11-CUDA

BRZ PREGLED

Šta je u stvari CUDA?

- CUDA je model programiranja i platforma za paralelno računastvo na akceleratorskim arhitekturama.
- U praksi taj model programiranja je integrisan u skup alata koje proizvodi Nvidia što uključuje osnovnu biblioteku, plus reimplementaciju nekakvih standardnih biblioteka za HPC u CUDA obliku.
  - cuBLAS
  - cuSPARSE
  - cuFFT
- Tu su takođe i originalne biblioteke posebne namene kao što je, npr. cuDNN i TensorRT koji su tako ključni za rad u veštačkoj inteligenciji.

Gde radi CUDA?

- Na Nvidia karticama
- Tačna lista je ovde:
  - https://developer.nvidia.com/cuda-gpus

Osnovne apstrakcije CUDA okruženja

- CUDA pokušava da omogući da se programira u kompleksnom okruženju akceleratorske tehnologije tako što inkorporira u sebe tri glavne apstrakcije:
  - Hijerarhija grupa niti
  - Hijerarhija deljenih memorija
  - Barijerna sinhronizacija

Hijerarhije

- Ključna ideja jeste da se problem koji se rešava podeli tako da se identifikuju
  - Grubi paralelizmi zadatka i podataka koji predstavljaju stvari koje CUDA runtime može da podeli na koliko god procesnih elemenata (SM-ova) ima na raspolaganju.
  - Sitni paralelizam podataka i niti koje se odnose na zadatke koje se moraju rešiti unutar SM-a kooperativno.

SM?

- SM je Streaming Multiprocessor, osnovna gradivna jedinica CUDA-sposobnog uređaja.
- Specifikacije SM-a koga koristimo su jako bitne i variraju između verzije.
- Termin za verziju je u CUDA svetu 'compute capacity' i mi ovde posmatramo 8.6 arhitekturu.
- Neophodno je, prilikom napredne optimizacije, kod prilagoditi koliko god je to moguće okruženju u kome se izvršava, tako da u zavisnosti od kartice koju koristite morate gledati drugu specifikaciju.
- Sve su dostupne u zvaničnom vodiču za programiranje CUDA arhitekture.

Compute Capability 8.6 SM specifikacija

- 128 FP32 jezgra za proračune
- 64 FP64 jezgra za proračune
- 64 INT32 jezgra za proračune
- 4 jezgra tenzorskog tipa (treća generacija) pomešane preciznosti
- 16 posebna jezgra za proračun transcedentalnih funkcija jednostruke preciznosti
- 4 raspoređivača spleta (warp, više o tome kasnije)
- Keš za konstantnu memoriju
- 128kb unificirane keš memorije iz koje se alocira deljena memorija i L1 keš. Legalne vrednosti deljene memorije su 0,8,16,32,64,ili 100 KB.

Float i double

- Verovatno vam je već rečeno da na bilo kom modernom CPU-u, upotreba double je jednako dobra kao i upotreba float-a: čak je i na nekim mestima brža pošto se float vrednost samo proširuje da bi stala u FPU registre koji su sami po sebi double.
- Ovo nije slučaj u GPU programiranju.
- Float proračuni su tačno dvaput 'jefintiji' nego double proračuni, i dosta internih struktura (npr. modularizacija deljene memorije o čemu više kasnije) su napravljeni sa idejom da se prenose float vrednosti.
- Kao rezultat, gde god da ima smisla koristiti float, on se treba koristiti budući da se dupla preciznost sada 'plaća.'

Kernel

- Hijerarhija se eksponira kroz različite konstrukte u jeziku koji se koristi
- Mi ovde koristimo C++ koji je 'maternji' za CUDA okruženje
- U njemu među najosnovnijim konstruktima se nalazi paralelna funkcija odn. 'kernel.'
- Ako proglasimo da je neka funkcija kernel ono što želimo da kažemo kroz to CUDA okruženje jeste da želimo da možemo da datu funkciju pozovemo na više niti paralelno.
- Specifikacija na koliko niti se radi se određuje prilikom poziva, a svaka instanca funkcije tokom izvršavanja zna koja je nit odgovorno kroz ugrađenu promenljivu threadIdx

Kernel

```
__global__ void VecAdd(float* A, float* B, float* C){
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main(){
    //Definisati i alocirati A, B, i C ovde.
    VecAdd<<<1, N>>>(A, B, C);
}
```

N.B. Primeri preuzeti i adaptirani iz zvanične Nvidia dokumentacije:
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

threadIdx

- threadIdx je promenljiva koja će biti dostupna u telu kernel-a i identifikuje aktuelnu nit.
- U CUDA okruženju to je trokomponentni vektor sa X, Y, i Z vrednostima da bi olakšao deljenje stvari koje se lakše mapiraju na 2D ili 3D ćelije, na primer.
- Ovde prilikom invokacije specificiramo 1, N što znači da želimo 1D varijantu gde je samo .x element bitan, i to hoćemo jedan blok niti, koji je 1D i ima samo jednu dimenziju, N.
- Za višedimenzionalniju vrednost treba nam da threadsPerBlock bude ne jedan broj nego nešto sa više dimenzija: koristi se ugrađeni CUDA tip 'dim3'

threadIdx

```
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N]){
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main(){
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- Ovde imamo jedan blok koji se sastoji od blokova sa NxN elemenata.
- Valja voditi računa da se ovo razlikuje od 1D niza sa N^2 elemenata.
- Generalno je dobra ideja koristiti onoliko-dimenzionalne blokove koliko su podaci nad kojima se operiše

Veza između threadIdx i identifikatora niti

- Za 1D blok Dx
  - ID(x) = x
- Za 2D blok Dx * Dy
  - ID(x,y) = x + y*Dy
- Za 3D blok Dx * Dy * Dz
  - ID(x,y,z) = x + y*Dx + z*Dx*Dy

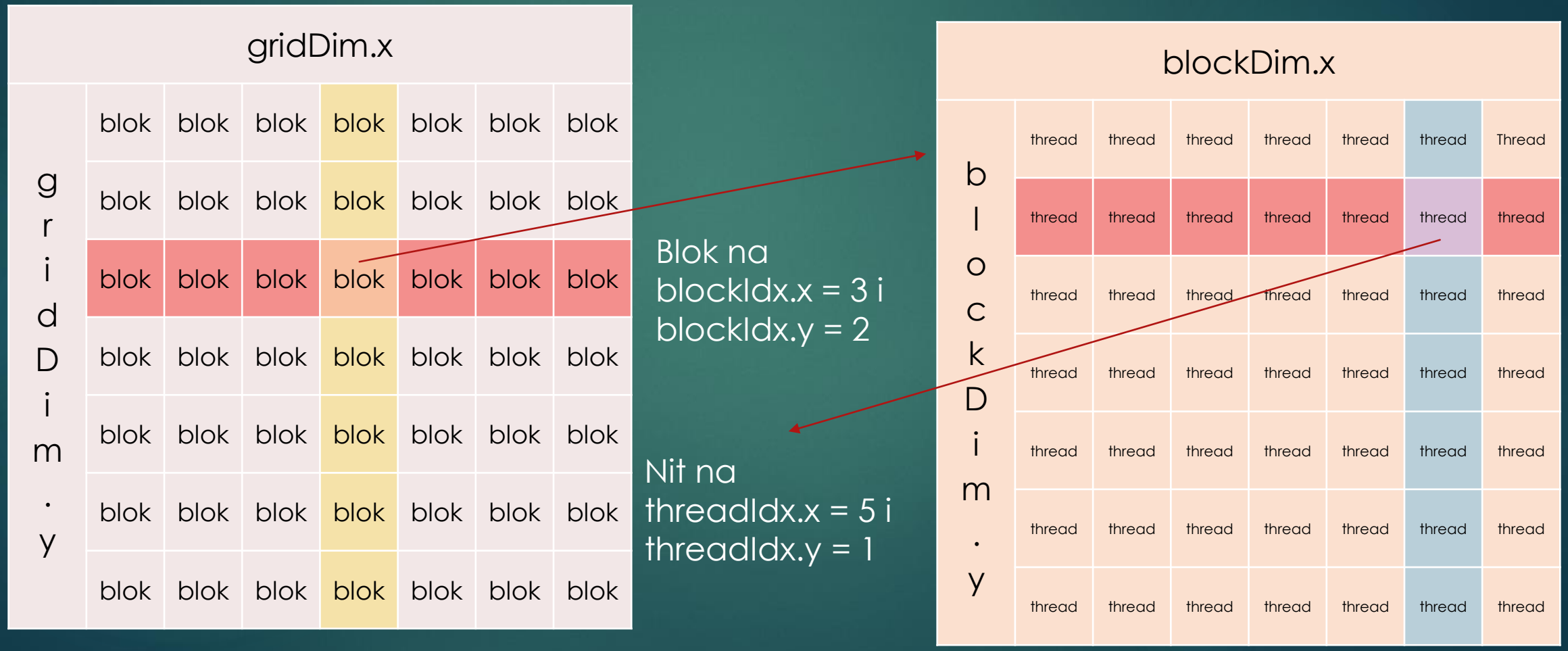
Limiti identifikatora niti

- Na modernim GPU uređajima, ograničenje je 1024 niti po bloku.
- Sa ovim se bori tako što se prave dodatni blokovi
- Ti dodatni blokovi, baš kao i niti unutar bloka, mogu da budu 1D, 2D, ili 3D na sličan način kao i niti unutar bloka
- Nema tvrde granice koliko može biti blokova, ali je pravilo da *između* blokova nema komunikacije, i da se blokovi izvršavaju u makojem redosledu: potpuno pod kontrolom runtime-a
- Takođe, svaki blok mora biti sasvim jednak.
- Dimenzije blokova određuje prvi parametar prilikom izvršavanja i to može biti dim3 ili int i funkcioniše apsolutno identično kao i u slučaju dimenzionisanja niti.

blockIdx i blockDim

- Dimenzije bloka su dostupne iz unutrašnjosti thread-a kroz dve promenljive, obe od kojih su trokomponentni vektori tipa dim3.
- blockIdx nam kaže u kom smo bloku, baš kao što je to govorio threadIdx za to u kojoj smo niti unutar bloka
- blockDim nam kaže kolike su dimenzije aktuelnog bloka po sve tri ose.
- Na raspolaganju je i gridDim ugrađena promenljiva koja nam kaže koliko ima ukupno blokova i kako su raspoređeni (struktura za raspoređivanje blokova se u CUDA okruženju zove 'grid')

Indeksi unutar CUDA okruženja (2D slučaj)



Primena blokova

```
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    //...
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    //...
}
```

Splet niti

- Na hardverskom nivou se još pojavljuje i koncept *spleta* (eng. Warp)
- Ovo nije apstrakcija namenjena programeru, koliko je podela koju iznuđuje hardver.
- Splet niti je grupa od 32 niti (tu grupaciju određuje hardver/runtime) koje su takve da izvršavaju *istu instrukciju u istom trenutku*.
- Ovo je deo tkzv. SIMT modela (single instruction, multiple threads).
- Valja primetiti da niti mogu imati grananja.
- U tom slučaju, splet raspoređivač će da izvrši *svaku granu* samo što će niti za koje data nit ne važi da isključi.
- Ovo znači da nam je od velike koristi ako su niti u 32-nitnim blokovima takve da imaju što manje grananja.

Saradnja unutar bloka

- Unutar jednog bloka, niti mogu da sarađuju.
- Sarađivanje unutar bloka između niti se bazira na dva koncepta: *deljena memorija* i *sinhronizacija izvršavanja*.
- Sinhronizacija izvršavanja se na primitivnom nivou može obavljati kroz ugrađenu funkciju __syncthreads() koja, kada se pozove, postane sinhronizaciona barijera: sve niti u jednom bloku stanu dok sve ne dostignu tu barijeru.
- Postoji i sofisticiraniji model za situacije gde je potrebna granularnija sinhronizacija kroz nešto što se zove 'API za grupe kooperacije' odn. 'Cooperative Groups API'
- Deljena memorija se samo može razumeti kroz njeno mesto u hijerarhiji memorije.

Memorijska hijerarhija

- U klasičnom modelu programiranja, sva memorija se tretira isto.
- U praksi, mi znamo da neke stvari idu u različite keš memorije, ali se ovo generalno govoreći ne podešava rukom.
- U CUDA svetu, memorijom se mora upravljati mnogo pažljivije
- Prva podela je između memorije host računara, tj. Radne memorije računara na kome se CUDA program izvršava i memorije GPU-a (tehnički termin je 'device' memorija)
- Unutar memorije GPU-a postoji kompleksna hijerarhija memorije bazirana na tome kome memorija 'pripada,' kako joj se pristupa i pod kojim uslovima, i da li je perzistentna.

Memorijska hijerarhija

- Hardverski govoreći, memorija GPU-a se deli na on-chip i off-chip memoriju.
- Off-chip memorija je memorija samog GPU uređaja i pristup njoj je komparativno spor i ima visoko kašnjenje.
- On-chip memorija je deo samog silikona grafičkog procesora, ima je malo, i jako je brza. Ta memorija je slična kešu, ali se može njom upravljati i potpuno direktno.

Memorijska hijerarhija

- Logički, memorija kojoj se pristupa u CUDA modelu je
  - Thread memorija koja je isključiva za nit i fizički je deo on-chip memorije.
  - Deljena (shared) memorija koja je isključiva za blok niti i pristup se može obaviti jako brzo.
  - Globalna memorija je dostupna svima, perzistentna je, ali je off-chip i spora.
  - Lokalna memorija je dostupna samo jednoj niti, ali je to samo apstrakcija nad globalnom memorijom, te je jednako spora.
  - Konstatna memorija se nalazi off-chip ali ima za sebe specifičan keš.
  - Memorija tekstura/površine je off-chip ali ima specifičan keš i specifično pravilo keširanja i učitavanja koje je optimizovano za operacije nad teksturama.

Strategije brzog pristupa

- Fundamentalna 'tajna' dobrog CUDA programiranja je dobra ideja o tome gde su podaci koji nam trebaju u bilo kom trenutku.
- Naš cilj:
  - Minimizirati broj transfera između host i device memorije što je više moguće: to je prvo usko grlo.
  - Minimizirati broj pristupa off-chip memoriji.
- Tipičan obrazac ponašanja je 'staging' pristup u kome se operacije dele u faze u kojoj svaka ima neku oblast memorije koja je zanima.
  - Ovo se radi posebno za blokove memorije i posebno za transfer host-device, mada ako je to moguće, ovaj drugi transfer se onda uradi samo jednom, na početku.

Staging za deljenu memoriju

- Tipično ponašanje jeste da:
  - Svaki blok učita u deljenu memoriju onaj deo iz globalne memorije koji je potreban za datu fazu.
  - Tipično svaki element kopira po jedna nit u paraleli.
  - Onda se unutar kernela aktivira barijerna sinhronizacija da bi bili time sigurni da je svaka nit završila svoje kopiranje, jer onda (zbog deljene memorije) svi mogu da pristupaju svim prekopiranim materijalima.
- Ovako se dramatično umanjuje cena kašnjenja po pristupu globalnoj memoriji, budući da se svakoj neophodnoj lokaciji pristupi jednom (prilikom kopiranja u deljenu memoriju) umesto svako put kada je potrebnom nekoj niti.

Napredne strategije optimiziacije pristupa deljenoj memoriji

- Deljena memorija ima specijalizovan način pristupa koji dramatično povećava protok ka memoriji pod vrlo specifičnim uslovima.
- Deljena memorija je podeljena u module (bank) koji su tako hardverski organizovani da sukcesivne 32-bitne reči odgovaraju sukcesivnim modulima, svaki od kojih ima protok od 32 bita po ciklusu sata.
- Ovo znači da ako bi od 32 niti svaka zatražila po 32-bitnu vrednost jednu za drugom, ovo bi se učitalo u jednom ciklusu budući da bi svaka nit pričala sa drugim modulom.
- Ako neke niti traže istu 32-bitnu reč, onda je takođe brzina očuvana.
- Problem nastane ako bude *konflikta modula* (eng. Bank conflict)

Napredne strategije optimiziacije pristupa deljenoj memoriji

- Konflikt modula nastaje kada dve niti zatraže u isto vreme reči koje se mapiraju na isti modul: ti pristupi se onda moraju serijalizovati.
- Ovo smanjuje povećanje u protoku koje inače nudi ovaj mehanizam podele memorije.
- Kao rezultat jako je bitno poravnati podatke u deljenoj memoriji tako da se ovi konflikti minimizuju.
- Ponekad ovo prirodno sledi ako su podaci ravnomerno distribuirani, ali postoje situacije gde memoriju treba reorganizovati da se održi konstantan stride između pristupa.

Efikasan pristup globalnoj memorji

- Globalna memorija (kada joj se mora pristupati) ima svoja ograničenja: svaki pristup toj memorji mora biti obavljen preko memorijskih transakcija koje su takve da se prenosi 32, 64, ili 128 bajta istovremeno.
- Memorijske transakcije moraju imati tkzv. *prirodno poravnanje*.
- To znači da njihova bazna adresa mora biti umnožak odgovarajuće veličine (32, 64, ili 128 bajta).
- Kada splet niti treba da pristupi memoriji, svi ti pristupi se kombinuju (tehnički termin je coalesce) u ove transakcije.
- Stoga efikasan pristup globalnoj memoriji pokušava da tako rasporedi pristupe globalnoj memorji da se mogu kombinovati u što manje transakcija

Linearan memorijski model

- Budući koliko je ovo sve zahtevno, neophodno je olakšati pristup memoriji koliko god je to moguće.
- Kao rezultat, novije verzije GPU-ova podržavaju linearan sistem pristupa memorji gde se apsolutno sva memorija dostupna i na host-u i na device-u tretira kao jedan, unificiran, džinovski memorijski prostor.
- To znači da pokazivači rade bez obzira gde su (tj. njihov sadržaj sam po sebi je dovoljan da nam kaže o kojoj memoriji je reč)
- Ova funkcionalnost je samo dostupna kroz CUDA runtime i odgovarajuće alate

CUDA alati

- Sva ova apstrakcija samo može da se stavi u praksu kroz programiranje koristeći odgovarajuće ekstenzije na neki programski jezik.
- CUDA se trudi da C++ modifikuje što je to manje moguće
- Uprkos tome, ima izmena i kao rezultat kod se mora kompajlirati posebnim kompajlerom koji izrađuje Nvidia, nvcc.
- Ovaj kompajler je odgovoran i za kod koji se izvršava na CPU-u, i tu uglavnom vodi računa o tome kako će se ekstenzije koda (kao što je recimo sintaksa za pozivanje kernela) pretvoriti u odgovarajuće pozive CUDA runtime-a.
- Što se GPU koda tiče, tu je nvcc više odgovoran i kompajlira C++ kod u posebnu binarnu formu, PTX koju kasnije drajver može da JIT-uje u formu specijalizovanu za dati specifični GPU.

Praktičan primer rukovanja sa memorijom

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) //Objasnimo kasnije
        C[i] = A[i] + B[i];
}

int main()
{
    int N = ...; //Ovo dobijemo kao parametar ili od korinsika ili...l
    size_t size = N * sizeof(float);

    //Alokacija u host memoriji
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Ovde bi ove vektore incijializovali iz nekog fajla ili...
```

Praktičan primer rukovanja sa memorijom

```
// Allokacija na GPU
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

//Ovo je komunikacija izmedju RAM i GPU-a
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

int threadsPerBlock = 256; //Ovo je standard
int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
//Ovo gore je kompatibilno sa tim da N nije deljiv sa 256, ali zbog toga nam treba
//onaj if u kernel kodu
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

Praktičan primer rukovanja sa memorijom

```
    //Vratimo rezultat na kraju
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    //Upotreba rezultata ovde

    // Oslobadjanje memorije na GPU
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);
}
```

Alokacija memorije za višedimenzionalne nizove

- Memorija se mora alocirati za rad sa 2D i 3D nizovima na poseban način koji vodi računa da su podaci poravnani za pristup.
- To poravnanje može da bude prilično komplikovano ali, srećom, CUDA pruža alate koji to čine relativno bezbolnim.
- cudaMallocPitch je za 2D nizove i ima adekvatnu cudaMemcpy2D fukciju.
- cudaMalloc3D je za 3D nizove i ima adekvatnu cudaMemcpy3D funkciju

Poravnan 2D niz primer

```
// CPU deo
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// GPU deo
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Pitch?

- Ovde su ubačeni dodatni podaci u svaki red, tako da pokazivač na početak reda mora da 'preskoči' onoliko bajtova koliko je tekući red puta puta pitch (fizička širina reda), a ne puta width (logička širina).
- Umesto da pitch računamo mi, alokaciona funkcija to izračuna umesto nas i da nam ga kao izlazni parametar.
- Da bi radilo sa bilo kakvim tipovima koje želimo, veličinu reda ova funkcija očekuje u bajtovima.

Poravnan 3D niz primer

```
// CPU kod
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                    height, depth);
cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);
```

Poravnan 3D niz primer

```
// GPU kod
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr,
                         int width, int height, int depth)
{
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
            }
        }
    }
}
```

Primer primene deljene memorije

- Sledi primer (preuzet kao i većina koda ovde iz zvanične dokumentacije) primene deljene memorije radi optimiziacije pristupa.
- Prvi deo primera je implementacija algoritma množenja matrice na GPU-u koji koristi 'naivan' pristup memoriji.
- Drugi deo primera je isti algoritam, ali sa primenom deljene memorije da se algoritam ubrza.

Naivan pristup memoriji

```
// Matrica se u memoriju stavi tako da idu prvo redovi pa onda kolone u njima. Formula je:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Blok je 16x16, opet, standard
#define BLOCK_SIZE 16

__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// CPU deo
// N.B. matrica je takva uvek da je umnožak veličine bloka
void MatMul(const Matrix A, const Matrix B, Matrix C){
    // Učitavanje
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size); //Ovo bi mogli da zakomplikujemo
    //tako da koristimo poravnanje kroz pitch kao ranije
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
```

Naivan pristup memoriji

```
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);//Blok je 16x16
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y); //Imamo onoliko blokova u grid-u
    //koliko treba za matricu
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Naivan pristup memoriji

```
// GPU deo
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C){
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

Šta je ovde problem?

- Svaka nit pristupi svim vrednostima matrice koje su joj potrebne.
- Tu prirodno dolazi do replikacije pristupa, ali po nepredvidivom rasporedu što znači da konsolidacija pristupa ne radi ni približno dobro kako bi trebala.
- Kao rezultat metod koji može da se koristi jeste da se adaptira nešto slično Kanonovom algoritmu gde se bloku da pod-matrica matrica C za proračun i de se proračun podeli u faze (kojih ima onoliko kolika je jedna dimenzija gird-a) u kojoj se u toj fazi dobave pod-matrice matrice A i B koje su neophodne za tu fazu.
- Te matrice se dobave jednom i smeste se u deljenu memoriju gde ih sve niti bloka koriste. Ubrzanje je dramatično.

Optimizovan pristup memoriji

```
// Ista matrica kao ranije, ali sada imamo i 'stride' parametar koji poravnava memoriju, kao što smo pričali
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Primetite upotrebu dezignatora __device__
// __global__ se izvršava na uređaju a zove sa host-a
// __device__ sa druge strane se izvršava na uređaju i zove samo iz uređaja
// Zbog efikasnoti poziva moramo da koristimo ovaj C-oliki pristup ovde umesto nešto C++-oliko što bi možda koristilo klasu
__device__ float GetElement(const Matrix A, int row, int col){
    return A.elements[row * A.stride + col];
}

__device__ void SetElement(Matrix A, int row, int col,
                           float value){
    A.elements[row * A.stride + col] = value;
}
```

Optimizovan pristup memoriji

```
// Ovo preuzima za neku matricu A, pod-matricu koja ima dimenzije veličina_bloka (BS) puta veličina bloka
// dakle 16x16, efektivno, i čiji je gornji levi element COL*BS desno i ROW*BS dole od početka matrice.
// Odnosno, ako zamislimo da je matrica A 'popločana' pod-matricama BSxBS, onda ovo dobavlja matricu koja je col matrica udesno
// i row matrica dole
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                + BLOCK_SIZE * col];
    return Asub;
}

#define BLOCK_SIZE 16

__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

Optimizovan pristup memoriji

```
// CPU deo
// Kao i ranije, smatramo da je matrica umnožak BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C){
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);

    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);

    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

Optimizovan pristup memoriji

```
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C){
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    //Akumulator proračunate vrednosti za matricu C, lokalan za individualnu nit
    float Cvalue = 0;

    int row = threadIdx.y;
    int col = threadIdx.x;
```

Optimizovan pristup memoriji

```
//Petlja kroz sve podmatrice A i B koje nam trebaju
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Sa __shared__ određujemo da je ova memorija namenjena svim nitima u bloku
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    //Svaka nit učita prvi element koji joj treba
    //u agregatu, učitaće se cele podmatrice
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    //Barijerna sinhronizacija koja se stara da je cela podmatrica učitana
    __syncthreads();
```

Optimizovan pristup memoriji

```
        // Klasično množenje matrice za ovaj element
        // Zato što smo sinhronizovani, znamo da kad nam zatrebaju drugi elementi da će biti tu
        // zato što je svaki element koji nama može trebati 'prvi' element za __neku__ od niti u ovom
        // bloku.
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        //Nova barijerna sinhronizacija, koja se postara da je proračun gotov za svaku nit da se ne bi
        //desilo da neka  nit 'prednjači' i krene da učitava nove podatke preko podataka koje trebaju
        //nekoj niti koja kasni.
        __syncthreads();
    }

    SetElement(Csub, row, col, Cvalue);
}
```