Paralelné programovanie CUDA

Bc. št. prog. Informatika - 2010/2011

Ing. Michal Čerňanský, PhD.

Fakulta informatiky a informačných technológií,

STU Bratislava

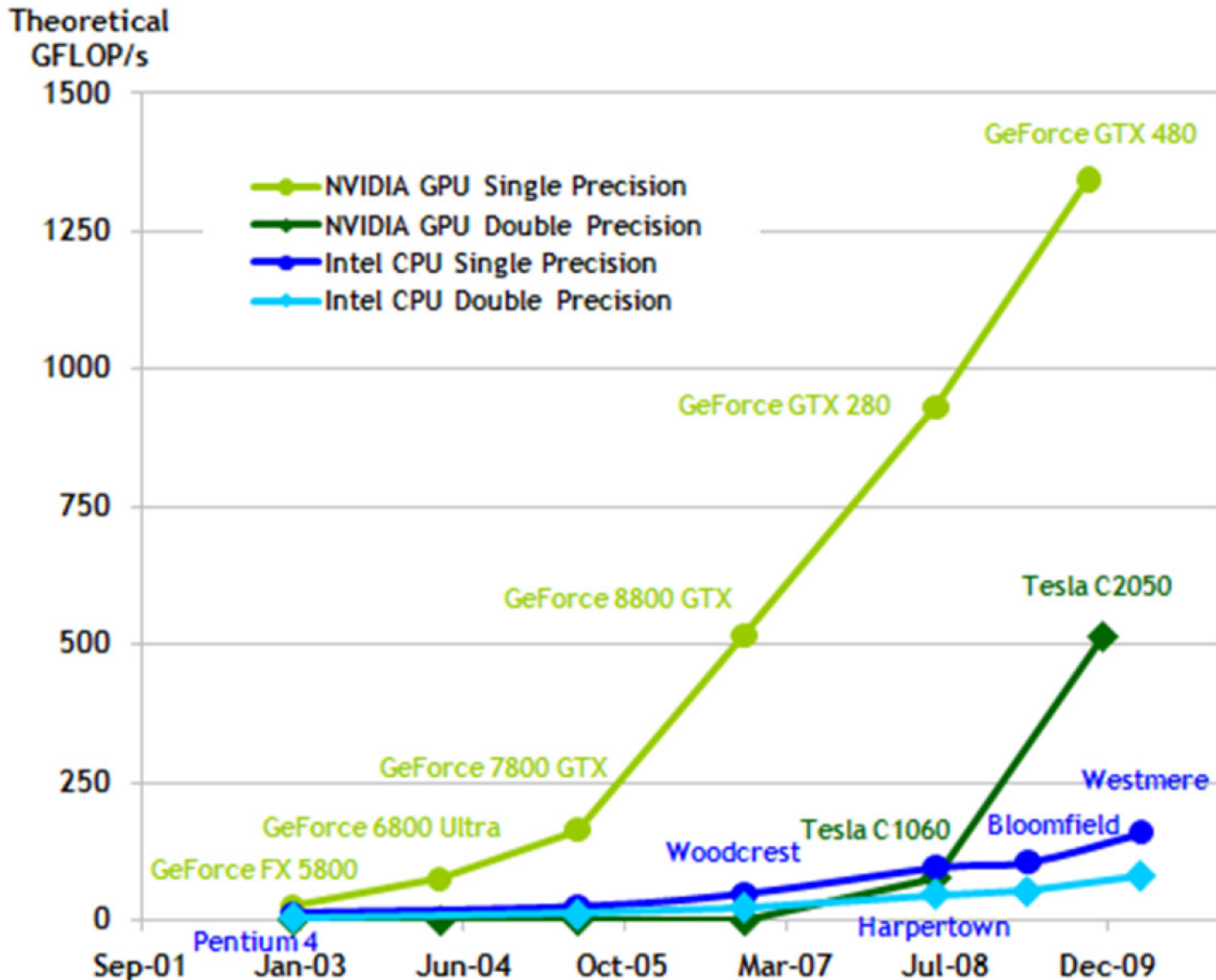
Prehľad tém

- Grafické procesory
- Realizácia všeobecných výpočtov na GPU
- Súčasné architektúry grafických procesorov

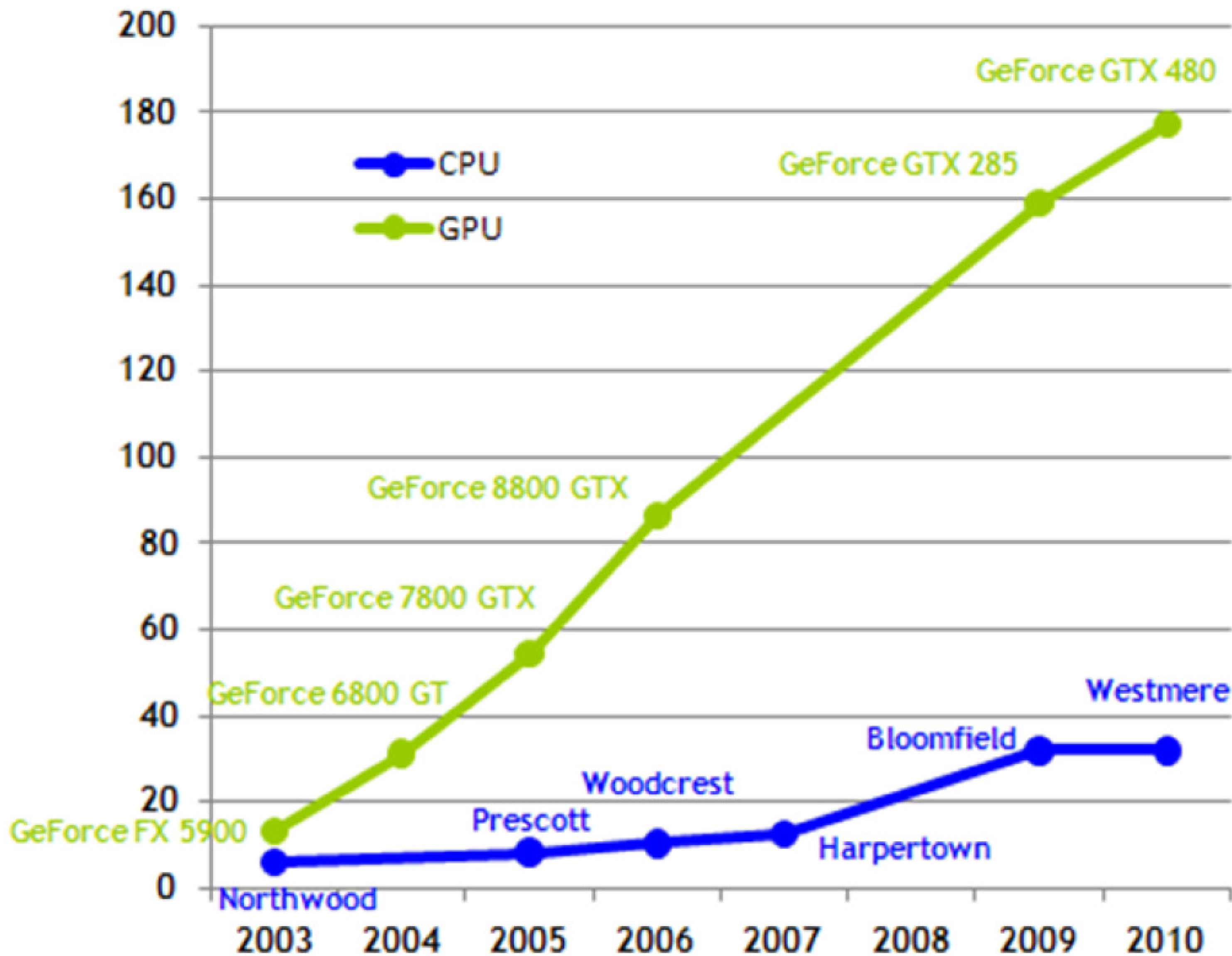
Grafické procesory

- Neustály a silný dopyt trhu po 3D grafike
- V reálnom čase a vysokom rozlíšení
- Programovateľné grafické procesorové jednotky
- Vysokoparalelné, mnohojadrové, viacvláknové
- Vysoký výpočtový výkon
- Vysoká pamäťová priepustnosť

Grafické procesory



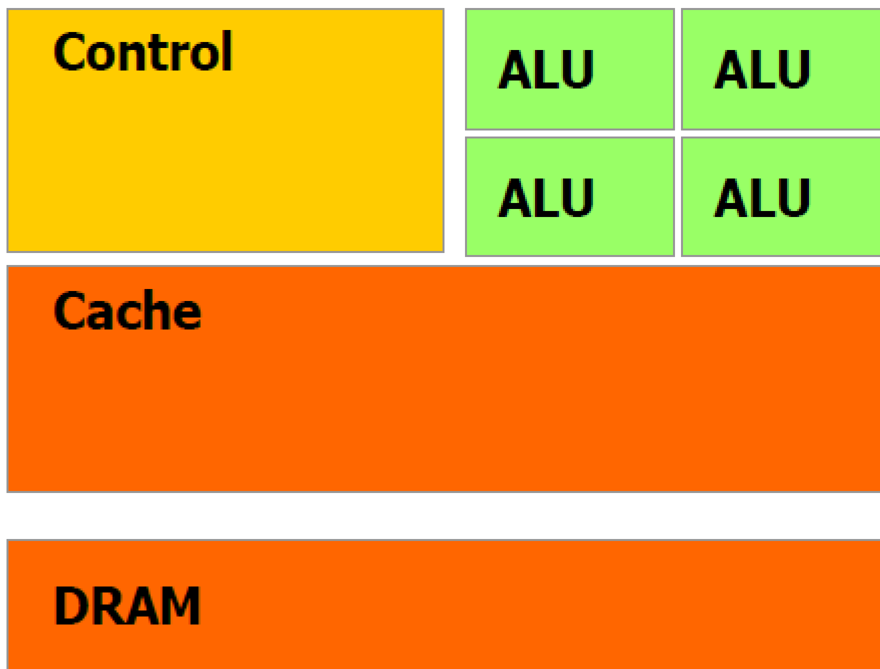
Grafické procesory

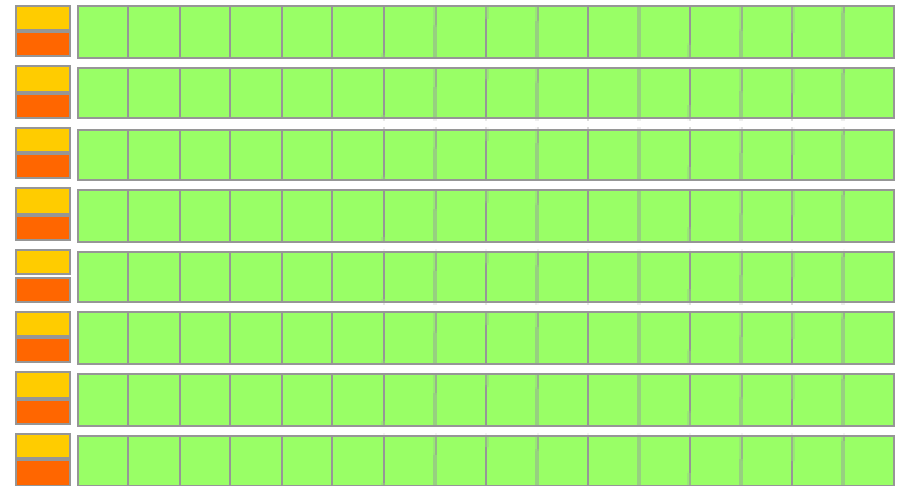


Grafické procesory

- Vysoký výpočtový výkon GPU
- GPU špecializované na výpočtovo náročné a vysokoparalelné výpočty – rendering v poč. grafike
- Viac polovodičových prvkov (tranzistorov) na spracovanie dát, než na cachovanie dát či riadenie toku

Grafické procesory





Grafické procesory

- GPU sú veľmi vhodné na dátovo paral. prob.
  - Rovnaký program na rôznych dátových prvkoch
  - Vysoká aritmetická intenzita – viac výpočtov než presunov (pomer aritmetických operácií k pamäťovým)
- Rovnaký program pre každý dátový prvok
  - Menšie nároky na riadenie toku
- Veľa dát a veľa výpočtov
  - Čas prístupu do pamäte je „schovaný“ za výpočty
  - Nie riešenie cez veľké cache pamäte

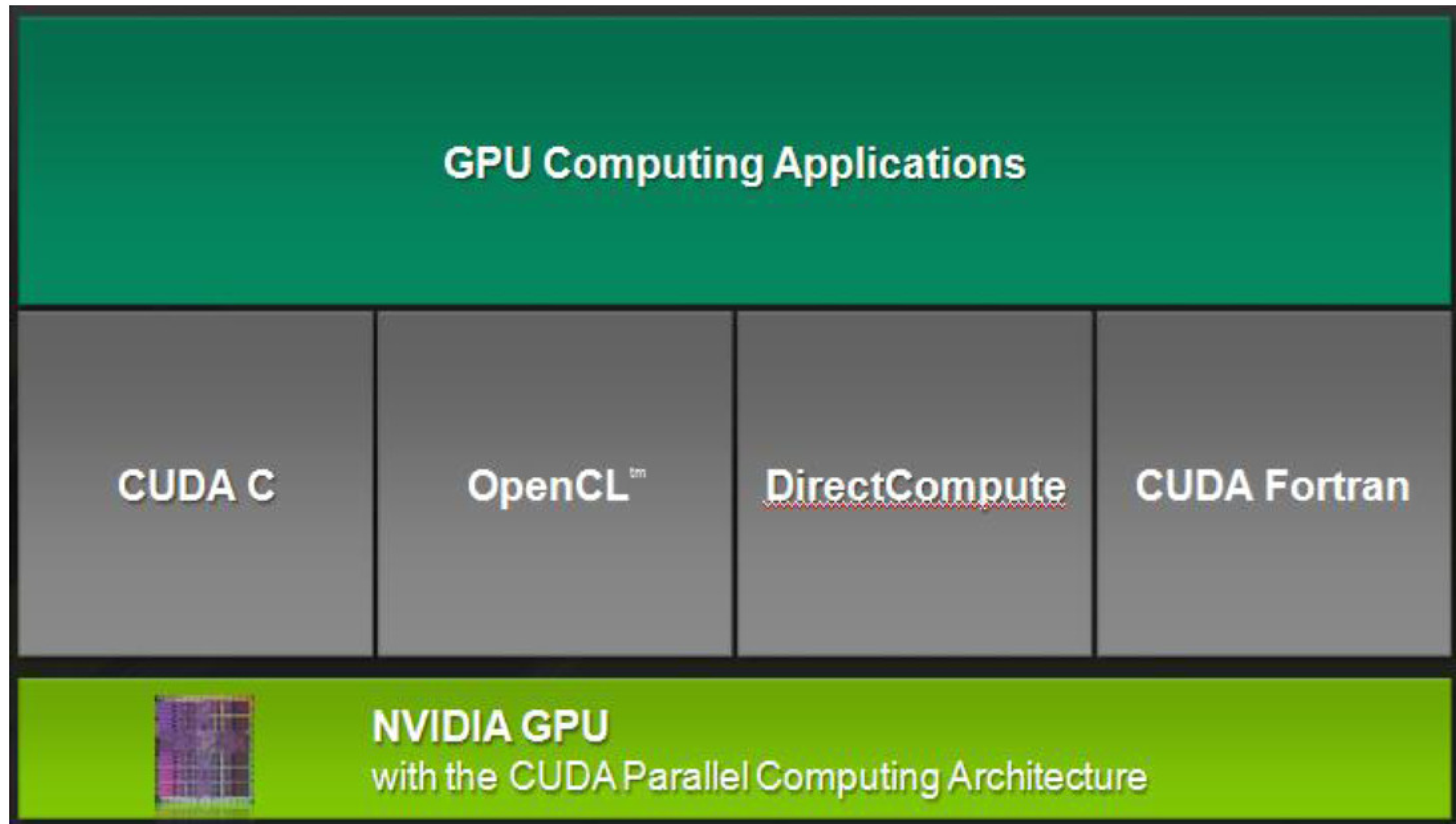
Grafické procesory

- Dátový paralelizmus - mapovanie dátových prvkov na súbežne vykonávané vlákna
- Vhodné pre mnohé aplikácie – problémy vyžadujúce spracovanie rozsiahlych údajov
- 3D renderovanie – mapovanie veľkých množín pixelov a vrcholov na paralelné vlákna
- Spracovanie obrazu, zvuku, kódovanie a dekódovanie videa, rozpoznávanie vzorov – tiež mapovanie obrazu a pixelov na paralelné vlákna
- Nielen vytváranie a spracovanie obrazu: všeobecné spracovanie signálov, fyzikálne simulácie, výpočtová ekonomika, výpočtová biológia

CUDA

- CUDA – Compute Unified Device Architecture
- Posledné dve generácie GPU od Nvidia
  - G80 a Fermi
- Programátorský model CUDA – možnosť využiť vlastnosti GPU
  - CUDA C
  - CUDA Driver API
  - CUDA Fortran

CUDA



CUDA

- Viacjadrové CPU a mnohojadrové GPU
  - Paralelné procesory, ktorých výkon zodpovedá Moorovmu zákonu
- Výzva – transparentná škálovateľnosť
  - Vytvoriť SW, ktorý využije rastúci počet výpočtových jadier, podobne ako 3D grafické aplikácie
- Riešenie - programátorský model CUDA

CUDA

- Programátorský model CUDA
- Dobre zvládnuteľný pre C programátorov
- Tri abstrakcie – minimálna množina rozšírení jazyka
  - Hierarchia vlákien
  - Zdieľané pamäte
  - Barierová synchronizácia

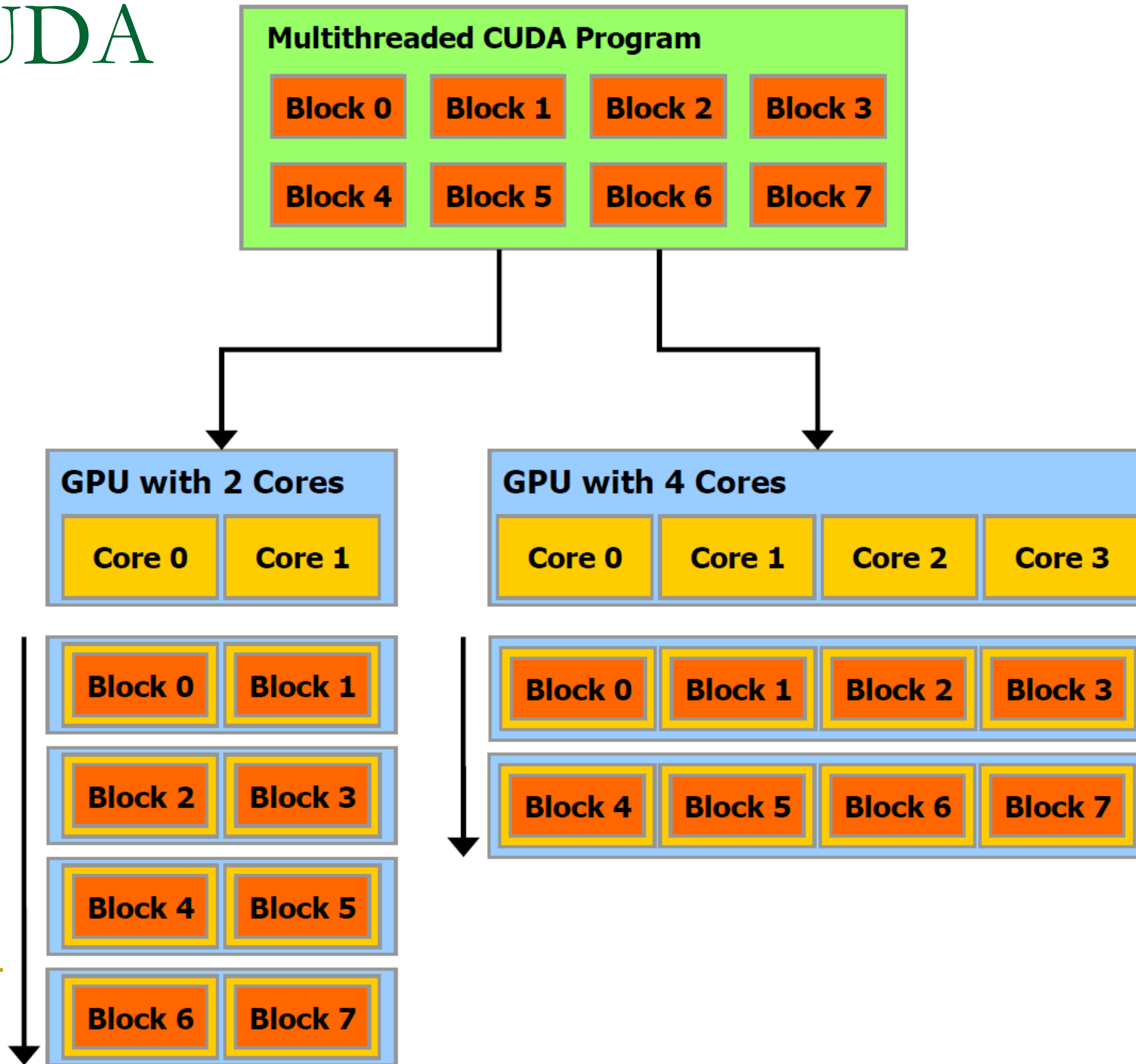
CUDA

- Jemnozrnný dátový paralelizmus + paradigma viacvláknového paralelného programovania vnorené do hrubozrnného paralelizmu a paralelných úloh
- Rozdelenie problému na „hrubšie“ súbežne riešiteľné nezávislé podproblémy riešiteľné blokmi vlákien a rozdelenie každého podproblému na „jemnejšie“ časti riešiteľné prostredníctvom vzájomnej kooperácie súbežne jednotlivými vláknami bloku

CUDA

- Takáto dekompozícia problému:
- Vysoká expresivita jazyka
  - Možnosť vyjadriť a zrealizovať paralelné riešenie prostredníctvom kooperujúcich súbežne vykonávaných vlákien
- Škálovateľnosť paralelnho riešenia
  - Možnosť naplánovať vykonanie bloku vlákien na ľubovoľnom voľnom procesore v ľubovoľnom poradí, sekvenčne alebo paralelne s inými blokmi
  - Možnosť riešiť problém na rôznych GPU (profesionálne riešenia aj jednoduchšie produkty)

CUDA



Programátorský model

- Kernel - rozšírenie j. C, špecifická funkcia
- Vykonaná N krát súbežne pomocou N CUDA vlákien (nie 1x ako klasická C funkcia)
- Deklarácia pomocou **__global__**
- Zavolanie pomocou novej syntaktickej konštrukcií definujúcej vykonanie **<<<…>>>**

Programátorský model

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

int main()
{
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
}
```

Programátorský model

- Hierarchia vlákien – vlákna organizované do blokov v zvolenej štruktúre
- **threadIdx** premenná – vektor s 3 zložkami – index vlákna v rámci bloku vlákien
- Možnosť organizovať vlákna do 1,2 alebo 3 rozmernej štruktúry
- Jedno, dvoj alebo troj rozmerný blok vlákien (vektor, matica, 3D matica)

Programátorský model

```
__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()
{
  // Kernel invocation
  dim3 dimBlock(N, N);
  matAdd<<<1, dimBlock>>>(A, B, C);
}
```

Programátorský model

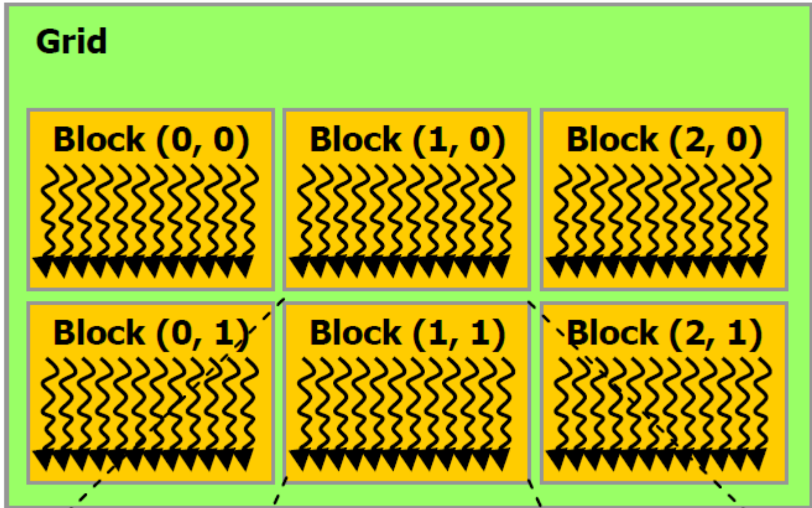
- Vzťah indexu (**threadIdx**) vlákna a ID vlákna
- Jednorozmerný blok o veľkosti (Dx)
  - Index (x): ID = x
- Dvojrozmerný blok o veľkosti (Dx, Dy)
  - Index (x,y): ID = x + y Dx
- Trojrozmerný blok o veľkosti (Dx, Dy, Dz)
  - Index (x,y,z): ID = x + y Dx + z Dx Dy

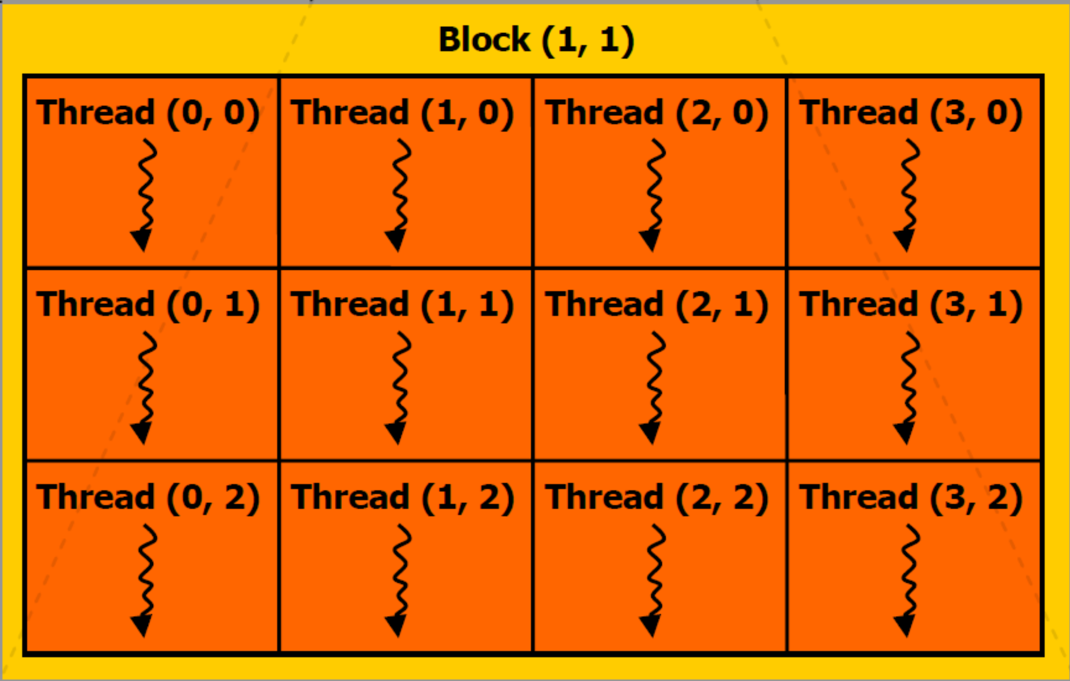
Programátorský model

- Kooperácia vlákien v bloku
  - Komunikácia prostredníctvom zdieľanej pamäti
  - Koordinácia prostredníctvom synchronizácie - **__syncthreads()** – barierová synchronizácia
- Zdieľaná pamäť – rýchla (malá latencia), podobne ako L1 cache
- **__syncthreads()** - odľahčená operácia, všetky vlákna bloku bežia na jednom jadre, počet vlákien v bloku je obmedzený pamäťovými zdrojmi jadra (multiprocesora)

Programátorský model

- Blok vlákien – max. 512 (1024) vlákien
- Kernel môže byť vykonaný nad viacerými blokmi – celkový počet počet vlákien je daný počtom blokov x počtom vlákien v bloku
- Viaceré bloky organizované do 1 alebo 2 rozmernej štruktúry – gridu (mriežky)
- Veľkosť gridu daná prvým parametrom v **<<<…>>>**
- Každý blok identifikovaný pomocou **blockIdx**
- Veľkosť bloku je sprístupnená cez **blockDim**


Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)


Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)

Programátorský model

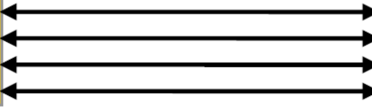
```
// Kernel definition
__global__ void MatAdd(float A[N][N],float B[N][N],float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N) C[i][j] = A[i][j] + B[i][j];
}
int main()
{
  ...
  // Kernel invocation
  dim3 threadsPerBlock(16, 16);
  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Programátorský model

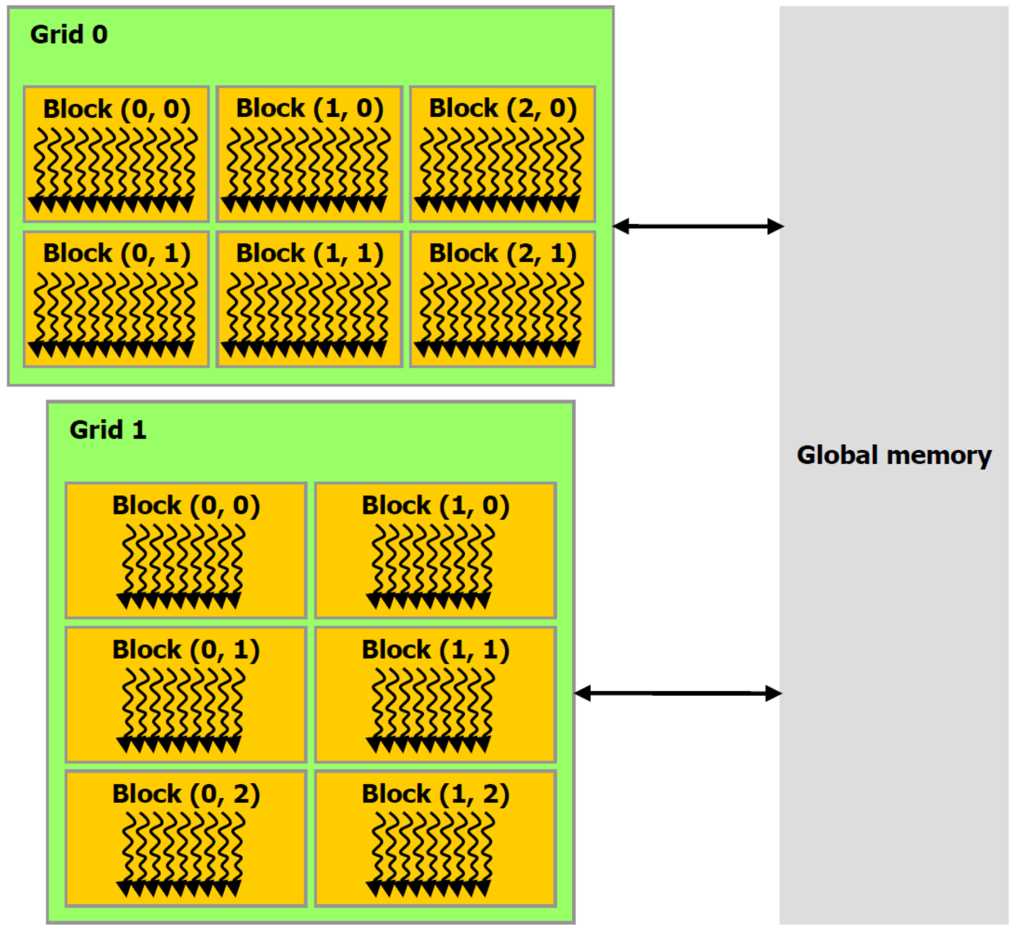
- Pamäťová hierarchia sprístupnená vláknam
- Súkromná lokálna pamäť vlákna (thread private local memory)
- Zdieľaná pamäť bloku vlákien (shared memory) - rovnaká životnosť ako má blok
- Hlavná pamäť (global memory) – prístupná všetkým vláknam všetkých blokov
- Pamäte iba na čítanie tiež prístupné všetkým vláknam: konštantná pamäť a pamäť textúr (constant a texture memory)

Thread

Per-thread local memory



Per-block shared memory



Programátorský model

- Heterogénny systém
- C program vykonávaný na CPU (host) volá kernel funkcie vykonávané na GPU zariadení (device) ako na koprocesore
- Hostiteľ aj zariadenie si udržujú vlastný pamäťový priestor: host a device memory
- Program riadi hlavnú, konštantnú a textúrovú pamäť (prístupné pre všetky vlákna kernelu) prostredníctvom volaní knižničných funkcií

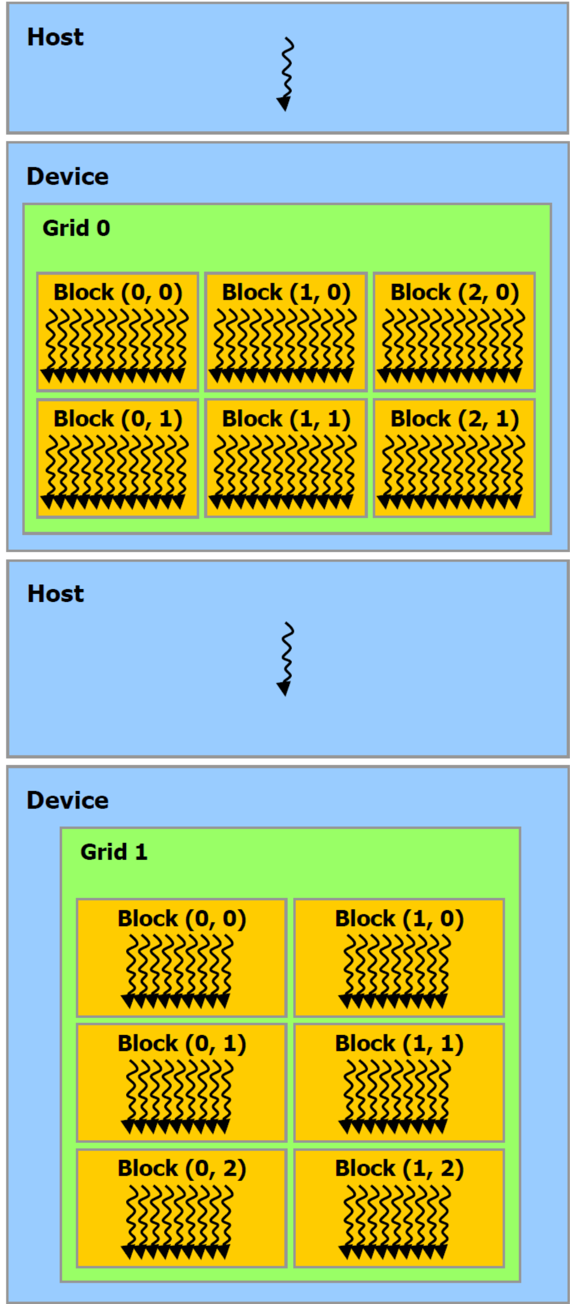
C Program Sequential Execution

`Serial code`

`Parallel kernel`
`Kernel0<<<>>>()`

`Serial code`

`Parallel kernel`
`Kernel1<<<>>>()`



Programátorský model

- Vlastnosti zariadenia (*compute capability*) sú dané havným a vedľajším číslom revízie (*major revision number a minor revision number) – Tesla C2050 – 2.0*
- *G80 – 1.x*
- *Fermi – 2.x*

Programátorký model CUDA C

- Pamäť zariadenia (Device Memory)
- Kernel vie pristupovať iba do pamäťovej hierarchie zariadenia
- Alokovateľná ako lineárna pamäť alebo CUDA arrays
- Lineárna pamäť:
  - cudaMalloc()
  - cudaFree
  - cudaMemcpy

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

// Host code
int main()
{
  int N = ...;
  size_t size = N * sizeof(float);

  // Allocate input vectors h_A and h_B in host memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);

 // Initialize input vectors
  ...
```

```
    // Allocate vectors in device memory
    float* d_A; cudaMalloc(&d_A, size);
    float* d_B; cudaMalloc(&d_B, size);
    float* d_C; cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =  (N + threadsPerBlock - 1)/threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

    ...
}
```

Programátorký model CUDA C

- Zdieľaná pamäť (Shared Memory)
- Významne rýchlejšia ako globálna pamäť
- Alokovaná pomocou kľúčového slova **__shared__**

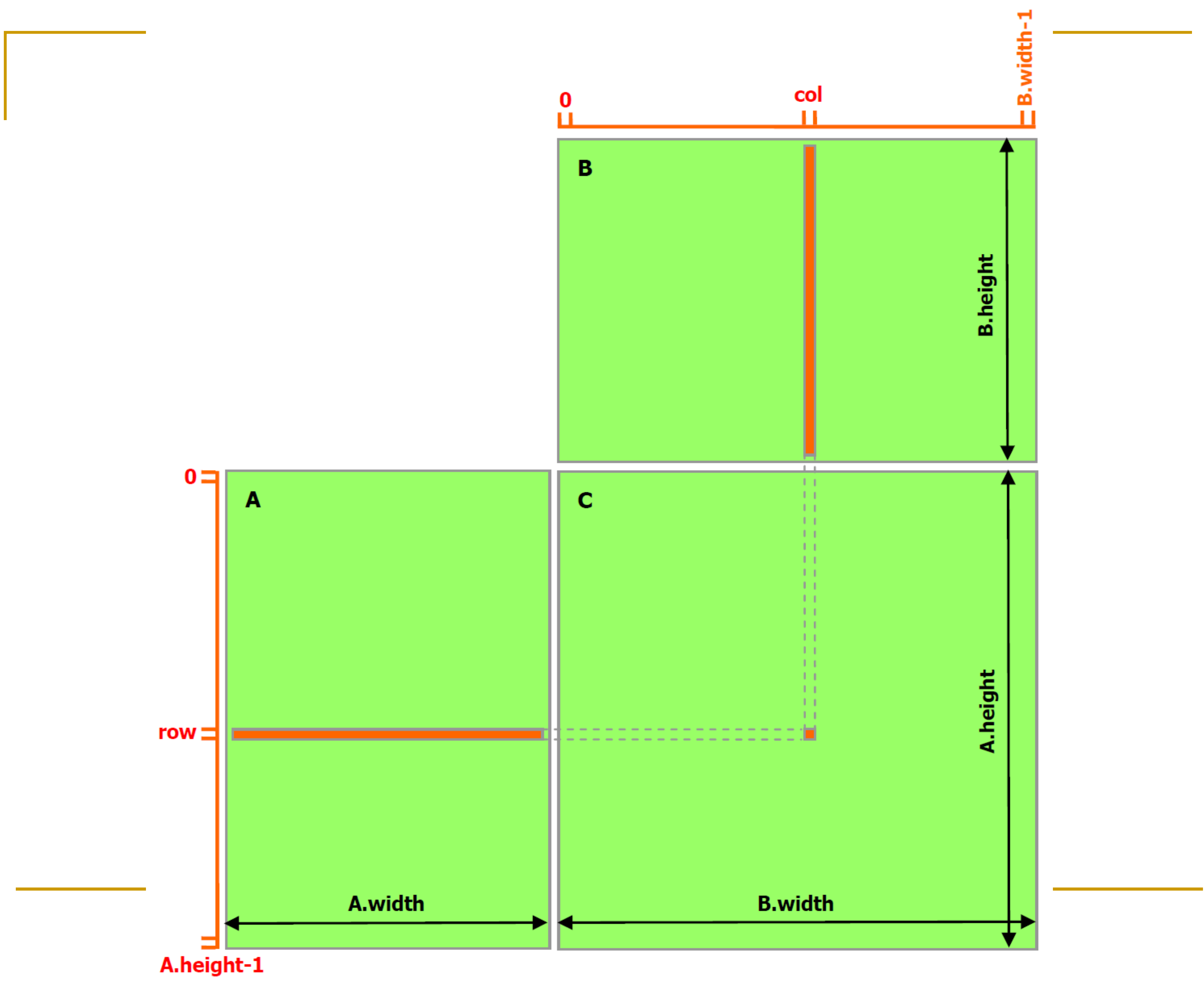
- Násobenie matice bez a so zdieľanou pamäťou

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
  int width;
  int height;
  float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);


// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
// void MatMul(const Matrix A, const Matrix B, Matrix C)
```

```
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
  // Load A and B to device memory
  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);
  cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

  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);

  // Allocate C in device memory
  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);
```

```
    // Invoke kernel
    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);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
    cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  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;
}
```



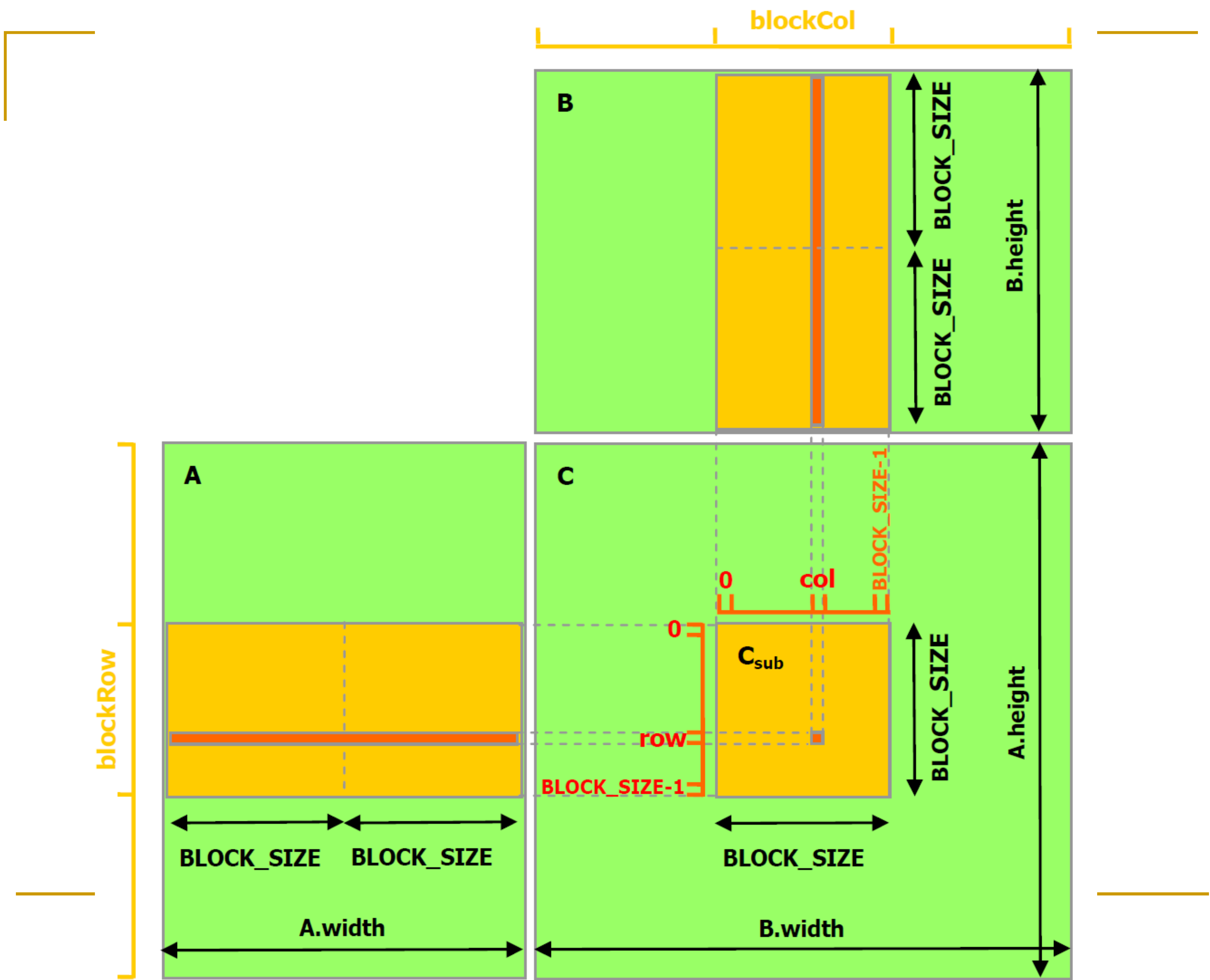
0
col
B.width-1
B
B.height
0
A
C
row
A.height
A.width
B.width
A.height-1

```
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
  return A.elements[row * A.stride + col];
}

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

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__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;
}
```

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;

  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;
```

```
// Loop over all the sub-matrices of A and B that are
// required to compute Csub
// Multiply each pair of sub-matrices together
// and accumulate the results
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

  // Get sub-matrix Asub of A
  Matrix Asub = GetSubMatrix(A, blockRow, m);

  // Get sub-matrix Bsub of B
  Matrix Bsub = GetSubMatrix(B, m, blockCol);

  // Shared memory used to store Asub and Bsub respectively
  __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

  // Load Asub and Bsub from device memory to shared memory
  // Each thread loads one element of each sub-matrix
  As[row][col] = GetElement(Asub, row, col);
  Bs[row][col] = GetElement(Bsub, row, col);
```

```
        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
          Cvalue += As[row][e] * Bs[e][col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```



blockCol
B
BLOCK_SIZE
BLOCK_SIZE
B.height
A
C
BLOCK_SIZE-1
0
col
0
Csub
row
BLOCK_SIZE-1
BLOCK_SIZE
A.height
blockRow
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
A.width
B.width

CUDA - HW Implementácia

- Prúdové multiprocesory (Streaming Multiprocessors)
- Blok vlákien je vykonávaný na jednom SM
- Keď je výpočet bloku ukončený, nový blok je naplánovaný na vykonanie na SM
- SM – súbežné vykonávanie stoviek vlákien
- SIMT arch. Single Instruction Multiple Thread
- HW vlákna, paralelizmus n úrovni inštrukcií, ale nie špekulatívne vykonávanie ani predikcia skokov

CUDA - HW Implementácia

- Vlákna sú plánované na vykonávanie b skupinách – warp (32 vlákien)
- Každé vlákno vlastná sada registrov – nezávislé vykonávanie
- Warp – jedna inštrukcia v čase, efektívne, ak všetky vlákna rovnaká inštrukcia
- SIMT – jedna inštrukcia riadi viaceré jadrá multiprocesora

Zdroje

- Nvidia CUDA Programming Guide http://developer.nvidia.com

- Obrázky prevzaté z:
  - Nvidia CUDA Programming Guide http://developer.nvidia.com