Paralelism de date din algebra liniara – inmultirea matricilor si sortari (3 clase de algoritmi).

Se doreste analiza de scalabilitate si eficienta, la modificarea dimensiunii problemei (dimensiuni matrici, dimensiuni sir), respectiv la modificarea dimensiunii masinii (numar de thread-uri - daca acest lucru este posibil a fi controlat si in ce mod).

Paralelism de date din algebra liniara

Inmultirea matricilor Aceast exemplu implementează multiplicarea a 2 matrici care utilizează memoria partajată pentru a asigura reutilizarea datelor, multiplicarea matricilor se face folosind abordarea tiling.

Constrângeri hardware:

Aceasta este partea ușor de cuantificat. Apendicele F al ghidului de programare CUDA curent afișează o serie de limite puternice care limitează numărul de thread-uri per bloc pe care le poate lansa un kernel. Dacă depășiți oricare dintre acestea, nucleul dvs. nu va funcționa niciodată. Ele pot fi rezumate în general ca:

1. Fiecare bloc nu poate avea mai mult de 512/1024 thread-uri în total (Compute capability 1.x sau 2.x respeciv mai noi)
2. Dimensiunile maxime ale fiecărui bloc sunt limitate la [512,512,64] / [1024,1024,64] (Compute 1.x / 2.x respeciv mai noi)
3. Fiecare bloc nu poate consuma mai mult de 8k / 16k / 32k / 64k / 32k / 64k / 32k / 64k / 32k / 64k registre totale (Compute 1.0,1.1 / 1.2,1.3 / 2.x- / 3.0 / 3.2 / 3.5-5.2 / 5.3 / 6-6.1 / 6.2 / 7.0)
4. Fiecare bloc nu poate consuma mai mult de 16kb / 48kb / 96kb de memorie partajată (Compute 1.x / 2.x-6.2 / 7.0)

Dacă rămâneți între aceste limite, orice nucleu pe care îl puteți compila cu succes se va lansa fără eroare.

Reglarea performanțelor. Aceasta este partea empirică. Numărul de thread-uri pe bloc pe care le alegeți în cadrul constrângerilor hardware prezentate mai sus poate și afectează performanța codului care rulează pe hardware. Modul în care se comportă fiecare cod va fi diferit și singura modalitate reală de cuantificare a acestuia este evaluarea și profilarea atentă.

1. Numărul de thread-uri pe bloc ar trebui să fie un număr rotund multiplu al dimensiunii warp-ului (set de thread-uri care partajeaza acelasi cod), care este de 32 pe toate componentele hardware actuale.
2. Fiecare unitate multiprocessor de streaming de pe unitatea de procesare grafică (GPU) trebuie să aibă suficiente warp-uri active pentru a ascunde suficient toate memoriile diferite si latenta instructiunii pipeline a arhitecturii și a obține o viteză maximă de transfer. Abordarea ortodoxă aici este încercarea de a obține o ocupație optimă a hardware-ului.

Trebuie cunsocut faptul că dimensiunea blocului pe care o alegeți (în intervalul dimensiunilor blocurilor legale definite de constrângerile de mai sus) poate și are impact asupra vitezei de funcționare a codului, dar depinde de ce hardware este folosit și codul pe care îl executați.

Deoarece se lucreaza în multipli “warp size”, spațiul de căutare este foarte finit și cea mai bună configurație pentru o anumită bucată de cod este relativ ușor de găsit.

```
/**
 * Inmultirea matricilor (CUDA Kernel) pe device: C = A * B
 * wA este latimea matricei A si wB e latimea matricei B
 */
```

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A,float *B, int wA,int wB)
{
   // indexul blocului
   int bx = blockIdx.x;
   int by = blockIdx.y;

   // indexul threadului
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int aBegin = wA * BLOCK_SIZE * by;// Indexul primei sub-matrici a matricei A proceste de catre bloc
   int aEnd   = aBegin + wA - 1; // Indexul ultimei sub-matrici a matricei A proceste de catre bloc
   int aStep  = BLOCK_SIZE; // Dimensiunea pasului folosit pentru a itera prin sub-matricile lui A
   int bBegin = BLOCK_SIZE * bx; // Indexul primei sub-matrici a matricei B proceste de catre bloc
   int bStep  = BLOCK_SIZE * wB; // Dimensiunea pasului folosit pentru a itera prin sub-matricile lui B

   // Csub e folosit pentru a pastra elemental blocului sub-matrice
   // care e calculate de thread
   float Csub = 0;


   // Bucla peste toate sub-matricile lui A si B
   // necesare pentru a calcula blocul sub-matrice
   for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
      // Declararea vectorului de memorie partajata As folosit pentru
      // a stoca sub-matricea lui A
      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

      // Declararea vectorului de memorie partajata Bs folosit pentru
      // a stoca sub-matricea lui B
      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

      // Incarcarea matricilor din memoria device-ului
      // in memoria partajata; fiecare thread incarca
      // un element al fiecarei matrici
      As[ty][tx] = A[a + wA * ty + tx];
      Bs[ty][tx] = B[b + wB * ty + tx];

      // Sincronizare pentru a asigura ca sunt încarcate matricele
      __syncthreads();

      // Inmultirea celor doua matrice
      // fiecare thread calculează un element
      // din sub-matricea blocului
#pragma unroll

      for (int k = 0; k < BLOCK_SIZE; ++k) {
         Csub += As[ty][k] * Bs[k][tx];
      }
      // Sincronizare pentru a asigura că precedentul
      // calculul se face înainte de încărcarea a două noi
      // sub-materici ale lui A și B în următoarea iterație
      __syncthreads();
   }
```

```
    // Se scrie sub-matricea blocului în memoria dispozitivului;
     // fiecare thread scrie un element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Apelul metodei din host se face prin:

```
if (block_size == 16) {
        MatrixMulCUDA<16> <<< grid, threads >>>(d_C, d_A, d_B,
                                dimsA.x, dimsB.x);
    } else {
        MatrixMulCUDA<32> <<< grid, threads >>>(d_C, d_A, d_B,
                                dimsA.x, dimsB.x);}
```

Pentru testare s-a folosit GPU Device 0: "GeForce 920M" cu capabilitatea de calcul 3.5.

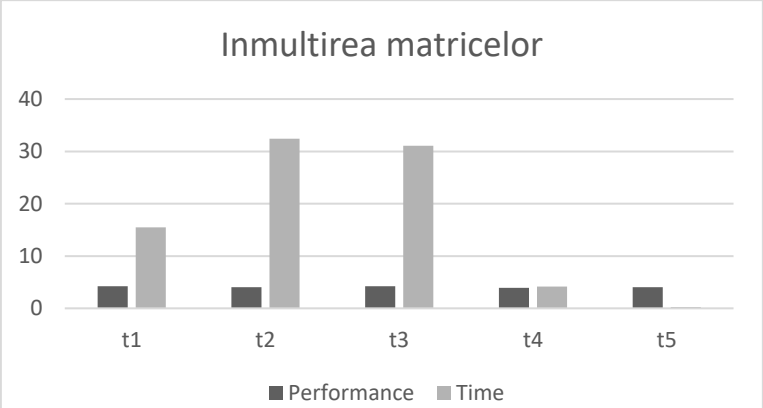
Performanta = `double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);`

Time = `float msecPerMatrixMul = msecTotal / nIter;`
`// nInter costant set to 300 – Numar de executii: Pentru determinarea timpului mediu de executie a algoritmului`

Size = `double flopsPerMatrixMul = 2.0 * static_cast<double>(dimsA.x) * static_cast<double>(dimsA.y) * static_cast<double>(dimsB.x);`

WorkgroupSize = `threads.x * threads.y`

Test1:
MatrixA(320,320), MatrixB(320,320)
Performance= 4.23 GFlop/s,
Time= 15.481 msec,
Size= 65536000 Ops,
WorkgroupSize= 1024 threads/block
Test2:
MatrixA(320,320), MatrixB(640,320)
Performance= 4.05 GFlop/s,
Time= 32.397 msec,
Size= 131072000 Ops,
WorkgroupSize= 256 threads/block
Test3:
MatrixA(320,320), MatrixB(640,320)
Performance= 4.21 GFlop/s,
Time= 31.098 msec,
Size= 131072000 Ops,
WorkgroupSize= 1024 threads/block
Test4:
MatrixA(160,160), MatrixB(320,160)
Performance= 3.91 GFlop/s,
Time= 4.189 msec,
Size= 16384000 Ops,
WorkgroupSize= 256 threads/block
Test5:
MatrixA(64,64), MatrixB(128,64)
Performance= 4.07 GFlop/s,
Time= 0.257 msec,
Size= 1048576 Ops,
WorkgroupSize= 1024 threads/block



Test	Performance	Time	Size	WorkgroupSize
t1	4.23	15.481	65536000	1024
t2	4.05	32.397	131072000	256
t3	4.21	31.098	131072000	1024
t4	3.91	4.189	16384000	256
t5	4.07	0.257	1048576	1024

Sorting networks

Aceast exemplu implementează sortarea bitonic și sortarea odd-even merge sort (cunoscută și sub numele de Batcher's sort), algoritmi aparținând clasei rețelelor de sortare. Deși în general subeficient, pentru secvențe mari în comparație cu algoritmi cu o complexitate algoritmică asimptotica mai bună (de exemplu, merge sort sau radix), aceștia pot fi algoritmii preferați de sortare pentru sortarea loturilor de perechi de vectori de dimensiuni mici și mijlocii (cheie, valoare) . Consultați un excelent tutorial al lui H. W. Lang http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/indexen.htm

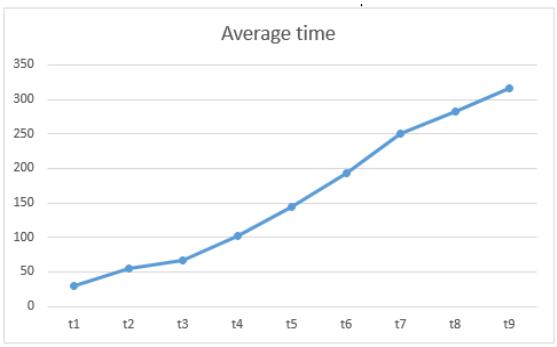
```
extern "C" uint bitonicSort(
   uint *d_DstKey, uint *d_DstVal, uint *d_SrcKey, uint *d_SrcVal, uint batchSize,   uint arrayLength, uint dir)
{
   //Nothing to sort
   if (arrayLength < 2)
      return 0;

   //Only power-of-two array lengths are supported by this implementation
   uint log2L;
   uint factorizationRemainder = factorRadix2(&log2L, arrayLength);
   assert(factorizationRemainder == 1);

   dir = (dir != 0);

   uint  blockCount = batchSize * arrayLength / SHARED_SIZE_LIMIT;
   uint threadCount = SHARED_SIZE_LIMIT / 2;

   if (arrayLength <= SHARED_SIZE_LIMIT)
   {
      assert((batchSize * arrayLength) % SHARED_SIZE_LIMIT == 0);
      bitonicSortShared<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_SrcKey, d_SrcVal, arrayLength,
dir);
   }
   else
   {
      bitonicSortShared1<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_SrcKey, d_SrcVal);

      for (uint size = 2 * SHARED_SIZE_LIMIT; size <= arrayLength; size <<= 1)
         for (unsigned stride = size / 2; stride > 0; stride >>= 1)
            if (stride >= SHARED_SIZE_LIMIT)
            {
               bitonicMergeGlobal<<<(batchSize * arrayLength) / 512, 256>>>(d_DstKey, d_DstVal, d_DstKey,
d_DstVal, arrayLength, size, stride, dir);
            }
            else
            {
               bitonicMergeShared<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_DstKey, d_DstVal,
arrayLength, size, dir);
               break;
            }
   }
   return threadCount;
}
```

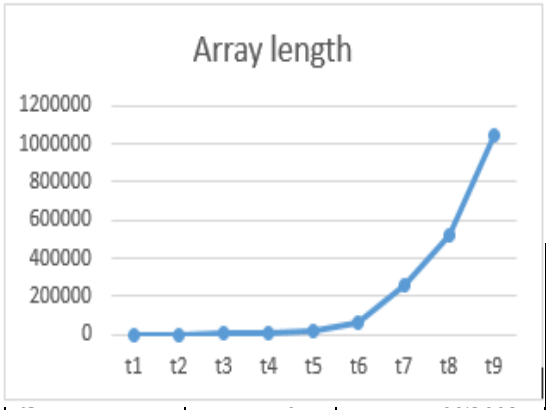
Pentru sortarea bitonic am analizat vectori de valori aleatoare pana la de lunigimi diferite. Lungimea sirului trebuie sa fie putere a lui 2 pentru ca algoritmul sa functioneze.

SHARED_SIZE_LIMIT = 1024U
uint blockCount = batchSize * arrayLength / SHARED_SIZE_LIMIT;
uint threadCount = SHARED_SIZE_LIMIT / 2;
Batch Size= N / arrayLength; //N= 1048576

Test1:
Array length 64 (16384 arrays per batch)
Average time: 29.851923 ms
Test2:
Testing array length 512 (2048 arrays per batch)...
Average time: 55.714073 ms
Test3:
Array length 1024 (1024 arrays per batch)...
Average time: 66.966805 ms
Test4:
Array length 4096 (256 arrays per batch)...
Average time: 101.646385 ms
Test5:
Testing array length 16384 (64 arrays per batch)...
Average time: 143.813797 ms
Test6:
Testing array length 65536 (16 arrays per batch)...
Average time: 193.573074 ms
Test7:
Testing array length 262144 (4 arrays per batch)...
Average time: 250.668076 ms
Test8:
Testing array length 524288 (2 arrays per batch)...
Average time: 281.882263 ms
Test9:
Testing array length 1048576 (1 arrays per batch)...
Average time: 316.694824 ms

Throughput = 3.3110 MElements/s,
Time = 0.31669 s,
Size = 1048576 elements,
NumDevsUsed = 1,
Workgroup = 512





t4	4096	101.6464
t5	16384	143.8138
t6	65536	193.5731
t7	262144	250.6681
t8	524288	281.8823
t9	1048576	316.6948

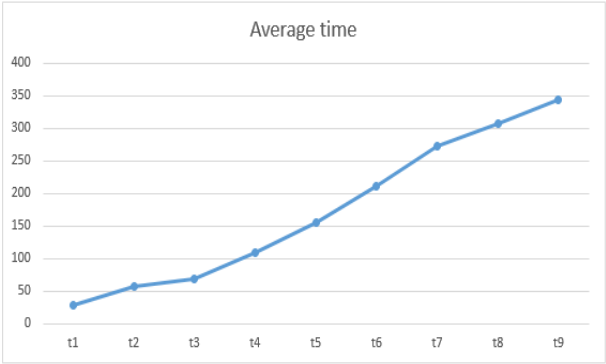
Pentru sortarea odd-even merge sort am analizat vectori de valori aleatoare pana la de lunigimi diferite. Lungimea sirului trebuie sa fie putere a lui 2 pentru ca algoritmul sa functioneze.

Test1:
Testing array length 64 (16384 arrays per batch)...

Average time: 28.540970 ms
Test2:
Testing array length 512 (2048 arrays per batch)...
Average time: 56.123219 ms
Test3:
Testing array length 1024 (1024 arrays per batch)...
Average time: 67.628807 ms
Test4:
Testing array length 4096 (256 arrays per batch)...
Average time: 108.058861 ms
Test5:
Testing array length 16384 (64 arrays per batch)...
Average time: 155.578781 ms
Test6:
Testing array length 65536 (16 arrays per batch)...
Average time: 210.808594 ms
Test7:
Testing array length 262144 (4 arrays per batch)...
Average time: 273.582214 ms
Test8:
Testing array length 524288 (2 arrays per batch)...
Average time: 307.711761 ms
Test9:
Testing array length 1048576 (1 arrays per batch)...
Average time: 343.842712 ms

Test			Array Length	Average Time
t1			64	28.54097
t2	512	56.123219		
t3	1024	67.628807		
t4	4096	108.058861		
t5	16384	155.578781		
t6	65536	210.808594		
t7	262144	273.582214		
t8	524288	307.711761		
t9	1048576	343.842712		

sortingNetworks-oddeven:
Throughput = 3.0496 MElements/s,
Time = 0.34384 s,
Size = 1048576 elements,
NumDevsUsed = 1,
Workgroup = 512



Radix sort Thrust

Thrust este o bibliotecă de șabloane C ++ pentru CUDA bazată pe Standard Template Library (STL). Thrust vă permite să implementați aplicații paralele de înaltă performanță cu un efort minim de programare printr-o interfață la nivel înalt care este pe deplin interoperabilă cu CUDA C.

Thrust oferă o colecție bogată de primitive paralele de date, cum ar fi scanarea, sortarea și reducerea, care pot fi compuse împreună pentru a implementa algoritmi complexi cu cod sursă concis și lizibil. Descriindu-vă calculul în ceea ce privește aceste abstracții la nivel înalt, oferiți Thrust libertatea de a selecta automat cea mai eficientă implementare. Ca rezultat, Thrust poate fi utilizat în prototipizarea rapidă a aplicațiilor CUDA, în care productivitatea programatorului contează cel mai mult, precum și în producție, în care robustețea și performanța absolută sunt esențiale.

Containere: host_vector, device_vector
Gestionarea memoriei: Alocare, transferuri
Selecția algoritmului: Locația este implicită

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);
// copy host data to device memory
```

```
thrust::device_vector<int> d_vec = h_vec;
// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;
// read device values from the host
int sum = d_vec[0] + d_vec[1];
// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());
// memory automatically released
```

Aceast exemplu demonstrează un algoritm foarte rapid și eficient de sortare paralelă cu radix care utilizează biblioteca Thrust. Clasa RadixSort inclusă poate sorta fie perechi cheie-valoare (cu elemente de tip float sau unsigned integer), fie numai chei.

Test1:
Sorting 1048576 32-bit unsigned int keys and values
Throughput = 2.1238 MElements/s,
Time = 0.49372 s,
Size = 1048576 elements

Test2:
Sorting 1048576 32-bit float keys and values
Throughput = 2.1063 MElements/s,
Time = 0.49783 s,
Size = 1048576 elements

Test3:
Sorting 524288 32-bit unsigned int keys and values
Throughput = 2.0623 MElements/s,
Time = 0.25422 s,
Size = 524288 elements

Test4:
Sorting 524288 32-bit float keys and values
Throughput = 2.0457 MElements/s,
Time = 0.25628 s,
Size = 524288 elements