

Paralelní algoritmy v lineární algebře

Tomáš Oberhuber

`tomas.oberhuber@fjfi.cvut.cz`

22. dubna 2024

Video na Youtube

Násobení hustých matic

- ▶ Násobení husté matice s vektorem je poměrně přímočaré:
 - ▶ do vstupního vektoru přistupujeme opakovaně a může se vyplatit ho držet v keši CPU nebo sdílené paměti multiprocesrou na GPU
- ▶ Násobení dvou hustých matic na CPU jsme si ukázali již v části a optimálních přístupech do paměti na CPU.
- ▶ Podobný postup využívající násobení jednotlivých dlaždic matice se hodí i na GPU.

Ukážeme si násobení hustých matic na architekturách s distribuovanou pamětí.

Násobení hustých matic

- ▶ mějme matice $A, B, C \in \mathbb{R}^{n,n}$
- ▶ počítáme součin $C = AB$
- ▶ mějme p procesů a necht' p je mocnina dvou
- ▶ matice rozdělíme blokově na $\sqrt{p} \times \sqrt{p}$ bloků
- ▶ pak je

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} B_{k,j}$$

- ▶ výpočet bloku $C_{i,j}$ provádí ten proces, na který je blok mapován
- ▶ od ostatních procesů si musí vyžádat příslušné bloky $A_{i,k}$ a $B_{k,j}$

Násobení hustých matic

Příklad: pro $p = 9$

$$C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0}$$

$$C_{1,0} = A_{1,0}B_{0,0} + A_{1,1}B_{1,0} + A_{1,2}B_{2,0}$$

$$C_{2,0} = A_{2,0}B_{0,0} + A_{2,1}B_{1,0} + A_{2,2}B_{2,0}$$

$$C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1}$$

$$C_{1,1} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{2,1} = A_{2,0}B_{0,1} + A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{0,2} = A_{0,0}B_{0,2} + A_{0,1}B_{1,2} + A_{0,2}B_{2,2}$$

$$C_{1,2} = A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,2} = A_{2,0}B_{0,2} + A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

- vidíme, že např. v prvním kroku tři různé procesy potřebují současně bloky $A_{0,0}$, $A_{1,0}$, $A_{2,0}$ a podobně pro bloky z matice \mathbb{B}

Násobení hustých matic

- ▶ tyto konflikty jsou nevýhodné zejména pro architektury s distribuovanou pamětí
- ▶ změníme pořadí sčítání bloků v jednotlivých procesech
- ▶ každý řádek pro výpočet bloku $C_{i,j}$ otáčejeme $(i + j)$ -krát doleva

$$C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0}$$

$$C_{1,0} = A_{1,1}B_{1,0} + A_{1,2}B_{2,0} + A_{1,0}B_{0,0}$$

$$C_{2,0} = A_{2,2}B_{2,0} + A_{2,0}B_{0,0} + A_{2,1}B_{1,0}$$

$$C_{0,1} = A_{0,1}B_{1,1} + A_{0,2}B_{2,1} + A_{0,0}B_{0,1}$$

$$C_{1,1} = A_{1,2}B_{2,1} + A_{1,0}B_{0,1} + A_{1,1}B_{1,1}$$

$$C_{2,1} = A_{2,0}B_{0,1} + A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{0,2} = A_{0,2}B_{2,2} + A_{0,0}B_{0,2} + A_{0,1}B_{1,2}$$

$$C_{1,2} = A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} + A_{2,0}B_{0,2}$$

Násobení hustých matic

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

(a) Initial alignment of A

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

(b) Initial alignment of B

Zdroj: A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Pearson/Addison Wesley, 2003

Násobení hustých matic

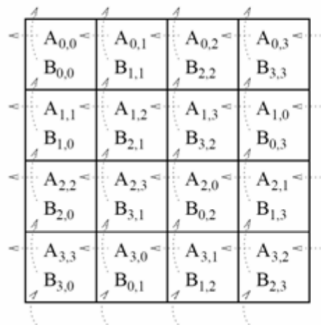
► a sumu

$$C_{i,j} = \sum_{k=0}^2 A_{i,k} B_{k,j}$$

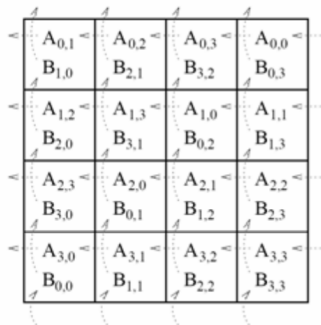
zaměníme za

$$C_{i,j} = \sum_{k=0}^2 A_{i,(k+i+j)\%3} B_{(k+i+j)\%3,j}$$

Násobení hustých matic



(c) A and B after initial alignment

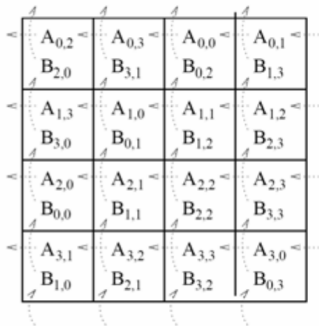


(d) Submatrix locations after first shift

Zdroj: A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Pearson/Addison Wesley, 2003

- v každém kroku algoritmu bloky matice A rotují o jeden doleva a bloky matice B o jeden nahoru

Násobení hustých matic



(e) Submatrix locations after second shift

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

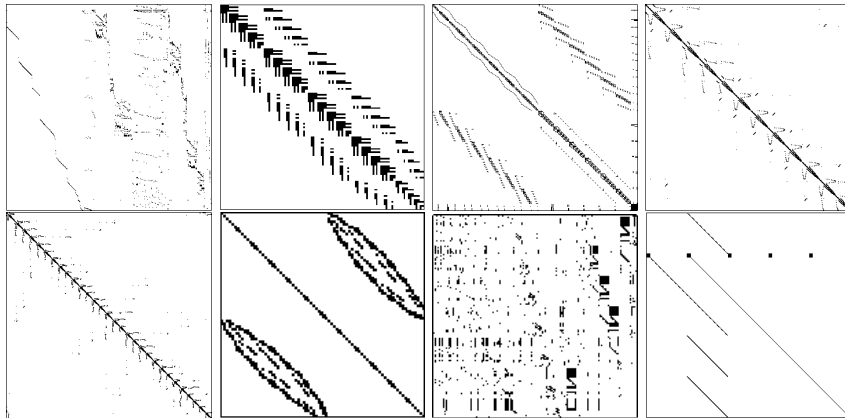
(f) Submatrix locations after third shift

Zdroj: A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Pearson/Addison Wesley, 2003

Řídké matice

Definition

Řídká matice je taková matice, která má většinu svých prvků nulových.



Zdroj: Matrix market

Řídké matice

- ▶ U takovýchto matic ukládáme jen nenulové prvky, čímž lze výrazně redukovat paměťové nároky.
- ▶ Existuje celá řada formátů pro ukládání řídkých matic.
- ▶ Mezi nejznámější patří:
 - ▶ COO - *coordinate list*
 - ▶ CSR - *compressed sparse rows*
 - ▶ Ellpack
- ▶ Cílem je maximálně redukovat objem dat nezbytný k reprezentaci matice.
- ▶ Jednak to šetří paměť, ale většina maticových operací je limitována datovou propustností, tj. méně dat = lepší výkon.

Řídké matice - COO formát

5		2					
		1					
	3						
5							
4							
	2					9	
		2			5		3
				1			7

values[]
columns[]
rows[]

5	2	1	3	5	4	2	9	2	5	3	1	7
0	2	2	1	0	0	1	6	2	5	7	4	7
0	0	1	2	3	4	5	5	6	6	6	7	7

Tento formát využívá tři pole, každé má tolik prvků, kolik je nenulových maticových prvků:

- ▶ `values` - hodnoty maticových prvků,
- ▶ `columns` - sloupcové indexy maticových prvků,
- ▶ `rows` - řádkové indexy maticových prvků.

Řídké matice - CSR formát

5		2												
		1												
	3													
5														
4														
	2						9							
		2			5			3						
				1						7				
values[]														
columns[]														
5	2	1	3	5	4	2	9	2	5	3	1	7		
0	2	2	1	0	0	1	6	2	5	7	4	7		
rowPointers[]														
0	2	3	4	5	6	8	11	13						

Předpokládáme, že nenulové prvky jsou uloženy po řádcích.

Pole `rows` nahradíme za `rowPointers`:

- ▶ to má velikost rovnu počtu řádku + 1
- ▶ pro každý řádek obsahuje index prvního prvku v řádku v polích `values` a `columns`
- ▶ toto pole se dá napočítat pomocí exkluzivního prefix-sum aplikovaného na pole udávající počet nenulových prvků v daném řádku matice

Řídké matice - Ellpack formát

5		2					
		1					
	3						
5							
4							
	2					9	
		2			5		3
			1				7

values[]			columns[]		
5	2	0	0	2	*
1	0	0	2	*	*
3	0	0	1	*	*
5	0	0	0	*	*
4	0	0	0	*	*
2	9	0	1	6	*
2	5	3	2	5	7
1	7	0	5	7	*

Předpokládáme, že v každém řádku je počet nenulových prvků téměř stejný.

- ▶ Pokud m je maximální počet nenulových prvků v jednotlivých řádcích matice, pak k -tý řádek začíná v polích `values` a `columns` na pozici $k \cdot m$.
- ▶ Do řádků, které mají menší zaplnění nenulovými prvky vložíme zarovnávací nuly, tzv. *padding elements*.

Řídké matice - SpMV

SpMV = *sparse-matrix vector multiplication*

```
1 void SpMV( const CSRMatrix& A, const Vector& x, Vector& b )
2 {
3     for( int row = 0; row < A.getRows(); row++ ) {
4         double aux( 0.0 );
5         const int begin = A.rowPointers[ row ];
6         const int end = A.rowPointers[ row + 1 ];
7         for( int i = begin; i < end; i++ )
8             aux += x[ A.columns[ i ] ] * A.values[ i ];
9         b[ row ] = aux;
10    }
11 }
```

- ▶ Do polí `values` a `columns` přistupujeme sekvenčně máme tak optimální přístupy do paměti.
- ▶ Vkládání nových prvků do formátů CSR je velice náročné.

Matice v TNL

TNL podporuje následující typy matic:

- ▶ husté matice,
- ▶ obecné řídké matice,
- ▶ tridiagonální a mutlidiagonální matice,
- ▶ lambda matice pro bezmaticové metody (*matrix-free methods*).

Matice v TNL - husté matice

Hustá matice je v TNL definována jako:

```
1  template< typename Real = double,  
2             typename Device = TNL::Devices::Host,  
3             typename Index = int,  
4             typename ElementsOrganization,  
5             typename RealAllocator >  
6  struct TNL::Matrices::DenseMatrix;
```

kde

- ▶ `Real` je typ maticových prvků.
- ▶ `Device` je typ zařízení, na kterém bude matice alokována.
- ▶ `Index` je typ pro indexování maticových prvků.
- ▶ `ElementsOrganization` definuje organizaci matice v paměti:
 - ▶ `TNL::Algorithms::Segments::RowMajorOrder` je uložení po řádcích - defaultní pro CPU,
 - ▶ `TNL::Algorithms::Segments::ColumnMajorOrder` je uložení po sloupcích - defaultní pro GPU.
- ▶ `RealAllocator` je alokátor pro maticové prvky.

Matice v TNL - husté matice

Nastavení maticových prvků:

```
1  template< typename Device >
2  void initializerListExample()
3  {
4      TNL::Matrices::DenseMatrix< double, Device > matrix {
5          { 1, 2, 3, 4, 5, 6 },
6          { 7, 8, 9, 10, 11, 12 },
7          { 13, 14, 15, 16, 17, 18 }
8      };
9
10     std::cout << "General dense matrix: " << std::endl << matrix << std::endl;
11
12     TNL::Matrices::DenseMatrix< double, Device > triangularMatrix {
13         { 1 },
14         { 2, 3 },
15         { 4, 5, 6 },
16         { 7, 8, 9, 10 },
17         { 11, 12, 13, 14, 15 }
18     };
19
20     std::cout << "Triangular dense matrix: "
21               << std::endl << triangularMatrix << std::endl;
22 }
```

Matice v TNL - husté matice

Nastavení maticových prvků:

```
1  template< typename Device >
2  void addElements()
3  {
4      TNL::Matrices::DenseMatrix< double, Device > matrix( 5, 5 );
5
6      for( int i = 0; i < 5; i++ )
7          matrix.setElement( i, i, i );
8
9      std::cout << "Initial matrix is: " << std::endl << matrix << std::endl;
10
11     for( int i = 0; i < 5; i++ )
12         for( int j = 0; j < 5; j++ )
13             matrix.addElement( i, j, 1.0, 5.0 );
14
15     std::cout << "Matrix after addition is: " << std::endl << matrix << std::endl;
16 }
```

Matice v TNL - husté matice

Nastavení maticových prvků:

```
1  template< typename Device >
2  void forRowsExample()
3  {
4      using MatrixType = TNL::Matrices::DenseMatrix<double, Device>;
5      using RowView = typename MatrixType::RowView;
6      const int size = 5;
7      MatrixType matrix( size, size );
8
9      // Set the matrix elements.
10     auto f = [] __cuda_callable__ ( RowView& row ) {
11         const int& rowIdx = row.getRowIndex();
12         if( rowIdx > 0 )
13             row.setValue( rowIdx - 1, -1.0 );
14         row.setValue( rowIdx, rowIdx + 1.0 );
15         if( rowIdx < size - 1 )
16             row.setValue( rowIdx + 1, -1.0 );
17     };
18     matrix.forAllRows( f );
19     std::cout << matrix << std::endl;
20
21     // Now divide each matrix row by its largest element.
22     matrix.forAllRows( [] __cuda_callable__ ( RowView& row ) {
23         double largest = std::numeric_limits< double >::lowest();
24         for( auto element : row )
25             largest = TNL::max( largest, element.value() );
26         for( auto element : row )
27             element.value() /= largest;
28     } );
29     std::cout << matrix << std::endl;
30 }
```

Matice v TNL - řídké matice

Řídká matice je v TNL definována jako:

```
1 template<
2     typename Real = double,
3     typename Device = TNL::Devices::Host,
4     typename Index = int,
5     typename MatrixType =
6         TNL::Matrices::GeneralMatrix,
7     typename Format =
8         TNL::Algorithms::Segments::CSR>
9 class TNL::Matrices::SparseMatrix;
```

- ▶ `Real` je typ maticových prvků.
 - ▶ pro `bool` jde o binární matice, které neukládají hodnoty prvků
- ▶ `Device` je typ zařízení, na kterém bude matice alokována.
- ▶ `Index` je typ pro indexování maticových prvků.

- ▶ `MatrixType` může být:
 - ▶ `TNL::Matrices::GeneralMatrix`
 - ▶ `TNL::Matrices::SymmetricMatrix`
- ▶ jak formát lze použít např.
 - ▶ Ellpack
 - ▶ Sliced Ellpack
 - ▶ Chunked Ellpack
 - ▶ Bisection Ellpack
 - ▶ CSR - scalar, vector, light
 - ▶ Adaptive CSR

Matice v TNL - řídké matice

Nastavení maticových prvků:

```
1  template< typename Device >
2  void initializerListExample()
3  {
4      TNL::Matrices::SparseMatrix< double, Device > matrix1 (
5          5, // number of matrix rows
6          5, // number of matrix columns
7          { // matrix elements definition
8              { 0, 0, 2.0 },
9              { 1, 0, -1.0 }, { 1, 1, 2.0 }, { 1, 2, -1.0 },
10             { 2, 1, -1.0 }, { 2, 2, 2.0 }, { 2, 3, -1.0 },
11             { 3, 2, -1.0 }, { 3, 3, 2.0 }, { 3, 4, -1.0 },
12             { 4, 4, 2.0 } } );
13
14     std::cout << "General sparse matrix: " << std::endl << matrix1 << std::endl;
15
16     TNL::Matrices::SparseMatrix< double, Device > matrix2( 5, 5 );
17     matrix2.setElements( {
18         { 0, 0, 2.0 },
19         { 1, 0, -1.0 }, { 1, 1, 2.0 }, { 1, 2, -1.0 },
20         { 2, 1, -1.0 }, { 2, 2, 2.0 }, { 2, 3, -1.0 },
21         { 3, 2, -1.0 }, { 3, 3, 2.0 }, { 3, 4, -1.0 },
22         { 4, 4, 2.0 } } );
23
24     std::cout << "General sparse matrix: " << std::endl << matrix2 << std::endl;
25 }
```

Matice v TNL - řídké matice

Nastavení maticových prvků:

```
1  template< typename Device >
2  void initializerListExample()
3  {
4      std::map< std::pair< int, int >, double > map;
5      map.insert( std::make_pair( std::make_pair( 0, 0 ), 2.0 ) );
6      map.insert( std::make_pair( std::make_pair( 1, 0 ), -1.0 ) );
7      map.insert( std::make_pair( std::make_pair( 1, 1 ), 2.0 ) );
8      map.insert( std::make_pair( std::make_pair( 1, 2 ), -1.0 ) );
9      map.insert( std::make_pair( std::make_pair( 2, 1 ), -1.0 ) );
10     map.insert( std::make_pair( std::make_pair( 2, 2 ), 2.0 ) );
11     map.insert( std::make_pair( std::make_pair( 2, 3 ), -1.0 ) );
12     map.insert( std::make_pair( std::make_pair( 3, 2 ), -1.0 ) );
13     map.insert( std::make_pair( std::make_pair( 3, 3 ), 2.0 ) );
14     map.insert( std::make_pair( std::make_pair( 3, 4 ), -1.0 ) );
15     map.insert( std::make_pair( std::make_pair( 4, 4 ), 2.0 ) );
16
17     TNL::Matrices::SparseMatrix< double, Device > matrix1( 5, 5, map );
18     TNL::Matrices::SparseMatrix< double, Device > matrix2( 5, 5 );
19     matrix2.setElements( map );
20
21     std::cout << "General sparse matrix: " << std::endl << matrix1 << std::endl;
22 }
```


Matice v TNL - řídké matice

Nastavení maticových prvků:

```
1  template< typename Device >
2  void forRowsExample()
3  {
4      /**
5       * Set the following matrix (dots represent zero matrix elements):
6       * /  2 . . . . \
7       * | 1 2 1 . . |
8       * | . 1 2 1. . |
9       * | . . 1 2 1 |
10      * \ . . . . 2 /
11      */
12      const int size( 5 );
13      using MatrixType = TNL::Matrices::SparseMatrix< double, Device >;
14      using RowView = typename MatrixType::RowView;
15      MatrixType matrix( { 1, 3, 3, 3, 1 }, size );
16
17      // Set the matrix elements.
18      auto f = [] __cuda_callable__ ( RowView% row ) {
19          const int rowIdx = row.getRowIndex();
20          if( rowIdx == 0 )
21              row.setElement( 0, rowIdx, 2.0 );    // diagonal element
22          else if( rowIdx == size - 1 )
23              row.setElement( 0, rowIdx, 2.0 );    // diagonal element
24          else
25          {
26              row.setElement( 0, rowIdx - 1, 1.0 ); // elements below the diagonal
27              row.setElement( 1, rowIdx, 2.0 );    // diagonal element
28              row.setElement( 2, rowIdx + 1, 1.0 ); // elements above the diagonal
29          }
30      };
31      matrix.forAllRows( f );
32      std::cout << matrix << std::endl;
33
34      // Divide each matrix row by a sum of all elements in the row.
35      matrix.forAllRows( [] __cuda_callable__ ( RowView% row ) {
36          double sum = 0.0;
37          for( auto element : row )
38              sum += element.value();
39          for( auto element: row )
40              element.value() /= sum;
41      } );
42      std::cout << matrix << std::endl;
43  }
```

Matice v TNL - redukce

Matice v TNL nabízí flexibilní redukci v rámci maticových řádků:

- ▶ Příklad pro násobení husté matice s vektorem.
- ▶ Příklad pro výpočet maximové normy husté matice.
- ▶ Příklad pro násobení řídké matice s vektorem.