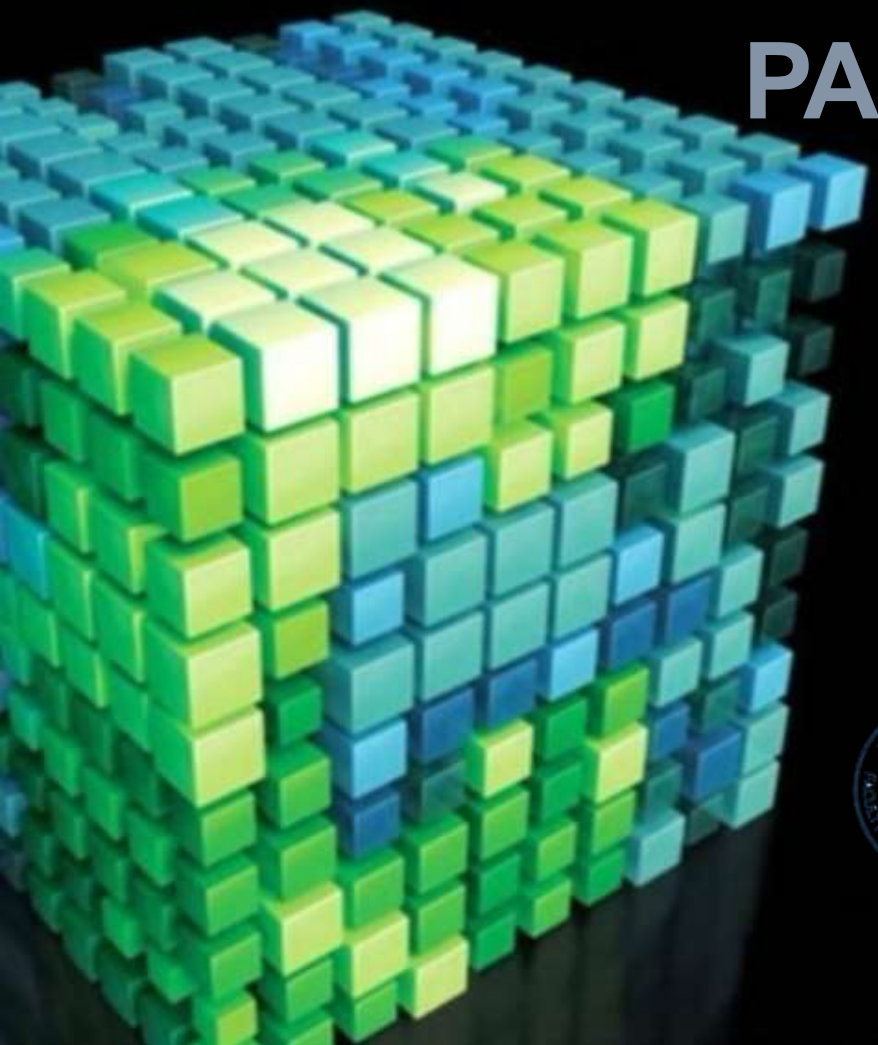


PARALELNI SISTEMI:
CUDA

MSc Aleksandra Stojnev
Prof. Dr. Natalija Stojanović

Primer: Pretraživanje niza cifara

- Koliko puta se javlja **“6”**?
- Niz od 16 elemenata, svaka nit ispituje 4 elementa, jedan blok u gridu
- Ključno:
  - organizacija koda
  - *global*, *host* i *device* funkcije

threadIdx.x = 0 ispituje in_array elemente 0, 4, 8, 12
threadIdx.x = 1 ispituje in_array elemente 1, 5, 9, 13
threadIdx.x = 2 ispituje in_array elemente 2, 6, 10, 14
threadIdx.x = 3 ispituje in_array elemente 3, 7, 11, 15

} Ciklična distribucija podataka

CUDA Pseudo-kod

GLAVNI PROGRAM:

- Inicijalizacija
  - Alokacija memorije na hostu za ulaze i izlaze podatke
  - Dodela vrednosti ulaznom nizu
- Poziv host funkcije
- Izračunavanje konačnog rezultata na osnovu rezultata pojedinačnih niti
- Prikaz rezultata

GLOBAL FUNKCIJA:

- Nit vrši obilazak svog podskupa niza elemenata
- Poziv device funkcije za poređenje sa “6”
- Izračunavanje lokalnih rezultata

HOST FUNKCIJA:

- Alociranje memorije na uređaju za kopiranje ulaza i izlaza
- Kopiranje ulaza na uređaj
- Postavljanje grida/bloka
- Poziv global funkcije
- Sinhronizacija posle kompletiranja
- Kopiranje izlaza uređaja na host

DEVICE FUNKCIJA:

- Upoređivanje tekućeg elementa sa “6”
- Vrati 1 ako su isti, u suprotnom 0

Glavni program: Preliminarni

GLAVNI PROGRAM:

- Inicijalizacija
  - Alokacija memorije na hostu za ulazne i izlazne podatke
  - Dodela vrednosti ulaznom nizu
- Poziv host funkcije
- Izračunavanje konačnog rezultata na osnovu rezultata pojedinačnih niti
- Prikaz rezultata

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  …
}
```

Glavni program: Poziv globalne funkcije

GLAVNI PROGRAM:

- Inicijalizacija (preskočeno)
  - Alokacija memorije na hostu za ulazne i izlazne podatke
  - Dodela vrednosti ulaznom nizu
- Poziv host funkcije
- Izračunavanje konačnog rezultata na osnovu rezultata pojedinačnih niti
- Prikaz rezultata

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute(
        int* in_arr, int* out_arr);

int main(int argc, char **argv)
{
  int *in_array, *out_array;

  /* inicijalizacija */
  …
  outer_compute(in_array,
  out_array);
    …
}
```

Glavni program: Izračunavanje i prikaz rezultata

GLAVNI PROGRAM:

- Inicijalizacija (preskočeno)
  - Alokacija memorije na hostu za ulazne i izlazne podatke
  - Dodela vrednosti ulaznom nizu
- Poziv host funkcije
- Izračunavanje konačnog rezultata na osnovu rezultata pojedinačnih niti
- Prikaz rezultata

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute(
        int* in_arr, int* out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  int sum = 0;
  /* inicijalizacija */ …
  outer_compute(in_array, out_array);
  for (int i=0; i<BLOCKSIZE; i++)
  {
    sum+=out_array[i];
  }
  printf (”Result = %d\n",sum);
}
```

Host funkcija: Inicijalizacija

HOST FUNKCIJA:

- Alociranje memorije na uređaju za kopiranje ulaza i izlaza
- Kopiranje ulaza na uređaj
- Postavljanje grida/bloka
- Poziv global funkcije
- Sinhronizacija posle kompletiranja
- Kopiranje izlaza uređaja na host

```
__host__ void outer_compute(
                    int* h_in_array,
                    int* h_out_array)
{
…
}
```

Host funkcija: Inicijalizacija

HOST FUNKCIJA:

- Alociranje memorije na uređaju za kopiranje ulaza i izlaza
- Kopiranje ulaza na uređaj
- Postavljanje grida/bloka
- Poziv global funkcije
- Sinhronizacija posle kompletiranja
- Kopiranje izlaza uređaja na host

```
__host__ void outer_compute(
                    int* h_in_array,
                    int* h_out_array)
{
    int *d_in_array, *d_out_array;

    cudaMalloc((void **)& d_in_array,
         SIZE* sizeof(int));

    cudaMalloc((void **)& d_out_array,
         BLOCKSIZE* sizeof(int));

    …
}
```

Host funkcija: Kopiranje podataka

HOST FUNKCIJA:

- Alociranje memorije na uređaju za kopiranje ulaza i izlaza
- Kopiranje ulaza na uređaj
- Postavljanje grida/bloka
- Poziv global funkcije
- Sinhronizacija posle kompletiranja
- Kopiranje izlaza uređaja na host

```
__host__ void outer_compute(
  int* h_in_array, int* h_out_array)
{
   int *d_in_array, *d_out_array;

   cudaMalloc((void **)& d_in_array,
        SIZE* sizeof(int));

   cudaMalloc((void **)& d_out_array,
        BLOCKSIZE* sizeof(int));

   cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int),
        cudaMemcpyHostToDevice);

   … do computation …

   cudaMemcpy(h_out_array,
        d_out_array,
        BLOCKSIZE*sizeof(int),
        cudaMemcpyDeviceToHost);

}
```

Host funkcija: Podešavanja i poziv globalne funkcije

HOST FUNKCIJA:

- Alociranje memorije na uređaju za kopiranje ulaza i izlaza
- Kopiranje ulaza na uređaj
- Postavljanje grida/bloka
- Poziv *global* funkcije
- Sinhronizacija posle kompletiranja
- Kopiranje izlaza uređaja na host

```
__host__ void
outer_compute( int* h_in_array,
               int* h_out_array)
{
   int *d_in_array, *d_out_array;
   cudaMalloc((void **)& d_in_array,
       SIZE* sizeof(int));
   cudaMalloc((void **)& d_out_array,
       BLOCKSIZE* sizeof(int));
   cudaMemcpy(d_in_array, h_in_array,
       SIZE*sizeof(int),
       cudaMemcpyHostToDevice);
   compute<<<(1,BLOCKSIZE)>>>
       (d_in_array, d_out_array);
   cudaThreadSynchronize();
   cudaMemcpy(h_out_array,
       d_out_array,
       BLOCKSIZE*sizeof(int),
       cudaMemcpyDeviceToHost);
}
```

Globalna funkcija

GLOBAL FUNKCIJA:

- Nit vrši obilazak svog podskupa niza elemenata
- Poziv device funkcije za poređenje sa “6”
- Izračunavanje lokalnih rezultata

```
__global__ void compute(int* d_in, int* d_out)
{
  d_out[threadIdx.x] = 0;

  for (int i=0; i<SIZE/BLOCKSIZE; i++)
  {
      int val = d_in[i*BLOCKSIZE + threadIdx.x];
     d_out[threadIdx.x] += compare(val, 6);
  }
}
```

Device funkcija

DEVICE FUNKCIJA:

- Upoređivanje tekućeg elementa sa “6”
- Vrati 1 ako su isti, u suprotnom 0

```
__device__ int compare(int a, int b)
{
  if (a == b)
      return 1;
  return 0;
}
```

Transponovanje matrice – Main

```
int main()
{
          double *host_InMat,*host_OutMat;
          double *device_InMat,*device_OutMat;

          int device_Count=get_DeviceCount();
          printf("\n\nNUmber of Devices : %d\n\n", device_Count);

          /* Device Selection, Device 1 */
          cudaSetDevice(0);

          int device;
          /* Current Device Detection */
          cudaGetDevice(&device);
          cudaGetDeviceProperties(&deviceProp,device);

          printf("Using device %d: %s \n", device, deviceProp.name);

*******
```

Transponovanje matrice – Main

****

```
/* event creation */
CUDA_SAFE_CALL(cudaEventCreate (&start));
CUDA_SAFE_CALL(cudaEventCreate (&stop));
/* allocating the memory for each matrix */
host_InMat = new double[size*size];
host_OutMat = new double[size*size];

if(host_InMat==NULL)
      mem_error("host_InMat","mattranspose",size,"double");

if(host_OutMat==NULL)
      mem_error("host_OutMat","mattranspose",size,"double");

/* filling the matrix with double precision */
fill_dp_matrix(host_InMat,size*size);

/* filling host_MatC with 0.0 value */
for(int i =0; i<size*size; i++)
          host_OutMat[i]=0.0;
```

****

Transponovanje matrice – Main

```
********
        /* allocating memory on GPU */
        HANDLE_ERROR(cudaMalloc( (void**)&device_InMat,size*size*sizeof(double)));
        HANDLE_ERROR(cudaMalloc( (void**)&device_OutMat, size*size*sizeof(double)));

        /* copying host matrix to device matrix */
        HANDLE_ERROR(cudaMemcpy((void*)device_InMat,
                    (void*)host_InMat,
                    size*size* sizeof(double),
                    cudaMemcpyHostToDevice ));

        HANDLE_ERROR(cudaMemcpy((void*)device_OutMat,
                    (void*)host_OutMat,
                     size*size*sizeof(double),
                     cudaMemcpyHostToDevice));

        HANDLE_ERROR (cudaEventRecord (start, 0));
        launch_kernel_MatTranspose(device_InMat,device_OutMat,size);
        //launching the kernel
        HANDLE_ERROR (cudaEventRecord (stop, 0));
        HANDLE_ERROR(cudaEventSynchronize (stop));
*******
```

Transponovanje matrice – Main

```
********
        /* computing elapsed time */
        float elapsedTime;
        HANDLE_ERROR (cudaEventElapsedTime ( &elapsedTime, start, stop));
        double Tsec = elapsedTime *1.0e-3;

        /* calling funtion for measuring Gflops */
        calculate_gflops(Tsec);

        /* printing the result on screen */
        print_on_screen("MAT TRANSPOSE",Tsec,size);

        /* retriving result from device */
      HANDLE_ERROR (cudaMemcpy((void*)host_OutMat, (void*)device_OutMat,
        size*size*sizeof(double) , cudaMemcpyDeviceToHost ));

 *******
```

Transponovanje matrice – Main

```
********
        printf("\n ------------");

        for(int i =0;i<size*size;i++)
           printf("%lf", host_OutMat[i]);

        matTransposeCheckResult(host_InMat,host_OutMat,size,size);
        /* free the device memory */
        double *array[2];
        array[0]=device_InMat;
        array[1]=device_OutMat;

        dfree(array,2);

        free(host_InMat);
        free(host_OutMat);
        cudaDeviceReset();
 }// end of main
```

Transponovanje matrice – Pomoćne funkcije

```
/* launch kernel function is called in main() */
void launch_kernel_MatTranspose(double *device_InMat, double
*device_OutMat, int size)
{
        dim3 dimBlock(BLOCKSIZE,BLOCKSIZE);
        dim3 dimGrid(size/dimBlock.x,size/dimBlock.y);

         /* checking the maximum limit of blocksize and gridsize */
        check_block_grid_dim(deviceProp,dimBlock,dimGrid);

        matTranspose<<<dimGrid,dimBlock>>>(device_InMat,
                                device_OutMat,size,BLOCKSIZE);
}
```

Transponovanje matrice – kernel

```
__global__
void matTranspose( double *device_InMat, double *device_OutMat,
                      int matRowColSize, int threadDim)
{
   int tindex = (threadDim * threadIdx.x) +  threadIdx.y;
   int maxNumThread = threadDim * threadDim;
   int pass = 0;
   int rowCount;
   int curColInd;
   while((curColInd = (tindex + maxNumThread * pass)) < matRowColSize)
   {
         for( rowCount = 0; rowCount < matRowColSize; rowCount++)
         device_OutMat[curColInd * matRowColSize + rowCount]
                           = device_InMat[rowCount* matRowColSize + curColInd];

         pass++;
   }
   __syncthreads();
}
```

Transponovanje matrice – HANDLE_ERROR

```
void HANDLE_ERROR(cudaError_t call)
{
    cudaError_t ret = call;
    switch(ret)
    {
       case cudaSuccess:
         break;
       case cudaErrorInvalidValue:
         printf("ERROR: InvalidValue:%i.\n",__LINE__);
         exit(-1);
         break;
       case cudaErrorInvalidMemcpyDirection:
         printf("ERROR:Invalid memcpy direction:%i.\n",__LINE__);
         exit(-1);
         break;
       default:
         printf("ERROR>line:%i.%d' ‘ %s\n",__LINE__,ret,cudaGetErrorString(ret));
         exit(-1);
         break;
      }
}
```

Transponovanje matrice – Pomoćne funkcije

```
int get_DeviceCount() /* Get the number of GPU devices present on the host */
{
    int count;
    cudaGetDeviceCount(&count);
    return count;
}
void fill_dp_matrix(double* vec,int size) /* Fill in the vector with double precision values */
{
    int ind;
    for(ind=0;ind<size;ind++)
           vec[ind]=drand48();
}
void mem_error(char *arrayname, char *benchmark, int len, char *type) /* Print memory error */
{
    printf("\nMemory not sufficient to allocate for array %s\n\tBenchmark : %s  \n\tMemory
           requested = %d number of %s elements\n",arrayname, benchmark, len, type);
    printf("\tAborting\n");
    exit(-1);
}
```

Transponovanje matrice – Pomoćne funkcije

```
/* Function to check grid and block dimensions */
void check_block_grid_dim(cudaDeviceProp devProp,dim3 blockDim,dim3 gridDim)
{
    if(blockDim.x >= devProp.maxThreadsDim[0]
    || blockDim.y >= devProp.maxThreadsDim[1]
    || blockDim.z >= devProp.maxThreadsDim[2])
    {
        printf("\nBlock Dimensions exceed the maximum limits:%d*%d*%d\n",
         devProp.maxThreadsDim[0],devProp.maxThreadsDim[1],devProp.maxThreadsDim[2]);
        exit(-1);
    }
    if(gridDim.x >= devProp.maxGridSize[0] || gridDim.y >= devProp.maxGridSize[1] ||
         gridDim.z >= devProp.maxGridSize[2])
    {
         printf("\nGrid Dimensions exceed the maximum limits:%d*%d*%d\n",
            devProp.maxGridSize[0],devProp.maxGridSize[1],devProp.maxGridSize[2]);
         exit(-1);
    }
}
```

Transponovanje matrice – Pomoćne funkcije

```
/* Function to calculate gflops */
double calculate_gflops(double &Tsec)
{
          double gflops=(1.0e-9 * (( 1.0 * size*size )/Tsec));
          return gflops;
}

/* prints the result on screen */
void print_on_screen(char * program_name,float tsec,int size)
{
          printf("\n---------------%s----------------\n", program_name);
          printf("\tSIZE\t TIME_SEC\t\n");
          printf("\t%d\t%f\t",size,tsec);
}

/* free memory */
void dfree(double * arr[],int len)
{
        for(int i=0;i<len;i++)
                HANDLE_ERROR(cudaFree(arr[i]));
}
```

Transponovanje matrice – Provera rezultata

```
/*************************************************************
function to check the result with sequential result
*************************************************************/
int matTransposeCheckResult(double *host_InMat,double *host_OutMat,int rows,int cols)
{
        int i,count,flag=0;
        double *temp_out;

        double  eps=EPS;
        double  relativeError=0.0;
        double  errorNorm = 0.0;

        assert((temp_out = (double *)malloc( sizeof(double) * rows*cols))!=NULL);

        int colIndex=0;

        while(colIndex != rows)
        {
                for(count = 0 ; count < cols; count++ )
                {
                        temp_out[colIndex+rows*count] =  host_InMat[count + rows   * colIndex];
                }
                colIndex++;
        }
***
```

Transponovanje matrice – Provera rezultata

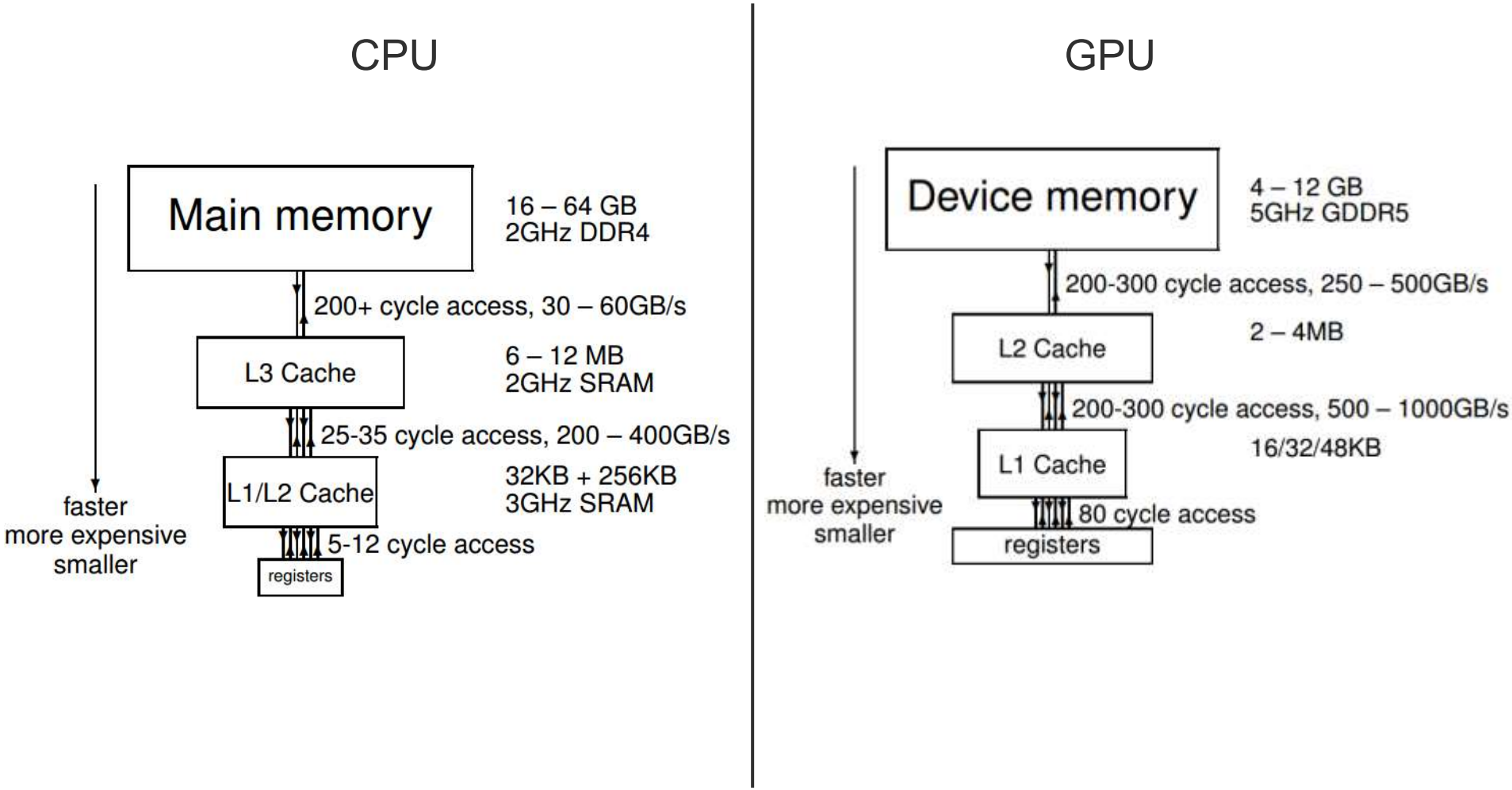
```
  for( i = 0; i < rows*cols; ++i) {
       if (fabs(temp_out[i]) > fabs(host_OutMat[i]))
            relativeError = fabs((temp_out[i] - host_OutMat[i]) / temp_out[i]);
       else
            relativeError = fabs((host_OutMat[i] - temp_out[i]) / host_OutMat[i]);

       if (relativeError > eps && relativeError != 0.0e+00)
            if(errorNorm < relativeError)
            {
                  errorNorm = relativeError;
                  flag=1;
            }
    }
    if( flag == 1)
       printf("\n Failed!\nMachine precision:%e Relative Error: \n", eps, errorNorm);
    else
        printf("\n \n\t\tResults verfication : Success\n\n");

    free(temp_out);
    return 0;
}
```

Tipovi promenljivih i memorija

Memorija

- Jedan od najvećih izazova u modernim računarskim arhitekturama
  - Brza izračunavanja gube poentu ukoliko nije moguće dovoljno brzo prebacivanje podataka
  - Kompleksne aplikacije zahtevaju mnogo memorije
  - Brze memorije su skupe
  - Uvodi se hijerarhijski dizajn memorije
- Brzina izvršavanja se oslanja na lokalnost podataka
  - Temporalna lokalnost
  - Prostorna lokalnost

Memorija


CPU
Main memory
16 – 64 GB
2GHz DDR4
200+ cycle access, 30 – 60GB/s
L3 Cache
6 – 12 MB
2GHz SRAM
25-35 cycle access, 200 – 400GB/s
L1/L2 Cache
32KB + 256KB
3GHz SRAM
5-12 cycle access
registers
faster
more expensive
smaller
GPU
Device memory
4 – 12 GB
5GHz GDDR5
200-300 cycle access, 250 – 500GB/s
L2 Cache
2 – 4MB
200-300 cycle access, 500 – 1000GB/s
L1 Cache
16/32/48KB
80 cycle access
registers
faster
more expensive
smaller

Primer 1.

```
__global__ void good_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[tid] = threadIdx.x;
}
```

- 32 niti u warpu će adresirati susedne elemente niza x
- Ako su podaci ispravno poravnati (aligned) tako da je x[0] na početku keš linije, onda će i x[0] - x[31] biti u istoj keš liniji ("*coalesced*" transfer)
- Na ovaj način dobija se savršena prostorna lokalost

Primer 2.

```
__global__ void bad_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[1000*tid] = threadIdx.x;
}
```

- U ovom slučaju, različite niti iz istog warpa pristupaju ne-susednim elementima niza x („strided“ array access)
- Svaki pristup uključuje različitu keš liniju, što se negativno odražava na performanse

Globalni nizovi

- Čuvaju se u „velikoj“ memoriji device-a
- Alocira ih host kod
- Pokazivače čuva host kod i prosleđuje ih kernelima
- Postoje sve dok ih host kod ne oslobodi
- Pošto se blokovi izvršavaju u proizvoljnom redosledu, ako jedan blok modifikuje elemenat niza, nijedan drugi blok ne treba čitati ili modifikovati taj isti element

Globalne promenljive

- Globalne promenljive se takođe mogu kreirati deklaracijom sa globalnim scope-om unutar fajla sa kernel kodom:

```
__device int reduction_lock=0;

__global__ void kernel_1(...)
{
...
}

__global__ void kernel_2(...)
{
...
}
```

Globalne promenljive

- `__device__` prefiks kaže nvcc-u da se radi o globalnoj promenljivoj na GPU, ne na CPU
- Promenljivu može da čita i modifikuje bilo koji kernel
- Životni vek joj je isti kao životni vek aplikacije
- Moguće je deklarisati i nizove fiksne dužine
- Moguće je čitanje i modifikacija od strane host koda korišćenjem specijalnih rutina `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol` ili standardne `cudaMemcpy` u kombinaciji sa `cudaGetSymbolAddress`

Konstante

- Jako slične globalnim promenljivama, osim što kerneli ne mogu da ih modifikuju
  - Definišu se globalnim scope-om unutar kernel fajla korišćenjem prefiksa `__constant__`
  - Inicijalizuje ih host kod korišćenjem `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol` ili `cudaMemcpy` u kombinaciji sa `cudaGetSymbolAddress`
- Dostupno je samo 64KB memorije za konstante, ali svaki SM ima 8-10KB keša
  - Kada sve niti čitaju istu konstantu, brzina čitanja jednaka je brzini čitanja iz registara
  - Ne zauzima registre, pa može biti od koristi za minimizaciju broja potrebnih registara

Konstante

- Vrednost konstante se postavlja u run-time-u.
- Kod često sadrži konstante čija se vrednost zna u vremenu kompajliranja:

```
#define PI 3.1415926f
a = b / (2.0f * PI);
```

- Ovakve konstante se ugrađuju u executable kod, pa ne zauzimaju registre
- `f` na kraju se koristi za jednostruku preciznost, jer je u C/C++:

single x double = double

Registri

- U okviru svakog kernela podrazumevano se individualne promenljive smeštaju u registre:

```
__global__ void lap(int I, int J, float *u1, float *u2)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int id = i + j*I;

    if (i==0 || i==I-1 || j==0 || j==J-1)
        u2[id] = u1[id];
    else
        u2[id] = 0.25f * (u1[id-1] + u1[id+1] + u1[id-I]
                                              + u1[id+I]);
}
```

Registri

- 64K 32-bit registara po SM
- Do 255 registara za svaku nit
- Do 2048 niti (najviše 1024 niti po bloku)
- Max registara za svaku nit => 256 niti
- Max broj niti => 32 registara za svaku nit
- Postoji velika razlika između “fat” i “thin” niti
- Ako je aplikaciji potrebno više registara, koristi se najpre L1 keš, a posle i memorija device-a
- Nije definisano da li konkretna promenljiva postaje niz sa jednim elementom na uređaju ili se vrednosti u registrima “čuvaju” u memoriji device-a, a kasnije se vraćaju
- U svakom slučaju, javlja se latenca usled korišćenja memorije uređaja

Lokalni nizovi

Šta se dešava ako aplikacija koristi nizove sa malim broj elemenata?

```
__global__ void lap(float *u)
{
    float ut[3];
    int tid = threadIdx.x + blockIdx.x*blockDim.x;

    for (int k=0; k<3; k++)
        ut[k] = u[tid+k*gridDim.x*blockDim.x];

    for (int k=0; k<3; k++)
       u[tid+k*gridDim.x*blockDim.x] =
       A[3*k]*ut[0]+A[3*k+1]*ut[1]+A[3*k+2]*ut[2];
}
```

Lokalni nizovi

- U jednostavnim situacijama (koje su jako česte), kompajler ove nizove konvertuje u skalare i smešta ih u registre:

```
__global__ void lap(float* u)
{
  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  float ut0 = u[tid + 0*gridDim.x*blockDim.x];
  float ut1 = u[tid + 1*gridDim.x*blockDim.x];
  float ut2 = u[tid + 2*gridDim.x*blockDim.x];

  u[tid + 0*gridDim.x*blockDim.x] = A[0]*ut0 + A[1]*ut1 + A[2]*ut2;
  u[tid + 1*gridDim.x*blockDim.x] = A[3]*ut0 + A[4]*ut1 + A[5]*ut2;
  u[tid + 2*gridDim.x*blockDim.x] = A[6]*ut0 + A[7]*ut1 + A[8]*ut2;
}
```

Lokalni nizovi

- Kod malo komplikovanijih situacija, kompajler stavlja nizove u memoriju device-a
  - Idalje se radi o lokalnom nizu jer svaka nit poseduje svoju privatnu kopiju
  - Podrazumevano se čuvaju u L1 kešu, i često se dešava da nikad ne budu premešteni u memoriju device-a
  - 48kB L1 keša se slika na 12k 32-bitnih promenljivih
    => 12 za svaku nit ako se koristi 1024 niti

Deljena memorija

- U kernelima, prefiks __shared__ deklariše podatke kao deljive između svih niti u bloku niti – svaka nit može čitati ili modifikovati vrednost:

```
__shared__ int x_dim;
__shared__ float x[128];
```

- Ovakvi podaci su neophodni za operacije koje zahtevaju komunikaciju između niti
- Korisni za re-use
- Alternativa za lokalne nizove u memoriji device-a

Deljena memorija

- Ako blok niti ima više od jednog warpa, nije definisano kada će koji warp krenuti i završiti
- Zbog toga je skoro uvek neophodna sinhronizacija niti kako bi se obezbedilo korektno korišćenje deljive memorije
- Instrukcija `__syncthreads()` postavlja “barijeru”: Nijedna nit/warp ne može nastaviti izvršenje iza barijere sve dok svi ostali nisu stigli do barijere
- Pored statički alocirane deljive memorije, moguće je i kreiranje dinamičkih nizova u deljivoj memoriji
- Ukupna veličina specificira se opcionim trećim argumentom pri pozivu kernela:

```
kernel<<<blocks, threads, shared_bytes>>>(...)
```

Read-only nizovi

- Pri radu sa konstantama, svaka nit čita istu vrednost
- U drugim slučajevima, imamo nizove čije se vrednosti elemenata ne menjaju, ali različite niti čitaju različite elemente
- Tada je korisno naglasiti kompajleru da se radi o read-only nizu:

```
const __restrict__
```

- Na hardeverskom nivou, koriste se instrukcije koje daju bolje performanse

CUDA kvalifikatori promenljivih

- Automatske promenljive bez kvalifikatora se smeštaju u registre
  - Osim velikih struktura i statičkih nizova koji se smeštaju u lokalnu memoriju
- Pokazivači mogu da pokazuju samo na objekte iz globalne memorije:
  - Alocirane na strani domaćina i prosleđene jezgru

    `__global__ void KernelFunc(float* ptr);`
  - Statički deklarisane objekte na strani uređaja

    `float* ptr = &globalVar;`
- Kvalifikator **__device__** je opcion ako su navedeni kvalifikatori **__shared__** ili **__constant__**

		Memorija	**Opseg**	**Životni vek**
__device__ __shared__	int SharedVar	Deljena	Blok	Blok
__device__	int GlobalVar	Globalna	Grid	Aplikacija
__device__ __constant__	int ConstantVar	Konstantna	Grid	Aplikacija

Memorija - hijerarhija

- Globalna memorija (GB)
- Deljena memorija (kB)
- Registri
- Lokalna memorija
- Constant memorija
- Texture memorija

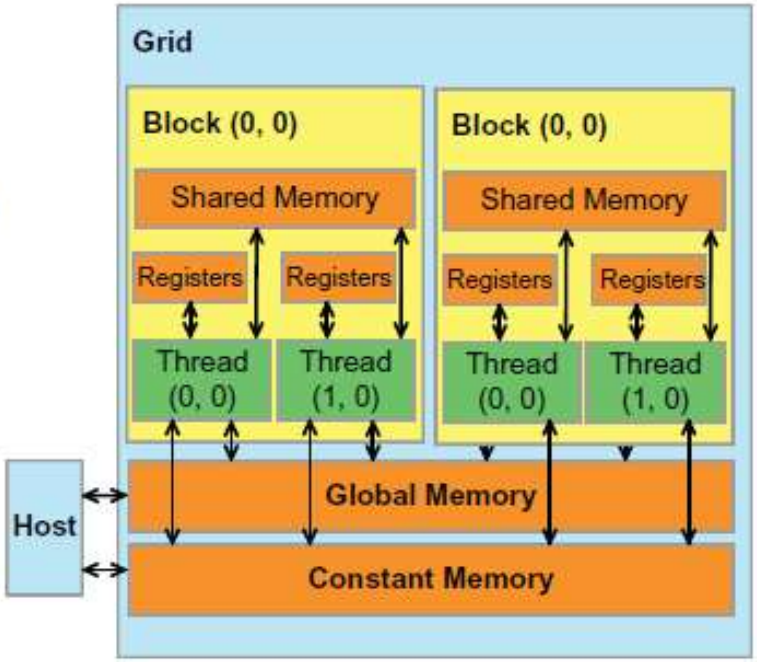
Memorija - hijerarhija

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

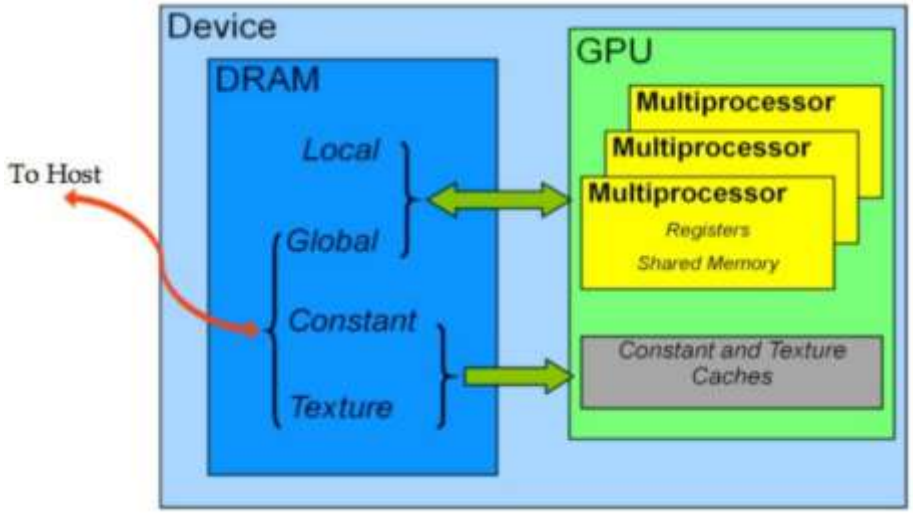
Host code can

- Transfer data to/from per grid global and constant memories



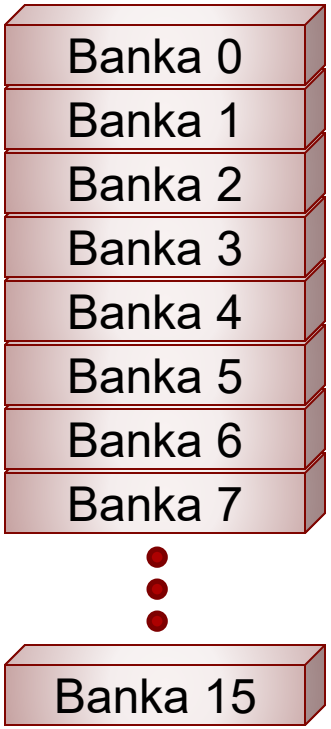
Memorija - hijerarhija

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation



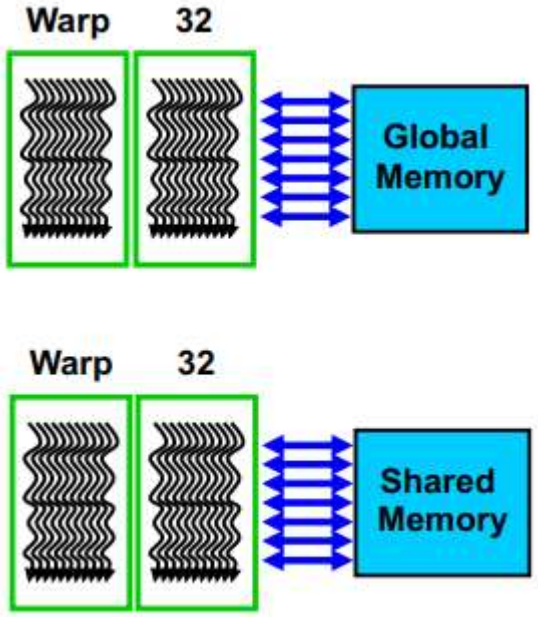
Paralelna memorijska arhitektura (1)

- Kod paralelne mašine, veliki broj niti pristupa memoriji
  - Memorija je preklopljena i podeljena u banke
    - I globalna i deljena memorija
  - Vrlo bitno za postizanje velikog propusnog opsega
- Svaka memorijska banka može da usluži jedan zahtev u jednom ciklusu
  - Celokupna memorija može simultano da usluži onoliko pristupa koliko ima memorijskih banki
- Više simultanih pristupa istoj banki dovodi do konflikta
  - Konfliktni pristupi se serijalizuju



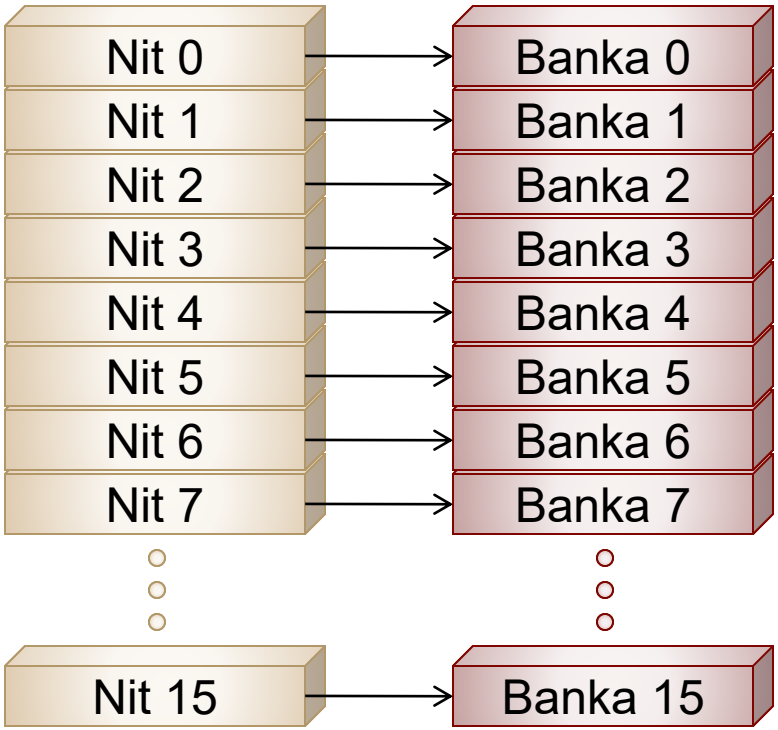
Paralelna memorijska arhitektura (2)

- Na novijim procesorima, memorija je podeljena u 32 banke
  - Uzastopne 32-bitne reči se dodeljuju uzastopnim memorijskim bankama
- Pristup memoriji na CUDA se kombinuje u transakcije
  - Najbolje performanse se dobijaju kada sve niti unutar warp-a pristupaju uzastopnim memorijskim lokacijama
    - Tada nema konflikata
  - Konflikti su mogući jedino unutar warp-a

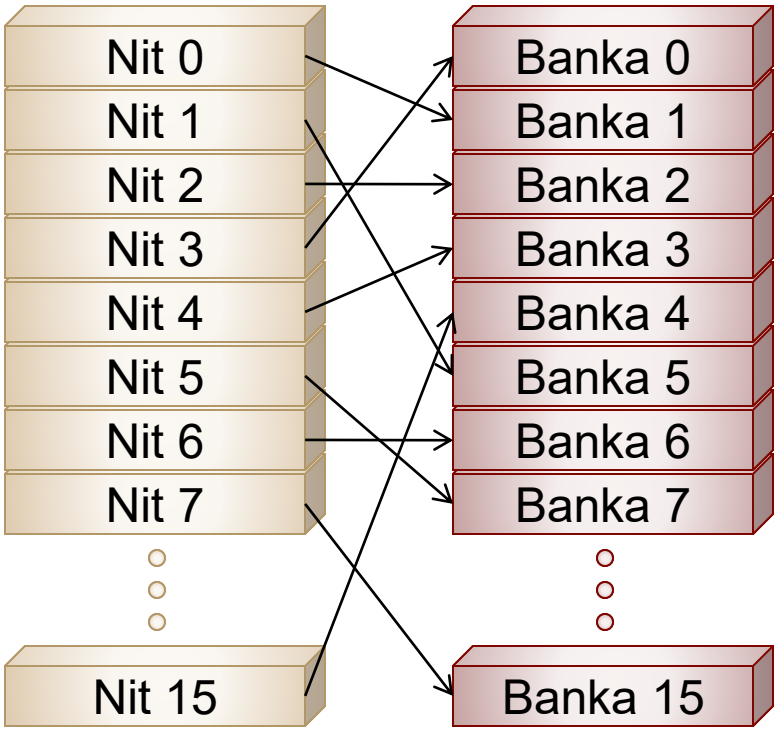


Primeri pristupa memoriji (1)

Nema konflikata
Linearno adresiranje, stride =1

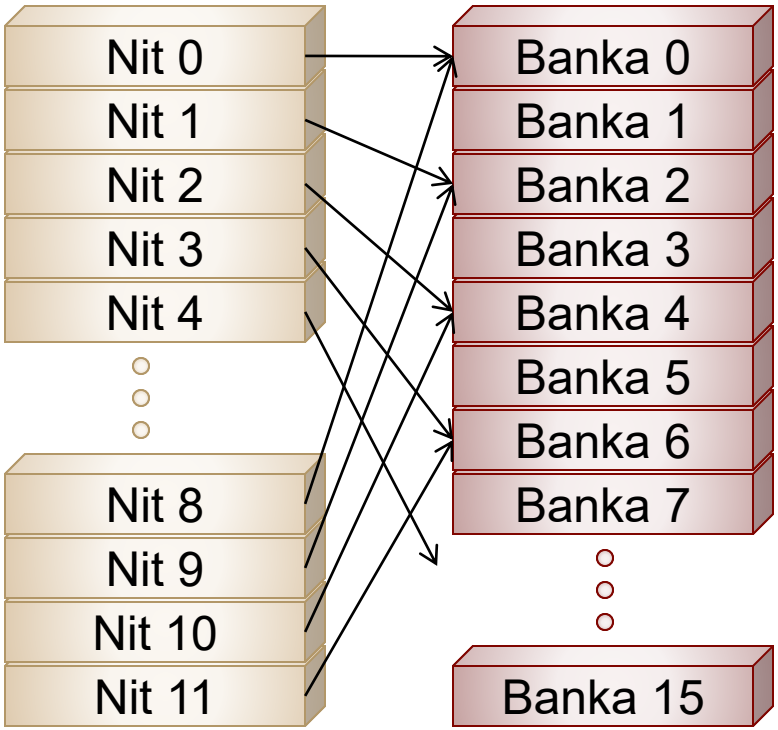


Nema konflikata
Slučajan pristup memoriji



Primeri pristupa memoriji (2)

Dvostruki konflikt
Linearno adresiranje, stride = 2



8-struki konflikt
Linearno adresiranje, stride = 8