Hledání nejkratší cesty v grafu

Jakub Javůrek & Martin Beránek

16. května 2016

Obsah

Seznam obrázků

1 Definice problému

1.1 Zadání

Implementujte minimálně tyto algoritmy hledání nejkratších cest v grafu (mezi všema dvojicemi uzlů):

- Dijkstra viz, nutná úprava algoritmu (najde všechny vzdálenosti z 1 bodu).
- Floydův-Warshallův algoritmus.
- Úkol: * x86 a Xeon.
- Paralelizace obout metod, u Dijkstry každé vlákno provádí hledání z jiného startovního bodu.
- CUDA.
- Paralelizace jen Floydův-Warshallova algoritmu.

2 Sekvenční řešení a implementace

2.1 Floyd Warshall algoritmus

Floyd-Warshall algoritmus slouží k vytvoření nejkratších cest v grafu mezi všemi dvojicemi vrcholů grafu [1]. Vzdálenosti mezi vrcholy jsou zaneseny v matici přechodů, ty nesmějí mít záporné hodnoty.

Psedokód:

```
procedure [array] FloydWarshall(D, P)
    for k in 1 to n do
        for i in 1 to n do
            for j in 1 to n do
                if D[i][j] > D[i][k] + D[k][j] then
                    D[i][j] = D[i][k] + D[k][j]
                    P[i][j] = P[k][j]
    return P
```

V poli P jsou uloženy předchůdci pro směrování v grafu. V poli D jsou uloženy vzdálenosti. Složitost algoritmu je $O(|U|^3)$.

2.1.1 Implementace

Následující část obsahuje implementaci algoritmu v jazyce `C++`.

```
void floyd(int **matrix, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i == j)
```

```
                matrix[i][j] = 0;
            else if (matrix[i][j] == 0)
                matrix[i][j] = LARGE_INT;
        }
    }
    int i, j, k;
    for (k = 0; k < n; k++)
    {
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                if (matrix[i][j] > matrix[i][k] + matrix[k][j])
                {
                    matrix[i][j] = matrix[i][k] + matrix[k][j];
                }
            }
        }
    }
```

2.2 Dijkstrův algoritmus

Dijkstrův algoritmus slouží pro nalezení nejkratších cest v grafu pro jeden vrchol grafu [2]. Algoritmus pracuje s maticí přechodu, která neobsahuje záporné hrany.

Psedokód:

```
procedure [array], [array] Dijkstra(Graph, source, D):
    create vertex set Q
      for v in Graph:
         D[v] = INFINITY
          prev[v] = UNDEFINED
          add v to Q

      D[source] = 0

      while Q is not empty:
          u = vertex in Q with min D[u]
          remove u from Q

          for neighbor v in u:
              alt = D[u] + length(u, v)
              if alt < D[v]:
                  dist[v] = alt
                  prev[v] = u
      return D, P
```

2.2.1 Implementace

Následující část obsahuje implementaci algoritmu v jazyce C++.

```
int * dijsktra(int ** edgeMatrix, int vertexCnt, int s){
    int * p = new int[vertexCnt];
    int * d = new int[vertexCnt];
    set <int> * N = new set<int>;
    for (int i = 0; i < vertexCnt; i++){
        d[i] = LARGE_INT; // infinity
        p[i] = -1; // uknown
        N->insert(i);
    }
    d[s] = 0;
    int next = s;
    while (!N->empty()){
        int u = next;
        N->erase(u);
        next = *N->begin();

        set<int>::iterator it;
        for (it = N->begin(); it != N->end(); ++it)
        {
            if (edgeMatrix[u][*it] > 0)
            {
                int alt = d[u] + edgeMatrix[u][*it];
if(alt < d[*it]){
d[*it] = alt;
}
                if (alt < d[*it]){
                    p[*it] = u;
                }
            }
            if (d[next] > d[*it])
                next = *it;
        }
    }
    return d;
}
```

Jelikož je v zadáno, že výsledkem musí být obsahovat vzdálenost všech vrcholy, je algoritmus volán postupně se všemi vrcholy grafu.

3 Instance dat

Pro oba algoritmy byla sestavená matice s velikostí $4000x4000$ s celkovou velikostí 31 MB. Matice byla generována pomocí předepsané aplikace `generator`. Následně byla upravena do jiné formy pro změnění vzdáleností mezi body. Z vygenerovaných vzdáleností 1 se stali náhodná čísla v řádu desítek.

Pro náhled je uveden začátek matice:

```
4000
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  ...
```

```
0 0 0 0 8 0 0 0 0 0 0 0 0 0  ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0  ...
0 0 4 0 0 0 0 0 0 0 0 0 0 0  ...
. . . . . .
. . . . . .
. . . . . .
```

4 OpenMP

4.1 Floyd Warshall algoritmus

Algoritmus byl upraven o direktivy pro překladač, který zapínají `OpenMP`. Vektorizaci bylo možné použít po úpravě podmínky v `for` cyklu. Byla nahrazena funkcí `min`.

```
void floyd(int **matrix, int n, int threadCnt)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i == j)
                matrix[i][j] = 0;
            else if (matrix[i][j] == 0)
                matrix[i][j] = LARGE_INT;
        }
    }
    if (threadCnt > 0)
        omp_set_num_threads(threadCnt); // nastavení počtu vláken

    int i, j, k;
    for (k = 0; k < n; k++)
    {
        #pragma omp parallel for private(i,j) // direktiva pro OpenMP
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                // upraveno pro vektorizaci
                matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);
            }
        }
    }

}
```

Překladač cyklus nevektorizoval s odůvodněním:

```
floyd.cpp:36: note: not vectorized: not suitable for gather load _33 = *_32;
floyd.cpp:36: note: bad data references.
```

K vektorizaci nedošlo kvůli nekonzistentní alokované paměti.

4.2 Dijkstrův algoritmus

Jedinou změnou pro Dijkstrův algoritmus bylo přidání `OpenMP` direktiv. Větší změny nebyly možné kvůli velkým datovým závislostem ve `for` cyklech. Ačkoliv bylo přidání jedno volání funkce `min`, zrychlení nebylo žádné, protože data, nad kterými je funkce vykována, závisí na zbytku těla cyklu.

Dalším problémem je použití složité datové struktury pro hledání minima v grafu. Práce s ní algoritmus samozřejmě ještě víc zpomaluje.

```
    if (threadCnt > 0)
        omp_set_num_threads(threadCnt); // nastavení počtu vláken

    #pragma omp parallel for private(j) // direktiva pro OpenMP
    for(j = 0; j < stc; j++){
        d[j] = dijsktra(matrix, stc, j);
    }


int * dijsktra(int ** edgeMatrix, int vertexCnt, int s){
    int * p = new int[vertexCnt];
    int * d = new int[vertexCnt];
    set <int> * N = new set<int>;
    for (int i = 0; i < vertexCnt; i++){
        d[i] = LARGE_INT; // infinity
        p[i] = -1; // uknown
        N->insert(i);
    }
    for (int i = 0; i < vertexCnt; i++){
        N->insert(i);
    }
    d[s] = 0;
    int next = s;
    while (!N->empty()){
        int u = next;
        N->erase(u);
        next = *N->begin();

        set<int>::iterator it;
        for (it = N->begin(); it != N->end(); ++it)
        {
            if (edgeMatrix[u][*it] > 0)
            {
                int alt = d[u] + edgeMatrix[u][*it];
                d[*it] = min(alt, d[*it]);
                if (alt < d[*it]){
                    p[*it] = u;
                }
            }
            if (d[next] > d[*it])
                next = *it;
```

```
        }
    }
    return d;
}
```

Překladač nevektorizoval cyklus hlavního výpočtu s odůvodněním:

```
dijks.cpp:32: note: not vectorized: control flow in loop.
```

K vektorizaci nedošlo kvůli řídícím strukturám v cyklu.

Naopak se povedl vektorizovat výpočet na začátku funkce nastavující `LARGE_INT` a `-1`. Překladač vypsal zprávu:

```
Vectorizing loop at dijks.cpp:20
dijks.cpp:20: note: create runtime check for data references LARGE_INT and *_27
dijks.cpp:20: note: create runtime check for data references LARGE_INT and *_30
dijks.cpp:20: note: create runtime check for data references *_27 and *_30
dijks.cpp:20: note: created 3 versioning for alias checks.
dijks.cpp:20: note: === vect_do_peeling_for_loop_bound ===
Setting upper bound of nb iterations for epilogue loop to 2
dijks.cpp:20: note: LOOP VECTORIZED.
dijks.cpp:15: note: vectorized 1 loops in function.
dijks.cpp:20: note: Completely unroll loop 2 times
```

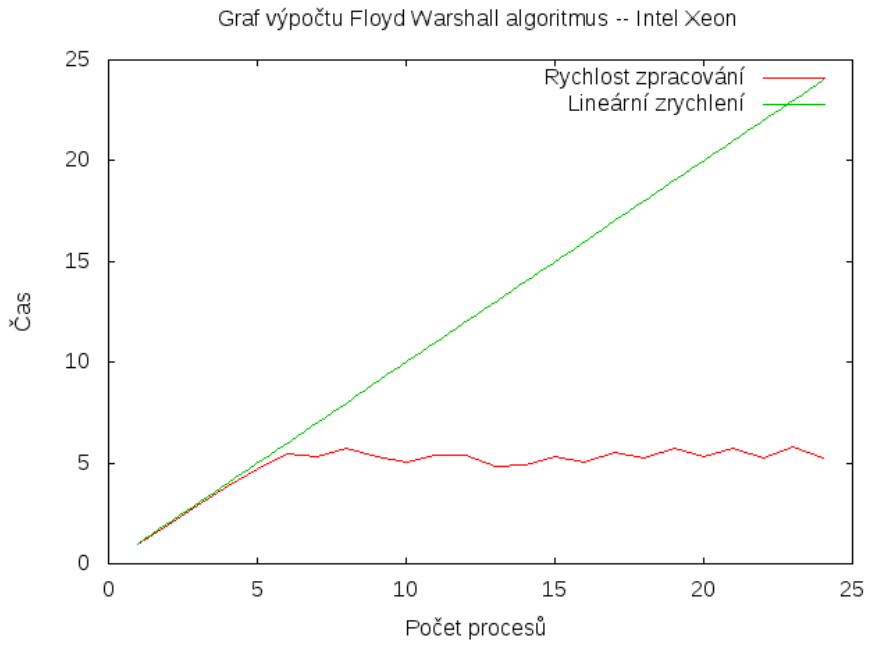
4.3 Naměřená data

První měření proběhlo na `Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz`. První měření proběhlo na Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz. Počet vláken provádějících výpočet byl zvyšován v následující řadě: 1, 2, ..., 24. Měření probíhalo pomocí skriptu v noci s kontrolou, zdali na clusteru nejsou další uživatelé. Měření proběhlo ve tři hodiny ráno z noci z neděle na pondělí.

Doby výpočtu je v následující tabulce (1):

Tabulka 1: Čas výpočtu Floyd Warshall algoritmu

Počet vláken	Rychlost ve vteřinách
1	155.992
2	79.9949
3	53.5526
4	40.3126
5	33.2484
6	28.4757
7	29.1981
8	27.3738
9	29.5177
10	30.8656
11	29.0657
12	29.0995
13	32.1369
14	31.6199
15	29.4036
16	30.7675
17	28.1378
18	29.7564
19	27.2729
20	29.3864
21	27.3018
22	29.6423
23	27.0364
24	29.5721

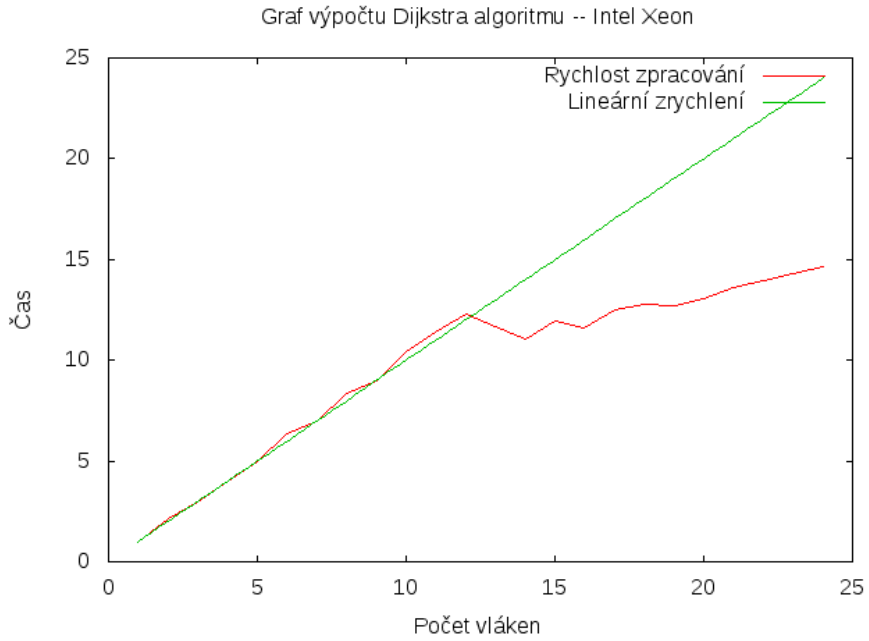


Obrázek 1: Graf zrychlení výpočtu Floyd

Zrychlení Floyd Warshallova algoritmu pro velké počty vláken přestává držet lineární zrychlení.

Tabulka 2: Čas výpočtu Dijkstra algoritmu

Počet vláken	Rychlost ve vteřinách
1	360.883
2	170.996
3	120.683
4	90.68
5	72.2292
6	57.0565
7	51.7025
8	43.2244
9	40.1845
10	34.5069
11	31.6283
12	29.3998
13	31.0053
14	32.7462
15	30.2443
16	31.1353
17	28.8648
18	28.312
19	28.4545
20	27.6419
21	26.5522
22	25.8142
23	25.2867
24	24.6004



Obrázek 2: Graf zrychlení výpočtu Dijkstra

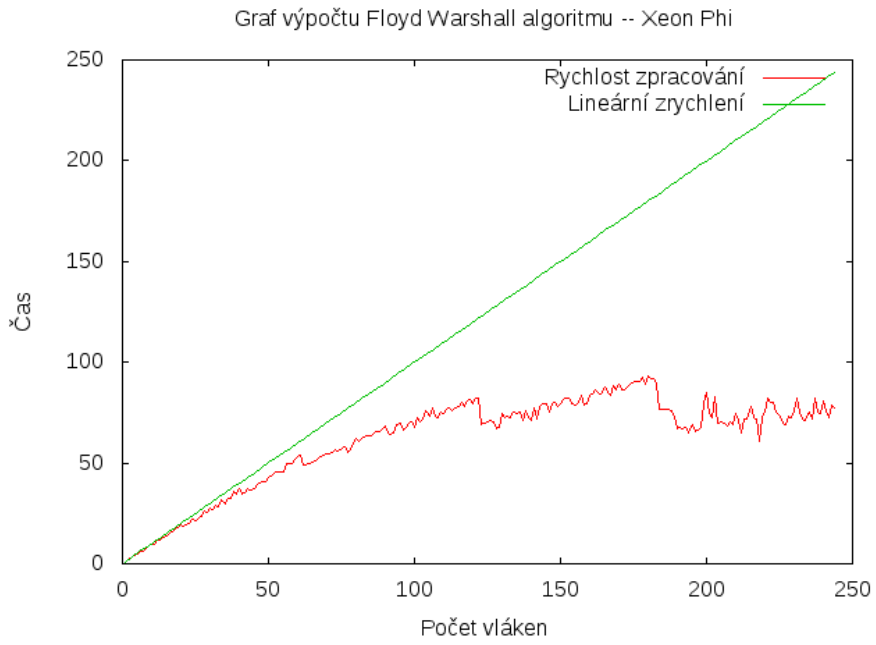
Zrychlení algoritmu se drží lineárního zrychlení. Důvodem také může být, že celý výpočet je v jedné instanci o řád pomalejší, než v případu Floyd Warshalla. Ve velkých počtech instancí se však doba zpracování přestává lišit.

Druhé měření proběhlo na Intel(R) Xeon(R) Phi(TM). Měření proběhlo pomocí skriptu v době, kdy na clusteru nikdo jiný nebyl. Měření tedy mohlo proběhnout v řadě 1, 2, ..., 244 vláken.

V následující tabulce jsou uvedeny pouze referenční hodnoty (3):

Tabulka 3: Čas výpočtu Floyd Warshall algoritmu

Počet vláken	Rychlost ve vteřinách
1	1337.32
10	142.572
20	68.1147
30	48.1608
40	35.6807
50	31.3623
60	25.2053
70	24.6364
80	21.6174
90	19.6177
100	19.7071
110	17.9346
120	16.7724
130	17.9536
140	18.7667
150	16.861
160	15.9678
170	14.9669
180	14.3501
190	19.8682
200	15.6912
210	17.896
220	17.7623
230	17.6404
240	16.5851

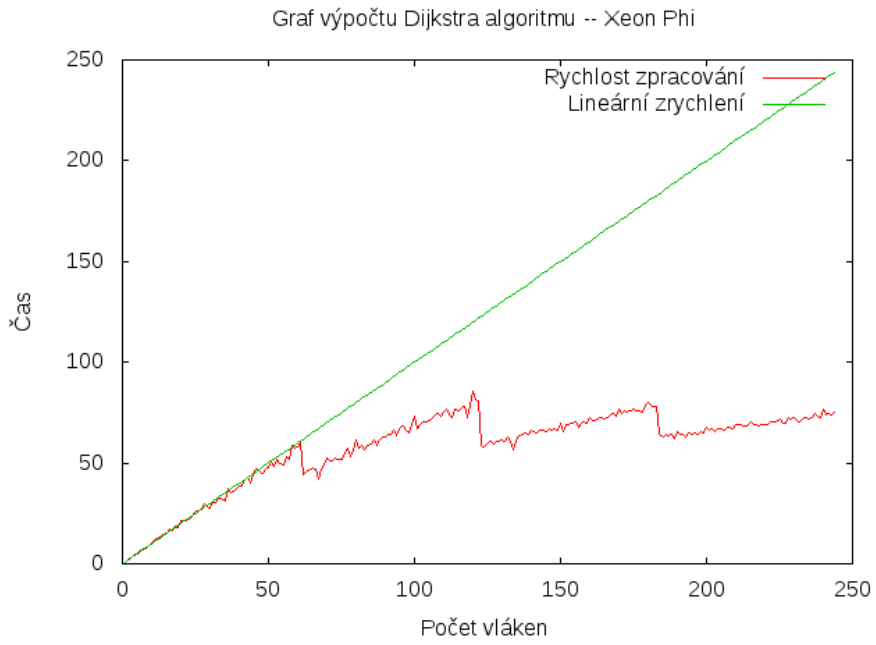


Obrázek 3: Graf zrychlení výpočtu Floyd

Zrychlení Floyd Warshallova algoritmu pro velké počty vláken přestává držet lineární zrychlení. Výkyvy po 61 jádrech ukazují na omezení v počtu vláken na kartu. V případě počtu vláken 62 jsou zabraná všechna jádra a jedno z nich pracuje se dvěma vlákny. Tím je celý výpočet zpomalen. Zrychlení se vylepšuje s použitím více 9 vláken na jádro. K největšímu zrychlení došlo při použití 183 vláken ($3 \cdot 61$).

Tabulka 4: Čas výpočtu Dijkstra algoritmu

Počet vláken	Rychlost ve vteřinách
1	2622.89
10	253.226
20	123.594
30	93.7867
40	67.3802
50	55.222
60	45.397
70	49.7902
80	42.7281
90	41.8506
100	35.6632
110	34.7739
120	30.6839
130	42.5086
140	39.5326
150	37.6043
160	36.0399
170	34.2481
180	32.6829
190	40.1083
200	38.612
210	37.8351
220	37.9962
230	36.09
240	34.1638



Obrázek 4: Graf zrychlení výpočtu Dijkstra

Zrychlení Dijkstra algoritmu pro velké počty vláken přestává držet lineární zrychlení. Na grafu je vidět jev, který byl už na měření Floyd Warshall algoritmu. Největší zrychlení nastalo při 122 vláknech.

5 CUDA

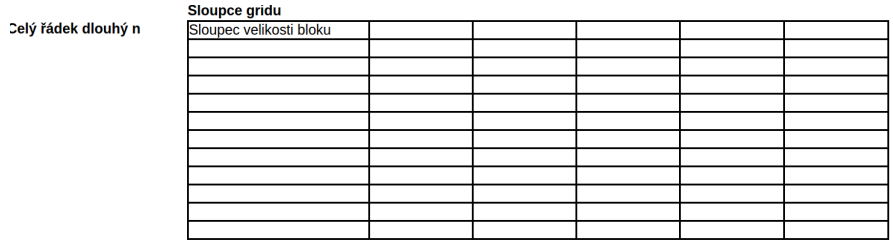
5.1 Floyd Warshall algoritmus

Volání grafické karty předchází inicializace a kopírování matice do `CUDA` zařízení. Prostor vláken výpočtu je rozdělen ve dvou dimenzích pomocí `dim3` gridu.

```
void gpu_floyd(int *matrix, int n){
  int *cumatrix;
  cudaMalloc((void **)&cumatrix, n * n * sizeof(int));
  cudaMemcpy(cumatrix, matrix,  n * n * sizeof(int), cudaMemcpyHostToDevice);

  dim3 dimGrid(( n + BLOCK_SIZE - 1 ) / BLOCK_SIZE, n);

  floyd_prepare<<<dimGrid, BLOCK_SIZE>>>(cumatrix, n);

  cudaThreadSynchronize();

  for (int k = 0; k < n; k++){
    floyd_kern<<<dimGrid, BLOCK_SIZE>>>(k, cumatrix, n);
    cudaThreadSynchronize();
```

```
  }

  cudaMemcpy(matrix, cumatrix, sizeof(int)*n*n,cudaMemcpyDeviceToHost);
  cudaFree(cumatrix);
}
```

Velikost bloku je stejná jako počet použitých vláken. Jejich dělení je na obrázku (5). Celý prostor je rozdělen tedy do dvou dimenzí rozdělených podle počtu alokovaných vláken.



Obrázek 5: Rozmístění vláken v gridu

Následně se zavolá funkce `floyd_prepare`, která na grafické kartě nastaví matici pro výpočet algoritmu. Po dokončení výpočtu se zavolá samotný algoritmus.

```
__global__ void floyd_prepare(int *matrix, int n){
  int column = blockIdx.x * blockDim.x + threadIdx.x;
  int idx = n * blockIdx.y + column;
  if(column >= n )
      return;
  if ( blockIdx.y == column)
    matrix[idx] = 0;
  else if(matrix[idx] == 0)
    matrix[idx] = INT_MAX/2;
}
```

Výpočet algoritmu [3]:

```
__global__ void floyd_kern(int k, int *matrix, int n){
  int column = blockIdx.x * blockDim.x + threadIdx.x; // pozice na radku
  if(column >= n ) // pokud přepluje, pravděpodobně jsme přejeli velikost bloku
      return;
  int idx = n * blockIdx.y + column; // přesná pozice v matici

  __shared__ int bmatch; // pro jedno k nejlepší hodnota v floyd

  if(threadIdx.x == 0)
      bmatch = matrix[n*blockIdx.y + k];

  __syncthreads();

  if(bmatch == INT_MAX/2)
```

```
     return;

  int tmp = matrix[k*n+column];

  if(tmp == INT_MAX/2)
    return;

  int current = bmatch + tmp;
  if(current < matrix[idx]){
    matrix[idx] = current; // matrix[i*n+j] = min(matrix[i*n+j], matrix[i*n+k] + matrix[k*n+j]);
  }
}
```

Na rozdíl od řešení pomocí cyklů každé vlákno představuje jeden samostatný kousek výpočtu. Všechna současně běžící vlákna mezi sebou sdílí `bmatch`, který představuje `matrix[i*n+k]`. Podle umístěná v gridu se rozhodnou, zdali je nejlepší cestou současná hodnota nebo součet `tmp` a `bmatch`, které představují `matrix[i*n+k] + matrix[k*n+j]`. V případě, že je `bmatch` nebo `tmp` roven nekonečnu, pak nemohou představovat v součtu nejlepší cestu a vlákno se ukončuje.

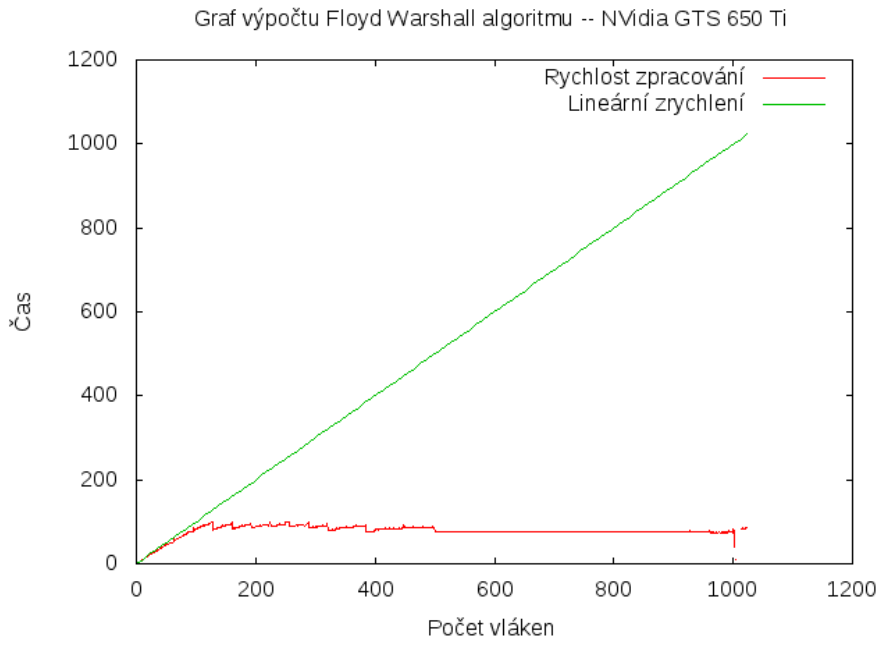
Měření proběhne pomocí měnění velikosti proměnnné `BLOCK_SIZE`, která představuje velikost sloupce v řádku a také počet vláken, které v buňce tabulky pracují.

5.2 Naměřená data

První měření probíhalo na `GTS 650 Ti`.

Tabulka 5: Čas výpočtu Floyd Warshall algoritmu na GTS 650 Ti v milisekundách

Počet vláken	Rychlost ve vteřinách
40	27695.6
80	14280.9
120	10626.7
160	10237
200	11102.6
240	10616.3
280	10881.6
320	10617.4
360	11483.2
400	12095
440	11870.1
480	11299.5
520	13275.9
560	13277.6
600	13277.2
640	13278.1
680	13277.2
720	13277.2
760	13278.5
800	13277.8
840	13277.4
880	13278.1
920	13104.1
960	12701
1000	12376.3

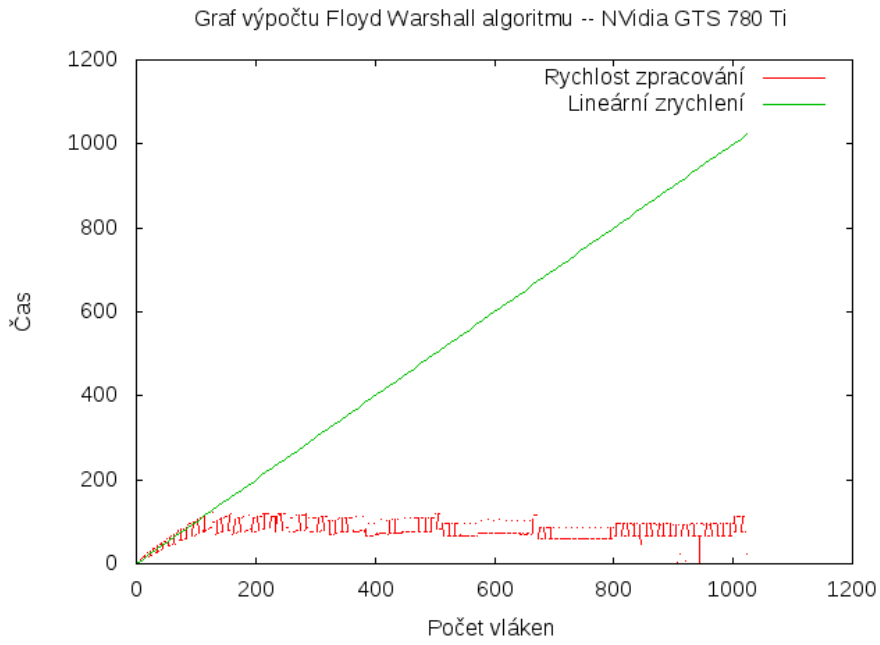


Obrázek 6: Graf zrychlení výpočtu Floyd Warshall

Druhé měření probíhalo na `GTS 780 Ti` na školním clusteru `star.fit.cvut.cz`.

Tabulka 6: Čas výpočtu Floyd Warshall algoritmu na GTS 780 Ti v milisekundách

Počet vláken	Rychlost ve vteřinách
33	13517.8
99	3383.1
165	3184.83
231	3024.53
264	4341.66
297	3113.47
363	4447.31
429	3211.24
495	4469.42
561	5044.13
594	4665.71
627	4701.09
693	5699.16
759	3928.79
825	5163.5
891	3555.05
924	5157.97
957	5192.25
1023	13610.1



Obrázek 7: Graf zrychlení výpočtu Floyd Warshall

5.3 Dijkstrův algoritmus

Volání grafické karty předchází inicializace a kopírování matice do `CUDA` zařízení. Prostor vláken výpočtu je rozdělen ve dvou dimenzích. Počet bloků odpovídá počtu uzlů děleným počtem vláken v bloku. Počet vláken v bloku je definován v `BLOCK_SIZE`.

```
.
.
.
int *cumatrix;
int *d;
cudaMalloc((void **)&cumatrix, stc * stc * sizeof(int));
cudaMemcpy(cumatrix, matrix,  stc * stc * sizeof(int), cudaMemcpyHostToDevice);
cudaMalloc((void **)&d, stc * stc * sizeof(int));

prepareArray<<<stc / BLOCK_SIZE, BLOCK_SIZE>>>(stc, d);
cudaError_t code = cudaThreadSynchronize();
if (code != cudaSuccess)
{
  fprintf(stdout, "GPUassert: %s \n", cudaGetErrorString(code));
}

dijsktra<<<stc / BLOCK_SIZE, BLOCK_SIZE>>>(cumatrix, stc, d);
code = cudaThreadSynchronize();
if (code != cudaSuccess)
{
```

```
  fprintf(stdout, "GPUassert: %s \n", cudaGetErrorString(code));
}

int *outM = new int[stc*stc];
cudaMemcpy(outM, d,  stc * stc * sizeof(int), cudaMemcpyDeviceToHost);
.
.
.
```

Následně se zavolá funkce `prepareArray`, která na grafické kartě nastaví matici pro výpočet algoritmu. Po dokončení výpočtu se zavolá samotný algoritmus.

```
__global__ void prepareArray(int vertexCnt, int* d)
{
  int threads = gridDim.x * gridDim.y * gridDim.z * blockDim.x * blockDim.y * blockDim.z; //celko
  int cycleCnt = (vertexCnt / threads > 0 ? vertexCnt / threads : 1); //velikost prace vlakna
  for (int cycle = 0; cycle < cycleCnt; cycle++)
  {
    int s = (blockIdx.x * blockDim.x + threadIdx.x) + threads * cycle;
    if(s >= vertexCnt)
      return;

    for (int i = 0; i < vertexCnt; i++)
    {
      d[vertexCnt *i+s] = INT_MAX / 2;
    }
  }
}
```

Jelikož na grafické kartě nelze použít implementaci ADT množiny std::set, vytvořili jsme pro naše účely jednoduchou vlastní implementaci:

```
class mySet
{
private:
  int size = 4000;
  bool N[4000]; //pro nase ucely staci takto omezit
  int cnt = 4000;

public:
  __device__ mySet(){}

  __device__ void init(int s)
  {
    this->cnt = s;
    for (int i = 0; i < s; i++)
    {
      N[i] = true;
    }
  }
```

```
  __device__ bool contains(int x)
  {
    return N[x];
  }

  __device__ void insert(int x)
  {
    if (N[x] == true)
      return;
    N[x] = true;
    cnt++;
  }

  __device__ void erase(int x)
  {
    if (N[x] == true)
    {
      N[x] = false;
      cnt--;
    }
  }

  __device__ bool empty()
  {
    return (cnt == 0);
  }

  __device__ int getCount()
  {
    return cnt;
  }
};
```

Výpočet algoritmu:

```
__global__ void dijsktra( int* __restrict__ edgeMatrix, int vertexCnt, int* d)
{
  int threads = gridDim.x * gridDim.y * gridDim.z * blockDim.x * blockDim.y * blockDim.z;
  int cycleCnt = (vertexCnt / threads > 0 ? vertexCnt / threads : 1); //velikost prace vlakna
  for (int cycle = 0; cycle < cycleCnt; cycle++)
  {
    int s = (blockIdx.x * blockDim.x + threadIdx.x) + threads * cycle;
    if(s >= vertexCnt)
      return;

    mySet N;

    N.init(vertexCnt);
```

```
    d[s*vertexCnt + s] = 0;
    while (!N.empty())
    {
      int localMin = INT_MAX;

      int cnt = N.getCount();
      int u = 0;
      int j = 0;
      for (int i = 0; i < vertexCnt && j < cnt; i++)
      {
        if (!N.contains(i)) continue;
        if (localMin > d[vertexCnt *i+s])
        {
          localMin = d[vertexCnt *i+s];
          u = i;
        }
        j++;
      }
      N.erase(u);

      for (int i = 0; i < vertexCnt; i++)
      {
        if (i == u || !N.contains(i)) continue;

        if (edgeMatrix[u + i*vertexCnt] > 0)
        {
          int alt = d[vertexCnt *u+s] + edgeMatrix[u + i*vertexCnt];
          atomicMin((d + vertexCnt * i + s), alt);
        }
      }
    }
  }
}
```

Idea za algoritmem je jednoduchá - vlákna si rozdělí matici grafu po sloupcích a postupně zpracovávají svůj přídel práce.

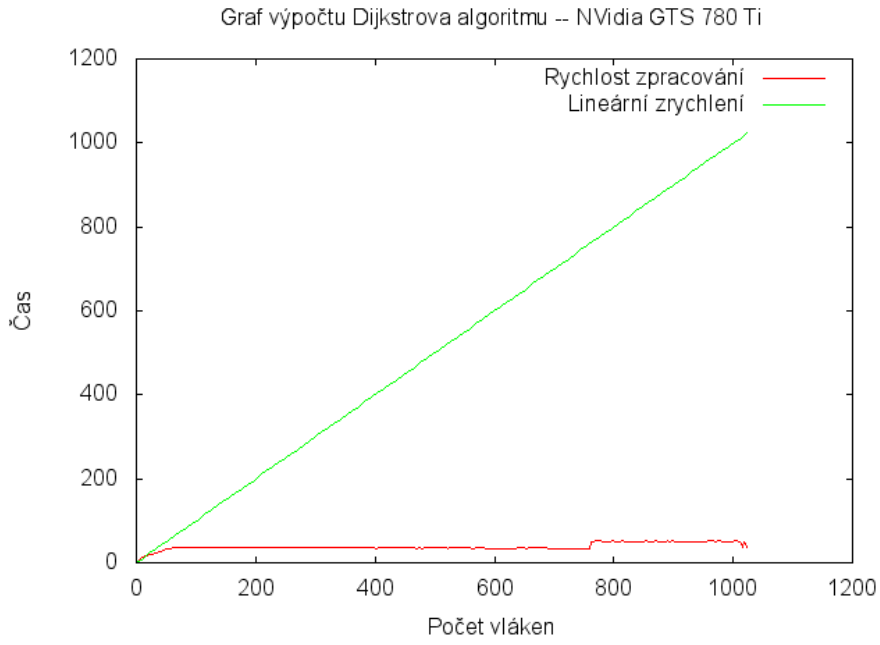
Měření proběhne pomocí měnění velikosti proměnnné `BLOCK_SIZE`, která představuje počet vláken, které v buňce tabulky pracují.

5.4 Naměřená data

Zde měření probíhalo pouze na `GTS 780 Ti` na školním clusteru `star.fit.cvut.cz`.

Tabulka 7: Čas výpočtu Dijkstrova algoritmu na GTS 780 Ti v milisekundách

Počet vláken	Rychlost ve vteřinách
11	97627.6
31	50964.5
47	38982.1
91	33748.9
151	33750
255	33738
403	34409
575	33763.7
763	23035.4
855	23018.2
951	23042.7
1023	23030.5



Obrázek 8: Graf zrychlení výpočtu Dijkstrova algoritmu

6 Závěr

Při paralelizaci na procesoru Intel Xeon dosahujeme většího zrychlení u dijkstrova algoritmu. Floyd-Warshallův zrychluje nejvíce 6x, kdežto dijkstrův až 12x - zřejmě lépe využívá virtuální vlákna. Díky tomu dosahují při vyšším počtu vláken oba algoritmy srovnatelných výsledků, přestože při sekvenčním běhu je Floyd-Warshallův přibližně dvakrát rychlejší (pro měřená vstupní data).

Při běhu na akcelerátoru Intel Xeon Phi dojde při vyšším počtu vláken k zanedbatelnému zrychlení u Floyd-Warshalla. Dijkstrův algoritmus je naopak mírně pomalejší - projevují se datové závislosti. Na obou grafech lze vidět úpadek ve výkonu při počtech vláken v intervalech po 61. Projevuje se zde omezení počtu vláken na kartu.

Implementace Floyd-Warshallova algoritmu v CUDA zrychluje prakticky lineárně přibližně do 100 vláken. Poté se již pohybuje okolo stálé konstanty. Dijkstrův algoritmus zrychluje jen asi do 50 vláken, poté je zde ještě menší skok okolo 800. Horší škálování je dáno jednak implmentací, druhak datovými závislostmi algoritmu.

Srovnáme-li časy výpočtů dosažené na grafické kartě s těmi z klasického CPU nebo akcelerátoru Xeon Phi vidíme, že u GPU dosahujeme zdaleka nejhorších výsledků - i při nejvyšších dosažených zrychleních se nevyrovnáme sekvenčnímu řešení na CPU.

Podle výsledků všech měření se zdá, že problém hledání nejkratší cesty je (za použití našich implementaci obou algoritmů) nejvýhodnější řešit na standardním CPU, zde Intel Xeon 2620 v2.

Reference

[1] ALGORITMY.NET: Floyd-Warshallův algoritmus. 2016, [cit. 2016-04-07]. Dostupné z: https://www.algoritmy.net/article/5207/Floyd-Warshalluv-algoritmus

[2] ALGORITMY.NET: Dijkstrův algoritmus. 2016, [cit. 2016-04-07]. Dostupné z: https://www.algoritmy.net/article/5108/Dijkstruv-algoritmus

[3] Hochberg, R.: Dynamic programming with CUDA – Part 1. 2016, [cit. 2016-05-02]. Dostupné z: http://www.shodor.org/media/content/petascale/materials/UPModules/dynamicProgrammingPartI/dynProgPt1ModuleDoc.pdf