Multiprocesorski sistemi (SI4MPS, IR4MPS, MS1MPS)

Laboratorijska vežba 4 - CUDA

Cilj laboratorijske vežbe je upoznavanje studenata sa korišćenjem CUDA tehnologije na platformama Windows i Linux i izrada, prevođenje i debagovanje jednostavnog programa koji koristi CUDA tehnologiju.

Podešavanja okruženja

CUDA tehnologija je dostupna na računarima koji poseduju grafičke kartice kompanije NVIDIA, serija 8000 i novijih. Spisak grafičkih kartica koje podržavaju ovu tehnologiju se može naći na adresi http://developer.nvidia.com/cuda-gpus, a sav neophodan softver i dokumentacija se mogu preuzeti u odgovarajućim sekcijma sajta http://developer.nvidia.com/. Da bi se koristila tehnologija na lokalnom računaru, potrebno je najpre:

1. Verfikovati da sistem poseduje grafičku karticu sa CUDA mogućnostima
2. Preuzeti odgovarajući CUDA developer (razvojni) drajver

– CUDA drajveri su integrisani u standardne video drajvere, tako da se mogu koristiti i standardni NVIDIA ForceWare video drajveri. Razvojni drajveri podržavaju širi spektar CUDA hardvera.

3. Instalirati drajver
4. Preuzeti odgovarajući CUDA Toolkit

– CUDA Toolkit sadrži neophodne alate, biblioteke i izvorne datoteke potrebne za prevođenje jednog CUDA programa. U okviru toolkita se nalazi nvcc kompajler potreban za prevođenje.

5. Preuzeti i instalirati NVIDIA GPU Computing SDK (software development kit)

– SDK sadrži primere različitih CUDA programa i projekata sa svim neophodnim podešavanjima za prevođenje korišćenjem Microsoft Visual Studio.

Detaljna uputstva za instaliranje, podešavanje i prvo izvršavanje CUDA programa se mogu naći na adresi http://developer.nvidia.com/nvidia-gpu-computing-documentation pod nazivom CUDA Getting Started Guide (Windows) ili CUDA Getting Started Guide (Linux) u zavisnosti koji operativni sistem se koristi.

Alternativno, koristiti CUDA okruženje je dostupno na računaru rtidev5.etf.rs. Na tom računaru je instalirana verzija 5.5 drajvera i toolkit-a.

Podešavanja okruženja za Visual Studio i Eclipse

Za rad sa CUDA okruženjem na Windows platformi se u alatu Microsoft Visual Studio se mogu koristiti primeri solution-a iz NVIDIA GPU Computing SDK. U njima su projekti unapred podešeni za rad sa CUDA okruženjem.

Alternativno, može se koristiti NVIDIA Nsight alat (http://www.nvidia.com/object/nsight.html) koji predstavlja proširenje za Visual Studio i omogućava prevođenje, debagovanje i profajliranje CUDA koda. Okruženje zahteva korišćenje operativnog sistema Windows 7 i verzije Visual Studio 2008 ili novije. Na operativnim sistemima Linux i MacOS, dostupna je i verzija NVIDIA Nsight alata koja se instalira kao proširenje za alat Eclipse.

Takođe, za verzije Visual Studio 2005 i 2008 može se instalirati besplatan čarobnjak koji u opciju File->New->Project dodaje novu karticu CUDA unutar koje se može izabrati CUDAWinApp. Izborom ove opcije biće napravljen novi projekat unapred podešen za prevođenje CUDA aplikacija i generisan kostur za jednostavnu CUDA aplikaciju. Čarobnjak se može skinuti sa adrese http://sourceforge.net/projects/cudavswizard/.

Korišćenje CUDA na rtidev5 računaru

CUDA okruženje je dostupno na računaru rtidev5.etf.rs. Na tom računaru je instalirana verzija 5.5 drajvera i toolkit-a. Prevođenje i povezivanje programa se može izvršiti pomoću sledeće komandne linije:

nvcc -lm -o dz6z1.exe dz6z1.cu

Takođe, svi primeri iz SDK-a su dostupni u folderu /usr/local/cuda/samples. Pomoćna bibliotek cutil više nije dostupna od verzije 5.0, tako da se odgovarajući pozivi koji se nalaze u pojedinim primerima moraju eliminisati.

Funkcije za merenje vremena

Za merenje vremena u CUDA programima mogu se koristiti funkcije koje rade sa CUDA događajima. Merenje vremena funkcioniše na principu vremenskih marki (tačaka). Da bi se merilo vreme, mora se prvo napraviti odgovarajući objekat događaja.

```
// create events for timing execution
cudaEvent_t start = cudaEvent_t();
cudaEvent_t stop  = cudaEvent_t();
cudaEventCreate( &start );
cudaEventCreate( &stop  );
```

Zatim treba zabeležiti početni trenutak, pozvati jezgro i zabeležiti krajnji trenutak. Nakon toga, treba pozvati funkciju cudaEventSynchronize() da se izvrši sinhornizacija i sačeka završetak rada jezgra.

```
// record time into start event
cudaEventRecord( start, 0 ); // 0 is the default stream id

// execute kernel

// record time into stop event
cudaEventRecord( stop, 0 );

// synchronize stop event to wait for end of kernel execution on stream 0
cudaEventSynchronize( stop );
```

Potom treba izračunati vreme pozivom funkcije cudaEventElapsedTime() i po potrebi uništiti objekat događaja jednom kada više nije potreban.

```
// compute elapsed time (done by CUDA run-time)
float elapsed = 0.f;
cudaEventElapsedTime( &elapsed, start, stop );

// release events
cudaEventDestroy( start );
cudaEventDestroy( stop  );
```

Više informacija o korišćenju CUDA događaja se može pročitati u CUDA C Programming Guide dokumentaciji.

Zadatak 1

Po uputstvima iz ovog dokumenta, prevesti i pokrenuti program deviceQuery koji dostavlja korisniku podatke o CUDA uređaju. Izvorni kod programa je dat u Dodatku A ovog dokumenta.

Zadatak 2

Sastaviti program koristeći CUDA okruženje koji računa zbir dva vektora proizvoljne dužine. Pretpostaviti veličinu bloka od 256 niti. Kostur izvornog koda programa je dat u Dodatku B ovog dokumenta.

Zadatak 3

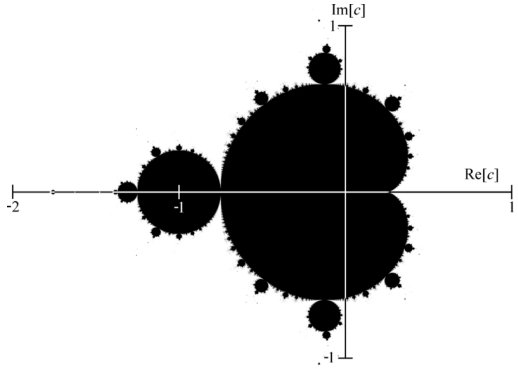
Sastaviti program koristeći CUDA okruženje koji obrće redosled elemenata u zadatom nizu proizvoljne dužine. Pretpostaviti veličinu bloka od 256 niti. Ugledati se na izvorni kod dat u Dodatku C ovog dokumenta.

Zadatak 4

Sastaviti program koristeći CUDA okruženje koji obrće redosled elemenata u zadatom nizu proizvoljne dužine. Radi bržeg izvršavanja programa, koristiti deljenu memoriju. Pretpostaviti veličinu bloka od 256 niti. Kostur izvornog koda programa je dat u Dodatku C ovog dokumenta.

Zadatak 5

Paralelizovati program koji izračunava površinu koju obuhvata Mandelbrotov skup (fraktal) korišćenjem CUDA tehnologije. Program putem komandne linije dobija broj tačaka za obradu, kao i maksimalni broj iteracija za ispitivanje uslova divergencije. Mandelbrotov skup je skup kompleksnih brojeva c za koji iteracija $z = z^2 + c$ ne divergira, kada se zada početni uslov $z = c$. Da bi se približno ustanovilo da li se neka tačka c nalazi u skupu, izvršava se konačan broj iteracija. Ukoliko je uslov $|z| > 2$ ispunjen, onda se smatra da se tačka nalazi izvan Mandelbrotovog skupa.



Mandelbrotov skup (crno) u kompleksnoj ravnini

Ne postoji teoretska formula za izračunavanje površine Mandelbrotovog skupa, već se ta vrednost može odrediti simulacijom. Mandelbrotov skup je simetričan, pa je dovoljno izračunati površinu gornje polovine. Površina se može izračunati generisanjem mreže tačaka u prozoru u kompleksnoj ravni. Prozor obuhvata tačke [-2.0, 0.5] po x osi i [0, 1.25] po y osi. Nakon generisanja, svaka tačka se iterira korišćenjem zadate formule konačan broj puta (npr. 2000 puta). Ukoliko je nakon iteriranja uslov $|z| > 2$ ispunjen, onda se smatra da se tačka nalazi izvan Mandelbrotovog skupa. Procena površine koju obuhvata skup se na kraju može izvršiti određivanjem odnosa broja tačaka koji se nalazi u skupu i ukupnog broja generisanih tačaka.

Izvorni kod programa je dat u Dodatku D ovog dokumenta. Paralelizacija se može izvršiti obradom jedne tačke svakoj niti koja se izvršava na grafičkom procesoru. na sledeći način:

1. Svaka nit će obraditi jednu tačku iz skupa, tako da je potebno ukloniti dva ugneždena **`for`** ciklusa iz sekvencijalne implementacije.
2. Blok niti će obraditi blok tačaka i izračunati broj tačaka na nivou bloka koje se nalaze van skupa operacijom redukcije na nivou bloka.
3. Potrebno je alocirati dovoljno memorije za smeštanje rezultata svakog bloka.
4. Potrebno je proračunati izvršnu konfiguraciju i pozvati jezgro.
5. Rezultate prebaciti nazad u operativnu memoriju centralnog procesora i izvršiti finalnu redukciju.

Mali CUDA podsetnik

```
cudaError_t cudaMalloc (void ** devPtr, size_t size);
cudaError_t cudaMemcpy (void * dst, const void * src, size_t count,
                        enum cudaMemcpyKind kind);
enum cudaMemcpyKind {cudaMemcpyHostToHost, cudaMemcpyHostToDevice,
                     cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice};
cudaError_t cudaFree (void* devPtr);
```

Dodatak A – deviceQuery program

```
// CUDA Device Query

#include <stdio.h>

// Print device properties
void printDevProp(cudaDeviceProp devProp) {
  printf("Major revision number:         %d\n",  devProp.major);
  printf("Minor revision number:         %d\n",  devProp.minor);
  printf("Name:                          %s\n",  devProp.name);
  printf("Total global memory:           %u\n",  devProp.totalGlobalMem);
  printf("Total shared memory per block: %u\n",  devProp.sharedMemPerBlock);
  printf("Total registers per block:     %d\n",  devProp.regsPerBlock);
  printf("Warp size:                     %d\n",  devProp.warpSize);
  printf("Maximum memory pitch:          %u\n",  devProp.memPitch);
  printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
  for (int i = 0; i < 3; ++i)
   printf("Maximum dimension %d of block:  %d\n", i, devProp.maxThreadsDim[i]);
   for (int i = 0; i < 3; ++i)
    printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
    printf("Clock rate:                    %d\n",  devProp.clockRate);
    printf("Total constant memory:         %u\n",  devProp.totalConstMem);
    printf("Texture alignment:             %u\n",  devProp.textureAlignment);
    printf("Concurrent copy and execution: %s\n",
           (devProp.deviceOverlap ? "Yes" : "No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout:      %s\n",  (
            devProp.kernelExecTimeoutEnabled ? "Yes" : "No"));
    return;
}

int main() {
    // Number of CUDA devices
    int devCount;
    cudaGetDeviceCount(&devCount);
    printf("CUDA Device Query...\n");
    printf("There are %d CUDA devices.\n", devCount);

    // Iterate through devices
    for (int i = 0; i < devCount; ++i) {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printDevProp(devProp);
    }

    printf("\nPress any key to exit...");
    char c;
    scanf("%c", &c);

    return 0;
}
```

Dodatak B – Kostur vectorAddition programa

```
// Vector Addition Kernel with timing
// without error checking

#define N 1024*1024*12

#define BLOCK_SIZE 256

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Compute vector sum C = A+B
void vecAddGold(int* A, int* B, int* C, int n){
        int i;
        for (i = 0; i < n; i++)
          C[i] = A[i] + B[i];
}

__global__ void vecAddKernel (int* devA, int* devB, int* devC, int n){

        // Use threadIdx.x, blockIdx.x and blockDim.x to compute memory
        // offset for the element to sum
        int idx = ...;
        if(...) devC[idx] = devA[idx] + devB[idx];

}

void vecAdd(int* A, int* B, int* C, int n){
        int size = n * sizeof(int);
        int *devA, *devB, *devC;

        // Memory allocations for devA, devB, devC

        // Memcopy devA, devB

        // Run ceil(N/256) blocks of 256 threads each
        vecAddKernel<<< ??? , ??? >>>(devA, devB, devC, n);

        // Memcopy devC

        // Free devA, devB, devC

}

int compareGold (int* C, int* D, int n) {
        int i;
        for (i = 0; i < n; i++) {
          if (C[i] != D[i]) {
            printf("Elements C[%d] = %d i D[%d] = %d mismatch!", i, i, C[i], D[i]);
            return 1;
          }
        }
        return 0;
}
```

```
int main (int argc, char **argv ) {
        int i, size = N *sizeof( int);
        int *A, *B, *C, *D;

        float time1, time2;

        // Allocate arrays
        A = (int*) malloc(size);
        B = (int*) malloc(size);
        C = (int*) malloc(size);
        D = (int*) malloc(size);

        // Load arrays

        srand(time(NULL));
        for (i = 0; i < N; i++) {
          A[i] = rand();
          B[i] = rand();
        }

        // CUDA

        vecAdd(A, B, C, N);

        // Sequential

        vecAddGold(A, B, D, N);

        //printf("CUDA vector addition time: %2.2f\n", time1);
        //printf("CPU vector addition time: %2.2f\n", time2 - time1);

        // Process results

        if (compareGold(C, D, N) != 0)
          printf("Test FAILED!\n");
        else
          printf("Test PASSED!\n");

        free(A);
        free(B);
        free(C);
        free(D);

        return 0;
}
```

Dodatak C – Kostur reverseArray jezgra (uz korišćenje deljene memorije)

```
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

// Part 2 of 2: implement the fast kernel using shared memory
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    extern __shared__ int s_data[];

    // Load one element per thread from device memory and store it
    // *in reversed order* into temporary shared memory
    ???

    // Block until all threads in the block have written their data to shared mem
    ???

    // write the data from shared memory in forward order,
    // but to the reversed block offset as before
    ???
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
{
    // pointer for host memory and size
    int *h_a;
    int dimA = 256 * 1024; // 256K elements (1MB total)

    // pointer for device memory
    int *d_b, *d_a;

    // define grid and block size
    int numThreadsPerBlock = 256;

    // Compute number of blocks needed based on array size and desired block size
    int numBlocks = dimA / numThreadsPerBlock;

    // Part 1 of 2: Compute the number of bytes of shared memory needed
    // This is used in the kernel invocation below
    int sharedMemSize = ???;

    // allocate host and device memory
    size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
    h_a = (int *) malloc(memSize);
    cudaMalloc( (void **) &d_a, memSize );
    cudaMalloc( (void **) &d_b, memSize );

    // Initialize input array on host
    for (int i = 0; i < dimA; ++i)
    {
        h_a[i] = i;
    }
```

```
    // Copy host array to device array
    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

    // launch kernel
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    reverseArrayBlock<<< dimGrid, dimBlock, sharedMemSize >>>( d_b, d_a );

    // block until the device has completed
    cudaThreadSynchronize();

    // check if kernel execution generated an error
    // Check for any CUDA errors
    checkCUDAError("kernel invocation");

    // device to host copy
    cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

    // Check for any CUDA errors
    checkCUDAError("memcpy");

    // verify the data returned to the host is correct
    for (int i = 0; i < dimA; i++)
    {
        assert(h_a[i] == dimA - 1 - i );
    }

    // free device memory
    cudaFree(d_a);
    cudaFree(d_b);

    // free host memory
    free(h_a);

    // If the program makes it this far, then the results are correct and
    // there are no run-time errors.  Good work!
    printf("Correct!\n");

    return 0;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}
```

Dodatak D – Mandelbrot skup

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

# define NPOINTS 1024
# define MAXITER 1000

#define ACCURACY 0.01

#define NUM_THREADS 256
#define BLOCK_DIM 16

struct complex{
  float real;
  float imag;
};

float MandlebrotSeq (int npoints, int maxiter) {
  int i, j, iter, numoutside = 0;
  float area, ztemp;
  struct complex z, c;

  for (i=0; i<npoints; i++) {
    for (j=0; j<npoints; j++) {
      c.real = -2.0+2.5*(float)(i)/(float)(npoints)+1.0e-7;
      c.imag = 1.125*(float)(j)/(float)(npoints)+1.0e-7;
      z=c;
      for (iter=0; iter<maxiter; iter++){
        ztemp=(z.real*z.real)-(z.imag*z.imag)+c.real;
        z.imag=z.real*z.imag*2+c.imag;
        z.real=ztemp;
        if ((z.real*z.real+z.imag*z.imag)>4.0e0) {
          numoutside++;
          break;
        }
      }
    }
```

```
  }

  area=2.0*2.5*1.125*(float)(npoints*npoints-numoutside)/(float)(npoints*npoints);

  printf("Numoutside sequential version: %d\n", numoutside);
  printf("Area of Mandlebrot set = %12.8f\n",area);



  return area;
}
__global__ void MandlebrotKernel (int* numOutsideSums, int npoints, int maxiter) {

  extern __shared__ int numOutside[];

  int x, y, iter;
  float creal, cimag, zreal, zimag, ztemp;

  // Calculate the points
  x = ???
  y = ???

  creal = -2.0+2.5*(float)(y)/(float)(npoints)+1.0e-7;
  cimag = 1.125*(float)(x)/(float)(npoints)+1.0e-7;

  zreal = creal;
  zimag = cimag;

  // Calculate the address in the shared memory
  numOutside[ ??? ] = 0;

  for (iter = 0; iter < maxiter; iter++) {
    ztemp = (zreal * zreal) - (zimag * zimag) + creal;
    zimag = zreal * zimag * 2 + cimag;
    zreal = ztemp;
    if ((zreal * zreal + zimag * zimag) > 4.0e0) {
      numOutside[threadIdx.y * blockDim.x + threadIdx.x] = 1;
    }
  }

  x = threadIdx.y * blockDim.x + threadIdx.x;
```

```
  __syncthreads();

  // Do reduction in shared memory
  for (int iter = ???; iter > 0; iter >>= 1) {
    if ( x < iter) {
      numOutside[x] += numOutside[x + iter];
    }
    __syncthreads();
  }

  // Calculate the address in global array
  if (x == 0) {
    numOutsideSums[ ??? ] = numOutside[0];
  }

}
float MandlebrotGPU (int npoints, int maxiter) {
  int i, numoutside = 0, grid, size;
  float area;

  int *numOutside, *numOutsideGPU;

  // Calculate the grid size
  grid = ???
  size = grid * grid * sizeof(int);

  dim3 gridDim(grid, grid);
  dim3 blockDim(BLOCK_DIM, BLOCK_DIM);

  numOutside = (int*) malloc(size);

  // Allocate the memory
  cudaMalloc( ??? );

  // Launch the kernel
  MandlebrotKernel<<< ???, ???, ???>>>(numOutsideGPU, npoints, maxiter);

  printf("Launching kernel: %s\n", cudaGetErrorString(cudaGetLastError()));
```

```
  // Memory transfer back
  cudaMemcpy( ??? );

  for (i = 0; i < grid * grid; i++) {
    numoutside += numOutside[i];
  }

  area = 2.0*2.5*1.125*(float)(npoints*npoints-numoutside)/(float)(npoints*npoints);

  cudaFree(numOutsideGPU);
  free(numOutside);

  printf("Numoutside GPU version: %d\n", numoutside);
  printf("Area of Mandlebrot set = %12.8f\n", area);

  return area;
}

int main (int argc, char* argv[]) {

  float area1, area2;

  int npoints = NPOINTS, maxiter = MAXITER;

  if (argc == 3) {
    npoints = atoi(argv[1]);
    maxiter = atoi(argv[2]);
  }

  area1 = MandlebrotSeq(npoints, maxiter);
  area2 = MandlebrotGPU(npoints, maxiter);

  if (fabs(area1 - area2) < ACCURACY) {
    printf("Test PASSED!\n");
  } else {
    printf("Test FAILED!\n");
  }

  return 0;
}
```