Protokoll zum Praktikum Parallelrechner
Übung 2

Fakultät Informatik
TU Dresden

Christian Kroh
Matrikelnummer: 3755154
Studiengang: Informatik (Diplom)
Jahrgang: 2010/2011

3. Februar 2014, Dresden

Inhaltsverzeichnis

1 Parallelisierte Matrix-Multiplikation

1.1 Implementierung

Kernel Der auf dem GPU auszuführende Kernel durchläuft für ein Element (i,j) der Ziel-Matrix C, die Zeile i der Matrix A und die Spalte j der Matrix B. Das Produkt der jeweiligen Elemente wird zu dem Element (i,j) von C addiert.

Listing 1: a6.cu - Kernel

```
21 // Kernel that executes on the CUDA device
22 __global__ void multMatrixElement(uint32_t *a, uint32_t *b, uint32_t *c, int dim)
23 {
24 // int row = threadIdx.x + BLOCKDIMX * blockIdx.y;
25 // int column = blockIdx.x;
26   int row = threadIdx.y + blockIdx.y * BLOCKDIMY;
27   int column = threadIdx.x + blockIdx.x * BLOCKDIMX;
28
29   if (row<dim && column<dim){
30     uint32_t row_mult_dim, j;
31     uint32_t summe = (uint32_t) 0;
32
33     row_mult_dim = row * dim;
34
35         for ( j = 0; j < dim; j++ )
36         {
37             // C[i][j] += A[i][k] * B[k][j]
38             summe += a[ row_mult_dim + j ] * b[ j * dim + column];
39         }
40         c[ row_mult_dim + column ] = summe;
41   }
42 }
```

Speicher Speicher für die durch Arrays repräsentierten Matritzen muss auf dem normalen RAM und auch auf dem RAM der GPU allokiert werden.

Listing 2: a6.cu - Speicherallokation

```
58   // Allocate array on host
59   a_host = (uint32_t *)malloc(size);
60   b_host = (uint32_t *)malloc(size);
61   c_host = (uint32_t *)malloc(size);
62   c_test = (uint32_t *)malloc(size);
63
64   // Allocate array on device
65   CudaSafeCall(cudaMalloc((void **) &a_device, size)); CudaCheckError();
66   CudaSafeCall(cudaMalloc((void **) &b_device, size)); CudaCheckError();
67   CudaSafeCall(cudaMalloc((void **) &c_device, size)); CudaCheckError();
```

Matrizen-Initialisierung Matritzen A und B werden mir Zufallswerten initialisiert. Die Ergebnis-Matrix C wird mit Null-Werten initialisiert.

Listing 3: a6.cu - Matrizen-Initialisierung

```
70   a_host = random_mat( dim );
71   b_host = random_mat( dim );
72   c_host = zero_mat( dim );
73   c_test = zero_mat( dim );
```

Kopieren in den GPU-RAM Die initialisierten Matritzen A, B und C werden in den RAM der GPU kopiert.

Listing 4: a6.cu - Matrizen-Kopieren

```
75   CudaSafeCall(cudaMemcpy(a_device, a_host, size, cudaMemcpyHostToDevice)); CudaCheckError();
76   CudaSafeCall(cudaMemcpy(b_device, b_host, size, cudaMemcpyHostToDevice)); CudaCheckError();
77   CudaSafeCall(cudaMemcpy(c_device, c_host, size, cudaMemcpyHostToDevice)); CudaCheckError();
```

Bestimmen der Grid- und Block-Größen Definition der zu verwendenden Block- und Gridgrößen.

Listing 5: a6.cu - Grid- und Block-Größen-Definition

```
8  #define GRIDDIMX 1024
9  #define GRIDDIMY 1024
10 #define GRIDDIMZ 1
11
12 #define BLOCKDIMX 1
13 #define BLOCKDIMY 1
14 #define BLOCKDIMZ 1
```

Listing 6: a6.cu - Grid- und Block-Größen

```
80   dim3 griddim(GRIDDIMX, GRIDDIMY, GRIDDIMZ);
81   dim3 blockdim(BLOCKDIMX, BLOCKDIMY, BLOCKDIMZ);
```

Ausführen des Kernles auf Matritzen im GPU-RAM Aufruf des parallel auszufürenden Kernels.

Listing 7: a6.cu - Kernel-Aufruf

```
88   // Do calculation on device:
89   multMatrixElement <<< griddim, blockdim >>> (a_device, b_device, c_device, dim);
```

Zurückkopieren der Ergebnismatrix Die fertig berechnete Ergebnismatrix wird aus dem GPU-RAM zurückgeholt.

Listing 8: a6.cu - Ergebnis-Sichern

```
94   // Retrieve result from device and store it in host array
95   CudaSafeCall(cudaMemcpy(c_host, c_device, size, cudaMemcpyDeviceToHost)); CudaCheckError();
```

Cleanup Nicht mehr benötigte Speicherbereiche werden wieder freigegeben.

Listing 9: a6.cu - Freigabe des Speichers

```
132  // Cleanup
133  free(a_host); CudaSafeCall(cudaFree(a_device)); CudaCheckError();
134  free(b_host); CudaSafeCall(cudaFree(b_device)); CudaCheckError();
135  free(c_host); CudaSafeCall(cudaFree(c_device)); CudaCheckError();
136  free(c_test);
```

2 Zeitmessungen

Gemessen mit Nvidia Geforce GTX 460 und Nvidia Tesla K20X (Taurus).

2.1 Block/Grid-Definition Variante 1

```
24 // int row = threadIdx.x + BLOCKDIMX * blockIdx.y;
25 // int column = blockIdx.x;
```

Nvidia Geforce GTX 460: Matrix-Multiplikation Dimension 1024 x 1024

Griddim			Blockdim			Dimension	Runtime	GFLOP/s
X	Y	Z	X	Y	Z			
1024	1	1	1024	1	1	1024	1.0439s	2.06
1024	2	1	512	1	1	1024	1.0020s	2.14
1024	4	1	256	1	1	1024	0.9182s	2.34
1024	8	1	128	1	1	1024	0.8127s	2.64
1024	16	1	64	1	1	1024	0.6690s	3.21
1024	32	1	32	1	1	1024	0.4718s	4.55
1024	64	1	16	1	1	1024	0.4798s	4.48
1024	128	1	8	1	1	1024	0.6687s	3.21
1024	256	1	4	1	1	1024	1.1060s	1.94

Nvidia Tesla K20X: Matrix-Multiplikation Dimension 1024 x 1024

Griddim			Blockdim			Dimension	Runtime	GFLOP/s
X	Y	Z	X	Y	Z			
1024	1	1	1024	1	1	1024	0.2250s	9.54
1024	2	1	512	1	1	1024	0.2236s	9.60
1024	4	1	256	1	1	1024	0.2229s	9.63
1024	8	1	128	1	1	1024	0.2227s	9.64
1024	16	1	64	1	1	1024	0.2226s	9.65
1024	32	1	32	1	1	1024	0.2253s	9.53
1024	64	1	16	1	1	1024	0.2350s	9.14
1024	128	1	8	1	1	1024	0.2627s	8.18
1024	256	1	4	1	1	1024	0.3930s	5.46

2.2 Block/Grid-Definition Variante 2

```
26 int row = threadIdx.y + blockIdx.y * BLOCKDIMY;
27 int column = threadIdx.x + blockIdx.x * BLOCKDIMX;
```

Nvidia Geforce GTX 460: Matrix-Multiplikation Dimension 1024 x 1024

Griddim			Blockdim			Dimension	Runtime	GFLOP/s
X	Y	Z	X	Y	Z			
1024	1024	1	1	1	1	1024	4.2685s	0.50
512	512	1	2	2	1	1024	1.0868s	1.98
256	256	1	4	4	1	1024	0.2818s	7.62
128	128	1	8	8	1	1024	0.0830s	25.88
64	64	1	16	16	1	1024	0.0474s	45.27
32	32	1	32	32	1	1024	0.0549s	39.13

Nvidia Tesla K20X: Matrix-Multiplikation Dimension 1024 x 1024

Griddim			Blockdim			Dimension	Runtime	GFLOP/s
X	Y	Z	X	Y	Z			
1024	1024	1	1	1	1	1024	1.3625s	1.58
512	512	1	2	2	1	1024	0.3714s	5.78
256	256	1	4	4	1	1024	0.1029s	20.13
128	128	1	8	8	1	1024	0.0357s	60.14
64	64	1	16	16	1	1024	0.0241s	89.15
32	32	1	32	32	1	1024	0.0227s	94.51

3 Ergebnisse

3.1 Variante 1: $\frac{1}{BLOCKDIMX}$ Spalte in C-Matrix pro Block

```
24 // int row = threadIdx.x + BLOCKDIMX * blockIdx.y;
25 // int column = blockIdx.x;
```

Diese Auswahl der Elemente von den Matrizen A und B sorgt dafür, dass pro Block 1 Spalte der Matrix B und `BLOCKDIMX` Zeilen von A im Cache (gemeinsamen Speicher) liegen. Dadurch wird pro Block $\frac{1}{BLOCKDIMX}$ Spalte - also $\frac{1024}{BLOCKDIMX}$ Elemente - in der C-Matrix ermittelt.

3.2 Variante 2: $BLOCKDIMX * BLOCKDIMY$ Elemente in C-Matrix pro Block

```
26 int row = threadIdx.y + blockIdx.y * BLOCKDIMY;
27 int column = threadIdx.x + blockIdx.x * BLOCKDIMX;
```

Hier hingegen liegen pro Block BLOCKDIMX Spalten der B-Matrix und BLOCKDIMY Zeilen der A-Matrix im Cache (gemeinsamen Speicher) und es werden pro Block $BLOCKDIMX * BLOCKDIMY$ Elemente in der C-Matrix berechnet.

3.3 Benötigter Speicher

Variante 1 Der benötigte Speicher ergibt sich aus den in Betracht genommenen Zeilen von A pro Block.
Benötigter Speicher pro Block [Byte]: $(1 + BLOCKDIMX) * 1024 * 4$

Die besten Ergebnisse wurden für die Geräte bei folgenden Blockgrößen erreicht
Geforce GTX 460 (BLOCKDIMX = 32): $(1 + 32) * 1024 * 4 = 135,168$ KB benötigter Speicher pro Block
Tesla K20X (BLOCKDIMX = 64): $(1 + 64) * 1024 * 4 = 266,240$ KB benötigter Speicher pro Block

Variante 2 Hier ergibt sich der benötigte Speicher aus den in Betracht genommenen Spalten von B und Zeilen von A pro Block.
Benötigter Speicher pro Block [Byte]: $(BLOCKDIMX + BLOCKDIMY) * 1024 * 4$

Die besten Ergebnisse wurden für die Geräte bei folgenden Blockgrößen erreicht
Geforce GTX 460 (BLOCKDIMX = BLOCKDIMY = 16): $(16 + 16) * 1024 * 4 = 131,072$ KB benötigter Speicher pro Block
Tesla K20X (BLOCKDIMX = BLOCKDIMY = 32): $(32 + 32) * 1024 * 4 = 262,144$ KB benötigter Speicher pro Block

Dies entspricht ungefähr der Größe des Register File Space der jeweiligen Geräte. (Tesla K20x: 256KB, GTX 460: 128KB) Dieser erlaubt es pro Takt pro Thread 2 Operanden zu lesen und einen zu schreiben.

3.4 Unterschied zwischen Varianten

Der offensichtliche Unterschied zwischen Variante 1 und Variante 2 ist, dass bei voller Ausnutzung des Register File Space bei Variante 1 nur 32 (GTX 460) bzw. 64 (Tesla K20x) Elemente der Ergebnismatrix pro Block berechnet werden und durch Variante 2 16^2 (GTX 460) bzw. 32^2 (Tesla K20x). Was bei Variante 1 zu einer höheren Blockzahl und somit weniger parallel ausgeführter Threads führt bzw. mehr Daten, die aus langsameren Speicher geladen werden müssen.

3.5 Unterschied MPI-nutzende und sequentielle Berechnung

Dass eine Reihe von Threads/Prozessen, die mit denselben Daten arbeiten und nicht voneinander abhängig sind, schneller fertig werden als ein Prozess, der über die gleichen Daten das gleiche Ergebnis berechnen soll, sollte offensichtlich sein. Dennoch bleiben große Unterschiede abhängig von der jeweiligen Umsetzung.
Da parallelisierte Prozesse möglichst nicht voneinander abhängig sein sollten, ist es schlecht möglich Ergebnisse untereinander zu Teilen, wie es leicht in sequentiellen Programmen gemacht werden kann. (z.B. Optimierungen aus Übung 1)

4 Anhang

4.1 Quellcode

Listing 10: a6.cu - Quellcode

```
1  #include <stdio.h>
2  #include <cuda.h>
3  #include <stdint.h>
4  #include <time.h>
5  #include <sys/time.h>
6  #include "errorCheck.h"
7
8  #define GRIDDIMX 1024
9  #define GRIDDIMY 1024
10 #define GRIDDIMZ 1
11
12 #define BLOCKDIMX 1
13 #define BLOCKDIMY 1
14 #define BLOCKDIMZ 1
15
16
17 static inline uint32_t* random_mat( uint32_t n );
18 static inline uint32_t* zero_mat( uint32_t n );
19 static inline double gtod();
20
21 // Kernel that executes on the CUDA device
22 __global__ void multMatrixElement(uint32_t *a, uint32_t *b, uint32_t *c, int dim)
23 {
24 // int row = threadIdx.x + BLOCKDIMX * blockIdx.y;
25 // int column = blockIdx.x;
26   int row = threadIdx.y + blockIdx.y * BLOCKDIMY;
27   int column = threadIdx.x + blockIdx.x * BLOCKDIMX;
28
29   if (row<dim && column<dim){
30     uint32_t row_mult_dim, j;
31     uint32_t summe = (uint32_t) 0;
32
33     row_mult_dim = row * dim;
34
35         for ( j = 0; j < dim; j++ )
36         {
37             // C[i][j] += A[i][k] * B[k][j]
38             summe += a[ row_mult_dim + j ] * b[ j * dim + column];
39         }
40         c[ row_mult_dim + column ] = summe;
41   }
42 }
43
44 // main routine that executes on the host
45 int main(void)
46 {
47   double t_start, t_end;
48
49   uint32_t *a_device, *a_host;
50   uint32_t *b_device, *b_host;
51   uint32_t *c_device, *c_host;
52   uint32_t *c_test;
53
54   const int dim = 1024; // dimension of matrix
55   const int N = dim*dim; // Number of elements in arrays
56   size_t size = N * sizeof(uint32_t);
57
58   // Allocate array on host
59   a_host = (uint32_t *)malloc(size);
60   b_host = (uint32_t *)malloc(size);
61   c_host = (uint32_t *)malloc(size);
62   c_test = (uint32_t *)malloc(size);
```

```
63
64    // Allocate array on device
65    CudaSafeCall(cudaMalloc((void **) &a_device, size)); CudaCheckError();
66    CudaSafeCall(cudaMalloc((void **) &b_device, size)); CudaCheckError();
67    CudaSafeCall(cudaMalloc((void **) &c_device, size)); CudaCheckError();
68
69    // Initialize host array and copy it to CUDA device
70    a_host = random_mat( dim );
71    b_host = random_mat( dim );
72    c_host = zero_mat( dim );
73    c_test = zero_mat( dim );
74
75    CudaSafeCall(cudaMemcpy(a_device, a_host, size, cudaMemcpyHostToDevice)); CudaCheckError();
76    CudaSafeCall(cudaMemcpy(b_device, b_host, size, cudaMemcpyHostToDevice)); CudaCheckError();
77    CudaSafeCall(cudaMemcpy(c_device, c_host, size, cudaMemcpyHostToDevice)); CudaCheckError();
78
79    // define grid and block sizes:
80    dim3 griddim(GRIDDIMX, GRIDDIMY, GRIDDIMZ);
81    dim3 blockdim(BLOCKDIMX, BLOCKDIMY, BLOCKDIMZ);
82
83      printf("GridDim: x: %d y: %d z: %d \n",griddim.x,griddim.y,griddim.z);
84      printf("BlockDim: x: %d y: %d z: %d \n",blockdim.x,blockdim.y,blockdim.z);
85
86      t_start = gtod();
87
88    // Do calculation on device:
89    multMatrixElement <<< griddim, blockdim >>> (a_device, b_device, c_device, dim);
90    CudaCheckError();
91
92    cudaDeviceSynchronize(); t_end = gtod();
93
94    // Retrieve result from device and store it in host array
95    CudaSafeCall(cudaMemcpy(c_host, c_device, size, cudaMemcpyDeviceToHost)); CudaCheckError();
96
97      float gflops = ( ( double )2 * dim * dim * dim / 1000000000.0 ) / ( t_end - t_start );
98
99      printf("Dim: %4d runtime: %7.4fs GFLOP/s: %0.2f\n", dim, t_end - t_start, gflops );
100
101   uint32_t i_mult_dim, i_mult_dim_add_k, k_mult_dim, i, j, k;
102
103     /* Begin matrix matrix multiply kernel */
104     for ( i = 0; i < dim; i++ )
105     {
106     i_mult_dim = i * dim;
107        for ( k = 0; k < dim; k++ )
108        {
109       i_mult_dim_add_k = i_mult_dim + k;
110       k_mult_dim = k * dim;
111          for ( j = 0; j < dim; j++ )
112          {
113               // C[i][j] += A[i][k] * B[k][j]
114               c_test[ i_mult_dim + j ] += a_host[ i_mult_dim_add_k ] * b_host[ k_mult_dim +
                      j ];
115           }
116        }
117     }
118     /* End matrix matrix multiply kernel */
119
120   // Print results
121   bool testOk = true;
122   for (int i=0; i<N; i++){
123     if(c_host[i] != c_test[i]) {
124       printf("%d: %d, %d: %20d != %20d\n", i, (int) floor(i/dim), (i%dim), c_host[i],
              c_test[i]);
125       testOk = false;
126     }
127   }
```

```
128
129   if(testOk) printf("TEST PASSED\n"); else printf("TEST FAILED\n");
130
131
132   // Cleanup
133   free(a_host); CudaSafeCall(cudaFree(a_device)); CudaCheckError();
134   free(b_host); CudaSafeCall(cudaFree(b_device)); CudaCheckError();
135   free(c_host); CudaSafeCall(cudaFree(c_device)); CudaCheckError();
136   free(c_test);
137 }
138
139
140
141 /** @brief Get current timestamp in seconds.
142  *
143  * @return Returns current time stamp in seconds.
144  */
145 static inline double gtod( )
146 {
147     struct timeval act_time;
148     gettimeofday( &act_time, NULL );
149
150     return ( double )act_time.tv_sec + ( double )act_time.tv_usec / 1000000.0;
151 }
152
153 /** @brief Generate randomized matrix.
154  *
155  * @param dim Dimension for the generated matrix.
156  *
157  * @return Returns a pointer to the generated matrix on success, NULL
158  * otherwise.
159  */
160 static inline uint32_t* random_mat( uint32_t dim )
161 {
162     uint32_t *matrix = ( uint32_t* )malloc( sizeof( uint32_t ) * dim * dim );
163     if ( matrix == NULL )
164     {
165         return NULL;
166     }
167
168     srand( ( unsigned ) time( NULL ) );
169
170     for ( uint32_t i = 0; i < dim * dim; ++i)
171     {
172         matrix[ i ] = ( uint32_t )rand();
173     }
174
175   return matrix;
176 }
177
178
179 /** @brief Generate zero matrix.
180  *
181  * @param dim Dimension for the generated matrix.
182  *
183  * @return Returns a pointer to the generated matrix on success, NULL
184  * otherwise.
185  */
186 static inline uint32_t* zero_mat( uint32_t dim )
187 {
188     uint32_t* matrix = ( uint32_t* )malloc( sizeof( uint32_t ) * dim * dim );
189     if ( matrix == NULL )
190     {
191         return NULL;
192     }
193
194     for ( uint32_t i = 0; i < dim * dim; ++i)
```

```
195     {
196         matrix[ i ] = ( uint32_t )0;
197     }
198 
199   return matrix;
200 }
```