1$ są przechowywane niezerowe współczynniki z i-tego wiersza: $a_{i,c_{r_i}}, a_{i,c_{r_i+1}}, \ldots, a_{i,c_{r_{i+1}-1}}$.

W pamięci GPU tablice `r` i `c` umieszczę jedna za drugą w jednym bloku magazynowym, a tablicę `a` w drugim bloku.

Szader obliczeniowy ma też dostęp do bloku z tablicą `x`, w której są podane współrzędne wektora $\boldsymbol{x}$, i do bloku z tablicą `y`, do której ma być wpisany wynik.

Oprócz tego będą potrzebne dwie tablice robocze. W pierwszej z nich, o długości m, będą liczby niezerowych współczynników w kolejnych wierszach, tj. liczby składników do zsumowania.

W drugiej tablicy zmieści się N liczb rzeczywistych. Wątki szadera *jednocześnie* obliczą i zapamiętają w niej *wszystkie* iloczyny $a_{ij}x_j$, a potem będą obliczały sumy — wszystkie równolegle i stosując algorytm sumowania parami.

Początek szadera obliczeniowego jest taki:

GLSL

```
#version 450 core

#define GROUP_SIZE 64
layout (local_size_x = GROUP_SIZE) in;

layout (std430,binding = 0) buffer RowsCols { uint  rc[]; } rc;
layout (std430,binding = 1) buffer Coeff    { float a[]; } a;
layout (std430,binding = 2) buffer Xvec     { float x[]; } x;
layout (std430,binding = 3) buffer Yvec     { float y[]; } y;
layout (std430,binding = 4) buffer RowL     { uint  l[]; } lgt;
layout (std430,binding = 5) buffer Prod     { float b[]; } b;

uniform uint m, nnz, t;
uniform int  stage;

#define r(I) rc.rc[I]
#define c(I) rc.rc[m+1+(I)]
```


Makra `r` i `c` zapewniają dostęp do tablic `r` i `c`, parametr `I` jest indeksem.

Zmienne `m` i `nnz` mają wartości m i N.
Wartość zmiennej `t` określa liczbę składników w etapie sumowania iloczynów $a_{ij}x_j$.

Zmienna jednolita `stage` służy do wyboru etapu obliczeń. Szader obliczeniowy będzie wywoływany za pośrednictwem makra `EXECSTEP` lub `EXECSTAGE` — to ostatnie nadaje wartość zmiennej `stage`).

C

```
static GLuint program_id, uvloc[4]
static GLint  lgsize[3];

#define EXECSTEP(SIZE) \
  glDispatchCompute ( SIZE, 1, 1 ); \
  glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
#define EXECSTAGE(STAGE,SIZE) \
  { glUniform1i ( uvloc[0], STAGE ); EXECSTEP ( SIZE ) }
```

Procedura `main` szadera jest jedną wielką instrukcją przełącznika — wybierającego obliczenia dla kolejnych etapów.

GLSL

```
void main ( void )
{
  uint iid, j, k, p;

  iid = gl_GlobalInvocationID.x;
  switch ( stage ) {
```

Początkowe instrukcje procedury w C to przygotowanie — nadanie wartości zmiennym jednolitym `m` i `nnz`, przywiązanie buforów do punktów dowiązania bloków magazynowych i utworzenie buforów roboczych.

C

```
void GPUMultSparseMatrixVectorf ( GLuint m, GLuint n, GLuint nnz,
                               GLuint *lmax, GLuint rowscols, GLuint coeff,
                               GLuint x, GLuint y )
{
  GLuint auxb[2], gsize, t;

  glUseProgram ( program_id );
  glUniform1ui ( uvloc[1], m );
  glUniform1ui ( uvloc[2], nnz );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, rowscols );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, coeff );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, x );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, y );
  glGenBuffers ( 2, auxb );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, auxb[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, m*sizeof(GLuint),
                 NULL, GL_DYNAMIC_DRAW );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 5, auxb[1] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, nnz*sizeof(GLfloat),
                 NULL, GL_DYNAMIC_DRAW );
```


Etap `0` — znalezienie liczb niezerowych współczynników w poszczególnych wierszach:

C

```
  EXECSTAGE ( 0, gsize = (m+lgsize[0]-1)/lgsize[0] )
```

GLSL

```
case 0:
    if ( iid < m )
      lgt.l[iid] = r(iid+1) - r(iid);
    return;
```


Etap `1` — znalezienie największej liczby niezerowych współczynników w wierszu, wykonywany tylko jeśli zmienna `*lmax` ma wartość 0, w przeciwnym razie zakłada się, że ta liczba jest wartością zmiennej `*lmax`. Ponieważ to obliczenie psuje zawartość tablicy, etap `0` jest powtarzany.

C

```
  if ( !*lmax ) {
    glUniform1i ( uvloc[0], 1 );  /* stage = 1 */
    for ( t = m; t > 1; t = (t+1)/2 ) {
      glUniform1ui ( uvloc[3], t );
      EXECSTEP ( (t/2+lgsize0[0]-1)/lgsize0[0] )
    }
    glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, sizeof(GLuint),
                         lmax );
    EXECSTAGE ( 0, gsize )
  }
```

GLSL

```
case 1:
    if ( (j = iid + (t+1)/2) < t )
      if ( lgt.l[j] > lgt.l[iid] )
        lgt.l[iid] = lgt.l[j];
    return;
```

Etap 2 — obliczanie iloczynów $a_{ij}x_j$:

C
```
32:   EXECSTAGE ( 2, (nnz+lgsize0[0]-1)/lgsize0[0] )
```
GLSL
```
16: case 2:
17:     if ( iid < nnz )
18:       b.b[iid] = a.a[iid] * x.x[c(iid)];
19:     return;
```


Etap 3 — sumowanie iloczynów parami. Wartość zmiennej `t` jest maksymalną liczbą składników, ale poszczególne sumy mogą mieć ich mniej. Dlatego dodawanie wykonuje się tylko dla takich par, w których drugi składnik ma numer mniejszy niż `t` i mniejszy niż $r_{i+1} - r_i$ (po wykonaniu instrukcji w linii 21 zmienna `iid` ma wartość i).

C
```
33:   glUniform1i ( uvloc[0], 3 );  /* stage = 3 */
34:   for ( t = *lmax; t > 1; t = (t+1)/2 ) {
35:     glUniform1ui ( uvloc[3], t );
36:     EXECSTEP ( (m*(t/2)+lgsize0[0]-1)/lgsize0[0] )
37:   }
```
GLSL
```
20: case 3:
21:     p = t/2;  j = iid % p;  iid /= p;
22:     if ( iid < m ) {
23:       k = j + t - p;
24:       if ( k < lgt.l[iid] && k < t )
25:         b.b[r(iid)+j] += b.b[r(iid)+k];
26:     }
27:     return;
```


Etap 4 — kopiowanie obliczonych współrzędnych wektora $\boldsymbol{y}$ z tablicy roboczej do tablicy `y`. Jeśli i-ty wiersz macierzy A jest zerowy, to następuje przypisanie $y_i = 0$.

C
```
38:   EXECSTAGE ( 4, gsize )
39:   glDeleteBuffers ( 2, auxb );
40:   ExitIfGLError ( "GPUMultSparseMatrixVectorf" );
41: } /*GPUMultSparseMatrixVectorf*/
```
GLSL
```
28: case 4:
29:     if ( iid < m )
30:       y.y[iid] = lgt.l[iid] > 0 ? b.b[r(iid)] : 0.0;
31:     return;
32: default:
33:     return;
34:   }
35: } /*main*/
```

Na końcu procedura w C sprząta, tj. likwiduje bufory robocze.

Uwaga: Jeśli macierz jest symetryczna, to można ją reprezentować w jeszcze mniejszej ilości pamięci, przechowując tylko dolny (albo górny) trójkąt. Niestety, taka oszczędna reprezentacja znacznie skomplikowałaby i spowolniła algorytm mnożenia — iloczyny współczynników a_{ij} z górnego trójkąta z liczbami x_j trzeba by sortować względem i przed sumowaniem.

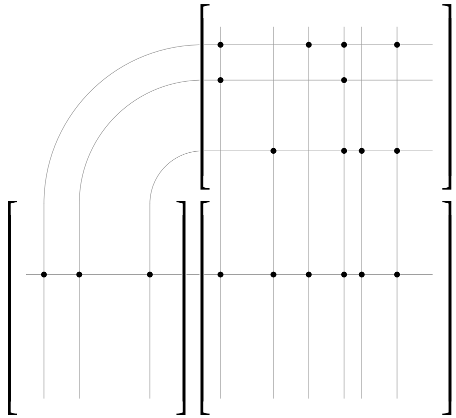
Mnożenie macierzy rzadkich

Spodziewając się, że iloczyn macierzy rzadkich, A i B o wymiarach $m \times n$ i $n \times l$, też jest macierzą rzadką, chcemy obliczyć ten iloczyn możliwie małym kosztem, przy czym dla wszystkich macierzy użyjemy reprezentacji opisanej wcześniej. Trzeba znaleźć miejsca, w których macierz $C = AB$ ma niezerowe współczynniki.

i-ty wiersz macierzy C jest sumą wierszy macierzy B pomnożonych przez kolejne współczynniki z i-tego wiersza macierzy A; można pomnożyć i zsumować tylko te wiersze, które są mnożone przez niezerowe współczynniki. Dalej, niezerowe współczynniki w i-tym wierszu macierzy C mogą być tylko w tych kolumnach, w których wybrane wiersze macierzy B mają niezerowe współczynniki.

$$\begin{array}{ccc} & & B \\ & & \downarrow \\ A & \rightarrow & C \end{array}$$

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$



Oczywiście, współczynnik c_{ij}, który jest sumą niezerowych iloczynów, może być zerem, ale to zignoruję. W razie potrzeby można zrobić dodatkowy etap obliczeń, który „wyczyści” otrzymaną macierz C ze współczynników zerowych lub mających wartość bezwzględną na poziomie błędów zaokrągleń.

Implementacja algorytmu mnożenia macierzy rzadkich składa się z procedury działającej na CPU i wywoływanego przez nią programu szaderów, zawierającego szader obliczeniowy. W tej implementacji przydadzą się *wszystkie* algorytmy przedstawione wcześniej, do rozwiązania podzadań na kolejnych etapach.

Początek szadera — punkty dowiązania buforów:

GLSL
```
#version 450 core

#define GROUP_SIZE 64

layout (local_size_x = GROUP_SIZE) in;

layout (std430,binding = 0) buffer RCA    { uint  rc[]; } rca;
layout (std430,binding = 1) buffer CoeffA { float a[]; }  aa;
layout (std430,binding = 2) buffer RCB    { uint  rc[]; } rcb;
layout (std430,binding = 3) buffer CoeffB { float a[]; }  ab;
layout (std430,binding = 4) buffer Auxb0  { uint  a[]; }  aux0;
layout (std430,binding = 5) buffer Auxb1  { float a[]; }  aux1;
layout (std430,binding = 6) buffer Auxb2  { uint  a[]; }  aux2;

```


Każda z macierzy A i B jest podana w dwóch blokach, przy czym dostęp do tablic `r` i `c` jest realizowany za pomocą odpowiednich makr. Są też trzy bloki robocze o nazwach `aux0`, `aux1` i `aux2` zawierające tablice, do których dostęp też jest realizowany przez makra objaśnione dalej.

GLSL
```
uniform int  stage;
uniform uint prN0, prN, prStep;
uniform uint ma, nnza, mb, nprod, nnzc, h, tablgt;
uniform bool reverse;

#define ra(I) rca.rc[I]
#define ca(I) rca.rc[ma+1+(I)]
#define rb(I) rcb.rc[I]
#define cb(I) rcb.rc[mb+1+(I)]
#define pairi(I) aux2.a[2*(I)]
#define pairj(I) aux2.a[2*(I)+1]
#define tab1(I) aux0.a[I]
#define tab2(I) aux0.a[tablgt+(I)]
#define tab3(I) aux0.a[tablgt+tablgt+(I)]

```

Procedura `main` szadera jest przełącznikiem wybierającym instrukcje do wykonania na etapie określonym przez wartość zmiennej jednolitej `stage`. Jeśli jest to `0`, to należy obliczyć ciąg sum prefiksowych w tablicy `aux0.a`, od miejsca `n0` do `n0+n-1`.

GLSL

```
void iPrefixSum ( uint i )  { .... } /* procedura opisana wcześniej */

void main ( void )
{
  uint i, j, k, l, m, p, q;

  i = gl_GlobalInvocationID.x;
  switch ( stage ) {
case 0:
    iPrefixSum ( i );
    return;
```


Procedura w C posługuje się takimi samymi makrami `EXECSTEP` i `EXECSTAGE` jak procedura mnożenia macierzy przez wektor. Parametry wejściowe opisują macierze A i B, a parametry wyjściowe są wskaźnikami zmiennych, którym będą przypisane liczba niezerowych współczynników oraz identyfikatory buforów (zarezerwowanych przez tę procedurę) z reprezentacją macierzy C.

C

```
static GLuint shader_id, program_id;
static GLuint uvloc[12];
static GLint  lgsize[3];

#define EXECSTEP(SIZE) \
  glDispatchCompute ( SIZE, 1, 1 ); \
  glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT );
#define EXECSTAGE(STAGE,SIZE) \
  { glUniform1i ( uvloc[0], STAGE ); EXECSTEP ( SIZE ) }

char GPUMultSparseMatricesf ( GLuint m, GLuint n, GLuint l,
                              GLuint nnza, GLuint rca, GLuint ca,
                              GLuint nnzb, GLuint rcb, GLuint cb,
                              GLuint *nnzc, GLuint *rcc, GLuint *cc )
{
```


Bufory z danymi macierzami są przywiązywane, zmienne jednolite `m`, `nnza` i `n` mają przypisywane wartości, a następnie procedura rezerwuje 5 buforów — dwa z nich pomieszczą wynik, a pozostałe trzy będą robocze. W pierwszym z nich ma być przechowywane $N_A + 1$ liczb całkowitych.

C

```
  GLuint auxb[5], nprod, _nnzc, maxnt, tablgt, i;

  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, rca );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, ca );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, rcb );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, cb );
  glUniform1ui ( uloc[4], m );
  glUniform1ui ( uloc[5], nnza );
  glUniform1ui ( uloc[6], n );
  glGenBuffers ( 5, auxb );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, auxb[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER,
                (nnza+1)*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
```


W etapie `1` jest obliczana liczba P iloczynów niezerowych współczynników — dla każdego współczynnika a_{ij} bierzemy liczbę niezerowych współczynników w j-tym wierszu macierzy B i obliczamy sumę tych liczb, następnie odczytujemy ją z bufora. Jeśli suma jest zerem, to koniec obliczeń — macierz C jest zerowa.

C

```
  EXECSTAGE ( 1, (nnza+lgsize[0]-1)/lgsize[0] )
  iPrefixSum ( 1, nnza );
  glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
                       nnza*sizeof(GLuint), sizeof(GLuint), &nprod );
  if ( !nprod ) {
    glDeleteBuffers ( 5, auxb );
    *nnzc = *rcc = *cc = 0;
    return false;
  }
```

GLSL

```
case 1:
    if ( i == 0 ) aux0.a[i] = 0;
    if ( i < nnza ) {
      j = ca(i);
      aux0.a[i+1] = rb(j+1)-rb(j);
    }
    return;
```

Rezerwujemy pozostałe dwa bufory robocze — pierwszy o pojemności $2P$ liczb całkowitych, a drugi przeznaczony na P liczb zmiennopozycyjnych. Dla każdego iloczynu $a_{ik}b_{kj}$ trzeba zapamiętać trójkę $(i, j, a_{ik}b_{kj})$ — dostęp do elementów i, j trójki zapewniają makra `pairi` i `pairj`.

C

```
glUniform1ui ( uloc[7], nprod );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 6, auxb[1] );
glBufferData ( GL_SHADER_STORAGE_BUFFER,
               2*nprod*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 5, auxb[2] );
glBufferData ( GL_SHADER_STORAGE_BUFFER,
               nprod*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
```

W etapie 2 obliczane są iloczyny i zapamiętywane trójki, które je opisują, ale wcześniej szader musi znaleźć, dla każdego iloczynu, liczby i, j oraz wybrać z tablic współczynniki do pomnożenia.

Robi się to metodą binarnego wyszukiwania — zmienna `p` otrzymuje indeks współczynnika a_{ik}, a zmienna `q` indeks współczynnika b_{kj} w odpowiedniej tablicy.

C

```
EXECSTAGE ( 2, (nprod+lgsize[0]-1)/lgsize[0] )
```

GLSL

```
case 2:
  if ( i < nprod ) {
    for ( p = 0, k = nnza;  k-p > 1; ) {
      l = p + (k-p)/2;
      if ( i >= aux0.a[l] ) p = l; else k = l;
    }
    for ( j = 0, k = ma;  k-j > 1; ) {
      l = j + (k-j)/2;
      if ( p >= ra(l) ) j = l; else k = l;
    }
    pairi(i) = j;
    q = rb(ca(p))+i-aux0.a[p];
    pairj(i) = cb(q);
    aux1.a[i] = aa.a[p]*ab.a[q];
  }
  return;
```


Następuje realokacja bufora na blok `aux0`; musi on teraz pomieścić trzy tablice o długościach T, T i N_C, gdzie $T = \max(P, m) + 1$. Odtąd dozwolone jest użycie makr `tab1`, `tab2` i `tab3` — używana w nich zmienna `tablgt` otrzymuje wartość T. Liczba N_C niezerowych współczynników macierzy C nie jest jeszcze znana, ale jest $N_C \leqslant P$.

C

```
glUniform1ui ( uloc[11], tablgt = nprod > m ? nprod+1 : m+1 );
glDeleteBuffers ( 1, &auxb[0] );
glGenBuffers ( 1, &auxb[0] );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, auxb[0] );
glBufferData ( GL_SHADER_STORAGE_BUFFER,
               (3*tablgt-1)*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
```


Kolejne trzy etapy mają na celu posortowanie trójek tak, aby iloczyny do zsumowania znalazły się obok siebie. Trójki są już posortowane w kolejności niemalejących indeksów i — trzeba posortować niemalejąco względem j podciągi trójek o tych samych indeksach i.

W etapie 3, za pomocą binarnego wyszukiwania znajdowane są początki podciągów do posortowania.

C

```
EXECSTAGE ( 3, (m+lgsize[0]-1)/lgsize[0] )
```

GLSL

```
case 3:
  if ( i == 0 ) { tab1(0) = 0;  tab1(ma) = nprod; }
  else if ( i < ma ) {
    for ( j = 0, k = nprod;  k-j > 1; ) {
      l = j + (k-j)/2;
      if ( pairi(l) < i ) j = l; else k = l;
    }
    tab1(i) = k;
  }
  return;
```

Etap 4 oblicza długości tych podciągów, a w etapie 5 jest znajdowana największa długość (odczytywana z początku tablicy `tab2` do zmiennej `maxnt`).

C

```
EXECSTAGE ( 4, (m+lgsize[0]-1)/lgsize[0] );
glUniform1i ( uloc[0], 5 );  /* uniform stage = 5; */
for ( i = m; i > 1; i = (i+1)/2 ) {
  glUniform1ui ( uloc[2], i );
  EXECSTEP ( (i/2+lgsize[0]-1)/lgsize[0] );
}
glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
                     tablgt*sizeof(GLuint), sizeof(GLuint), &maxnt );
```

GLSL

```
case 4:
    if ( i < ma )
      tab2(i) = tab1(i+1)-tab1(i);
    return;
case 5:
    if ( (j = i+(prN+1)/2) < prN ) {
      if ( tab2(i) < tab2(j) )
        tab2(i) = tab2(j);
    }
    return;
```


Kolejny etap, 6, to sortowanie trójek — nie zamieszczam procedury `NetSort`, która realizuje algorytm opisany wcześniej. Ta implementacja sortuje wszystkie ciągi jednocześnie. Na podstawie numeru wątku w globalnej grupie roboczej szader ustala numer `ns` podciągu, który sortuje i numer `np` pary w tym podciągu, którą ma uporządkować. Procedura `SortIt` oblicza indeksy elementów pary i wykonuje zadanie komparatora:

GLSL

```
void SortIt ( uint ns, uint np )
{
  uint  i, j, l, h2;
  float x;

  h2 = h >> 1;  l = np % h2;  np /= h2;
  i = tab1(ns)+np*h+l;  j = reverse ? tab1(ns)+(np+1)*h-l-1 : i + h2;
  if ( j < tab1(ns+1) ) {
    if ( pairj(i) > pairj(j) ) {
      l = pairj(i);  pairj(i) = pairj(j);  pairj(j) = l;
      x = aux1.a[i];  aux1.a[i] = aux1.a[j];  aux1.a[j] = x;
    }
  }
} /*SortIt*/
```


W etapie 7 do tablicy `tab1` są wpisywane jedynki — dla każdego pierwszego iloczynu w podciągu, który należy zsumować i zera dla pozostałych. Liczba tych jedynek jest liczbą N_C niezerowych współczynników macierzy C.

C

```
NetSort ( m, maxnt );
EXECSTAGE ( 7, (nprod+lgsize[0]-1)/lgsize[0] )
iPrefixSum ( 1, nprod );
glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
                     nprod*sizeof(GLuint), sizeof(GLuint), &_nnzc );
```

GLSL

```
case 6:
    if ( (j = i / prN0) < ma )
      SortIt ( j, i % prN0 );
    return;
case 7:
    if ( i == 0 )
      { tab1(0) = 0;  tab1(1) = 1; }
    else if ( i < nprod )
      tab1(i+1) = pairi(i-1) != pairi(i) || pairj(i-1) != pairj(i) ?
                  1 : 0;
    return;
```


Znając liczbę N_C, można zarezerwować bufory, w których zostanie umieszczona macierz C. Bufory są przywiązywane do punktów wcześniej używanych do dostępu do macierzy A.

C

```
glUniform1ui ( uloc[8], _nnzc );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, auxb[4] );
glBufferData ( GL_SHADER_STORAGE_BUFFER,
               _nnzc*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, auxb[3] );
glBufferData ( GL_SHADER_STORAGE_BUFFER,
               (m+1+_nnzc)*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
```

W etapie 8 do tablicy `c` są wpisywane numery kolumn, w których są niezerowe współczynniki macierzy *C*. Do tablicy `tab2` jest wpisywany ciąg liczb — indeksów początków podciągów iloczynów do zsumowania. Różnice obliczane w etapie 9 są długościami podciągów.

C

```c
EXECSTAGE ( 8, (nprod+lgsize[0])/lgsize[0] )
EXECSTAGE ( 9, (_nnzc+lgsize[0]-1)/lgsize[0] )
```

GLSL

```glsl
case 8:
    if ( i <= nprod ) {
      if ( i == nprod )
        tab2(nnzc) = nprod;
      else if ( tab1(i+1) > tab1(i) ) {
        ca(tab1(i)) = pairj(i);
        tab2(tab1(i)) = i;
      }
    }
    return;
case 9:
    if ( i < nnzc )
      tab3(i) = tab2(i+1) - tab2(i);
    return;
```


W etapie 10 znajdowana jest największa liczba składników do zsumowania. Potem jest odczytywana i etap 9 jest powtarzany, aby znów w tablicy `tab3` były te liczby składników.

C

```c
glUniform1i ( uloc[0], 10 );  /* uniform stage = 10; */
for ( i = _nnzc; i > 1; i = (i+1)/2 ) {
  glUniform1ui ( uloc[2], i );  /* uniform prN = i; */
  EXECSTEP ( (i/2+lgsize[0]-1)/lgsize[0] )
}
glBindBuffer ( GL_SHADER_STORAGE_BUFFER, auxb[0] );
glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
             (tablgt+tablgt)*sizeof(GLuint), sizeof(GLuint), &maxnt );
EXECSTAGE ( 9, (_nnzc+lgsize[0]-1)/lgsize[0] )
```

GLSL

```glsl
case 10:
    if ( (j = i+(prN+1)/2) < prN ) {
      if ( tab3(i) < tab3(j) )
        tab3(i) = tab3(j);
    }
    return;
```


Etap 11 to sumowanie parami składników.

C

```c
glUniform1i ( uloc[0], 11 );  /* uniform stage = 11; */
for ( i = maxnt; i > 1; i = (i+1)/2 ) {
  glUniform1ui ( uloc[2], i );  /* uniform prN = i; */
  EXECSTEP ( (_nnzc*(i/2)+lgsize[0]-1)/lgsize[0] )
}
```

GLSL

```glsl
case 11:
    p = prN/2;  j = i % p;  i /= p;
    if ( i < nnzc ) {
      k = j + prN - p;
      if ( k < tab3(i) && k < prN )
        aux1.a[tab2(i)+j] += aux1.a[tab2(i)+k];
    }
    return;
```


Etap 12 ma na celu przepisanie obliczonych sum, tj. współczynników macierzy *C*, na docelowe miejsca w buforze. Do tablicy pomocniczej `tab1` są wpisywane numery wierszy, w których są te współczynniki.

C

```c
EXECSTAGE ( 12, (_nnzc+lgsize[0]-1)/lgsize[0] )
```

GLSL

```glsl
case 12:
    if ( i < nnzc ) {
      aa.a[i] = aux1.a[tab2(i)];
      tab1(i) = pairi(tab2(i));
    }
    return;
```

W etapie `13` wypełniana jest tablica `r` reprezentacji macierzy C, numerami miejsc, od których w tablicach `c` i `a` zaczynają się opisy kolejnych wierszy tej macierzy. Tu znów konieczne okazało się wyszukiwanie binarne odpowiednich miejsc w tablicy `tab1`.

C

```
  EXECSTAGE ( 13, (m+lgsize[0]-1)/lgsize[0] )
```

GLSL

```
case 13:
    if ( i == 0 ) { ra(0) = 0;  ra(ma) = nnzc; }
    else if ( i < ma ) {
      for ( p = 0, k = nnzc;  k-p > 1; ) {
        l = p + (k-p)/2;
        if ( i > tab1(l-1) ) p = l; else k = l;
      }
      ra(i) = p;
    }
    return;
default:
    return;
  }
} /*main*/
```


Na końcu procedura w C musi już tylko przypisać odpowiednie wartości (liczbę N_C i identyfikatory buforów) zmiennym wskazywanym przez parametry i posprzątać (tj. zlikwidować bufory z tablicami roboczymi).

C

```
  *nnzc = _nnzc;
  *rcc = auxb[3];
  *cc = auxb[4];
  glUseProgram ( 0 );
  glDeleteBuffers ( 3, auxb );
  ExitIfGLError ( "GPUMultSparseMatricesf" );
  return true;
} /*GPUMultSparseMatricesf*/
```


Dziękuję