IF3230
Sistem Paralel dan Terdistribusi
paralel programming model: GPU

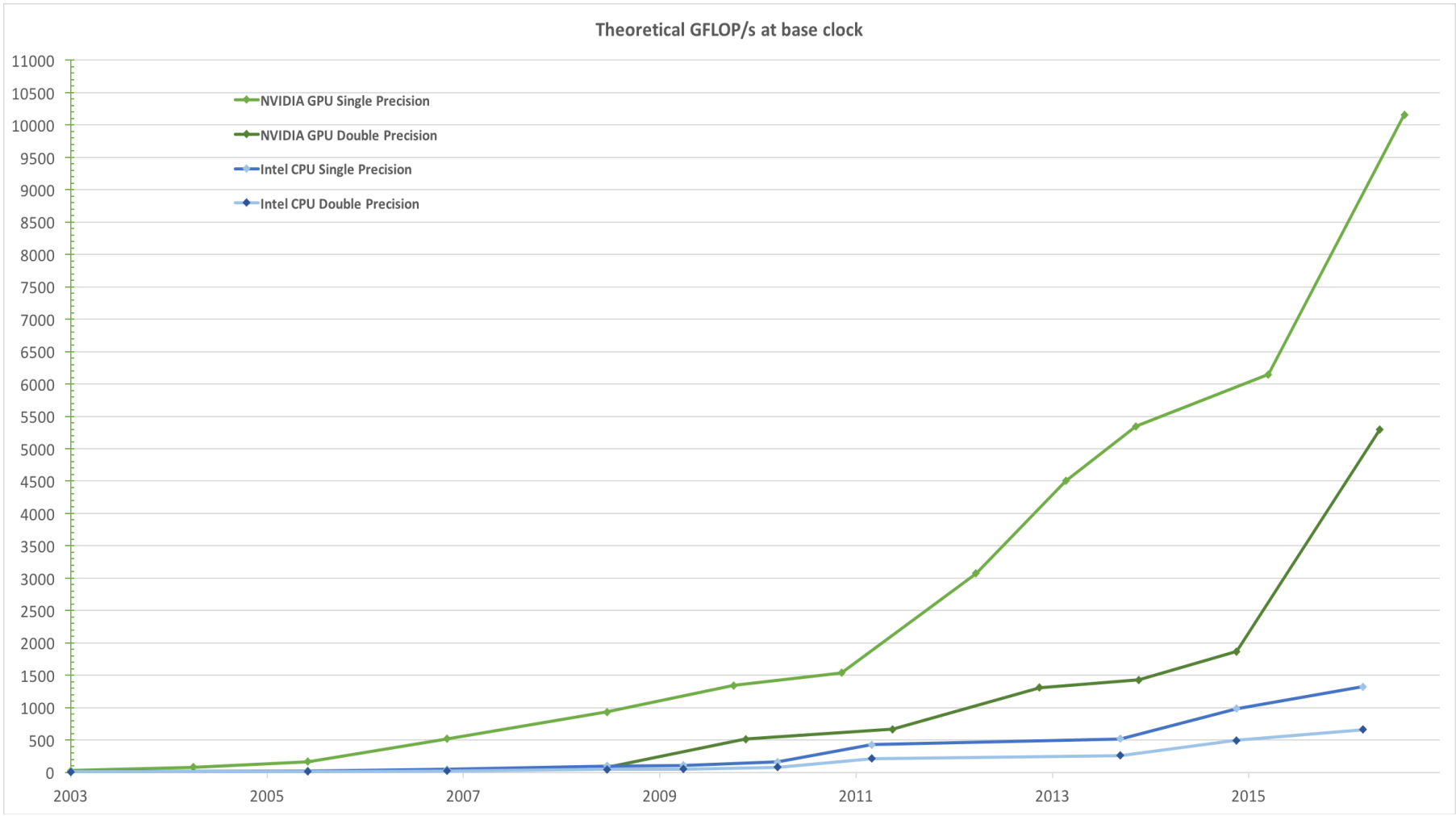
Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Judhi Santoso (judhi@staff.stei.itb.ac.id)
Anggrahita Bayu Sasmita (angga@staff.stei.itb.ac.id) Februari 2022
Informatika – STEI – ITB

GPGPU

- General Purpose computation on GPU
- GPU dirancang untuk dapat melakukan komputasi grafis dengan cepat
- Terdiri atas banyak core (RTX 3080: 8704 cores)
  - mampu menjalankan threads dalam jumlah sangat besar (orde 10.000 an)
  - Skala ekonomi besar => pasar games/grafis besar => perkembangan teknologi pesat
- Massively Parallel Computing: komputasi parallel menggunakan threads/parallel computing unit dalam jumlah besar

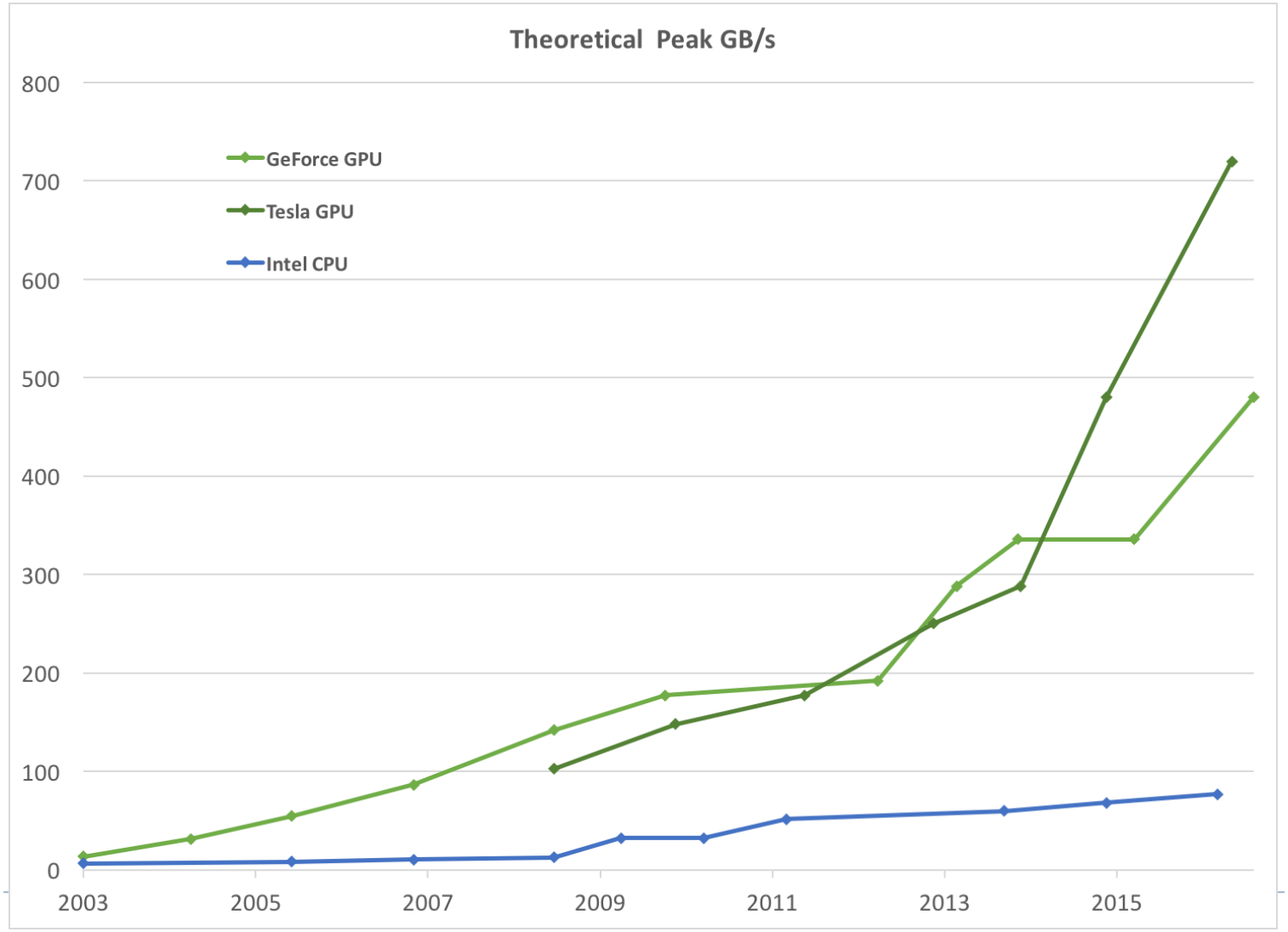
Mengapa Massively Parallel Processing?

- Tren perkembangan GPU vs CPU



Mengapa Massively Parallel Processing?

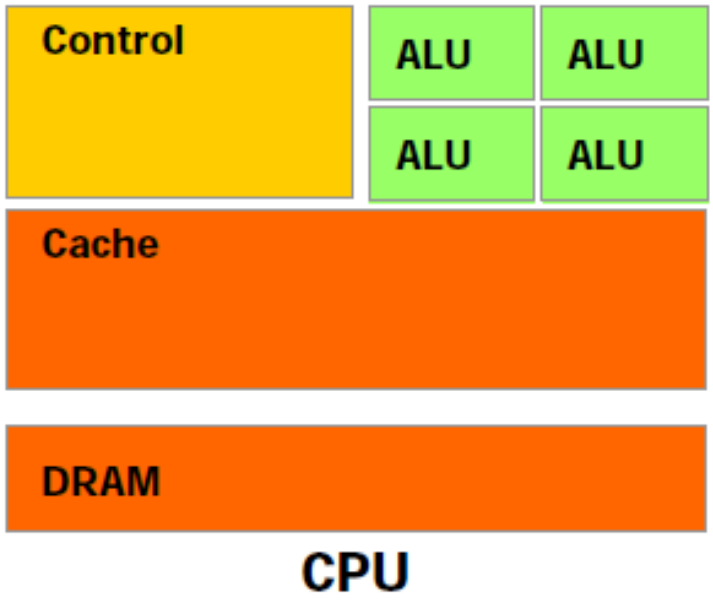
- A quiet revolution and potential build-up

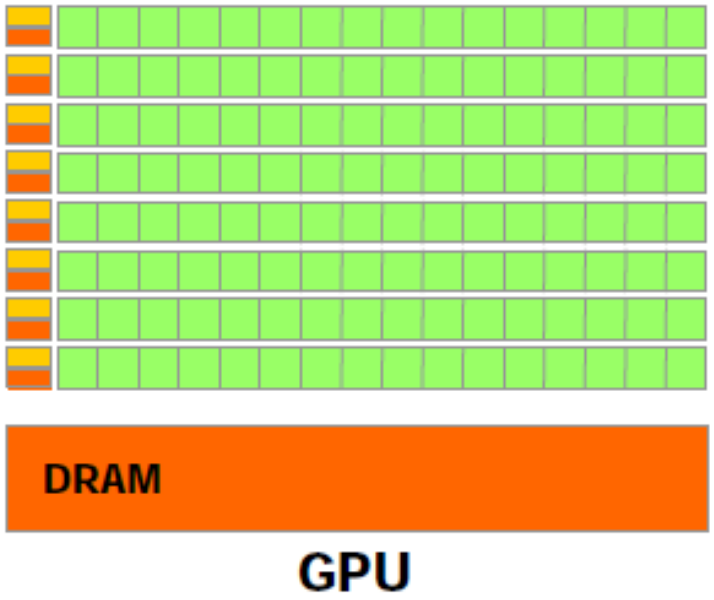


Computational Power

- GPU memiliki computational power besar dengan pendekatan:
- High throughput design (GPU) vs Latency oriented design (CPU)
  - GPU: Sebagian besar transistor digunakan untuk komputasi
  - CPU: sebagian transistor untuk mengurangi latency akses ke memori (untuk cache)
  - GPU menyediakan ALU banyak, namun tidak kompleks
  - CPU memiliki ALU yang mendukung operasi superscalar, seperti branch prediction, out of order execution, mampu mengeksekusi banyak jenis operasi dengan cepat

Ilustrasi perbandingan alokasi transistor pada CPU dan GPU





GPGPU

- GPGPU: menggunakan GPU untuk komputasi selain grafis
- GPU digunakan untuk akselerasi critical path pada aplikasi
  - critical path: bagian program yang memerlukan komputasi besar/waktu lama
- Cocok untuk data parallel algorithms
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

Kapan GPU tidak memiliki kinerja bagus

- Lack of parallelism.

  "*With great power comes great need for parallelism*", Spiderman

  Jika tidak ada parallelism, GPU tidak banyak membantu

- Thread divergence

  Thread pada GPU berjalan SIMD. Jika banyak branch/cabang tidak efisien

- Dynamic memory requirements

  Alokasi memori dilakukan pada CPU, dan akan membatasi algoritma yang membutuhkan alokasi dinamis

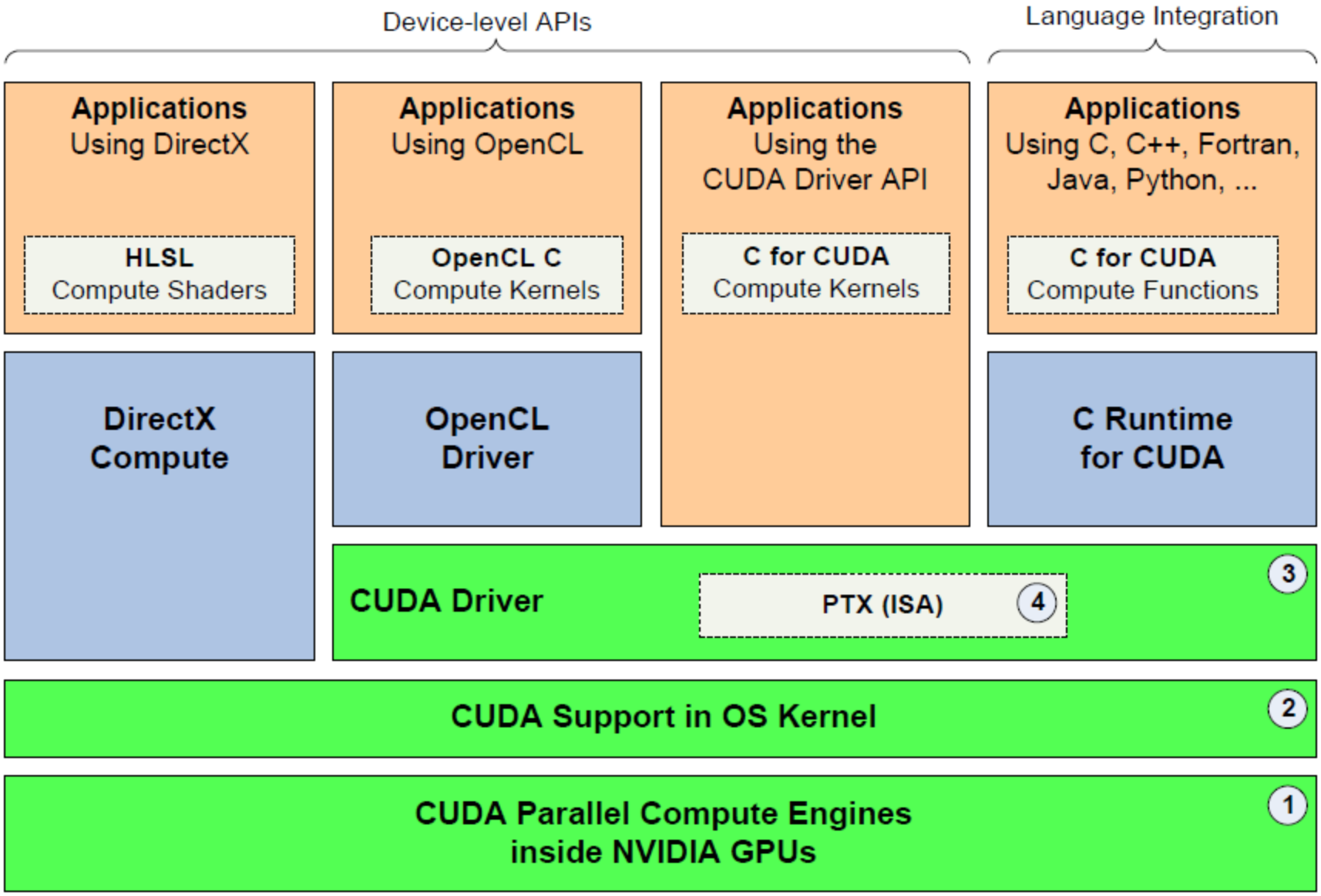
- Recursive algorithms

  GPU hanya memiliki resource memory stack terbatas

GPGPU & CUDA/OpenCL

- GPGPU: menggunakan graphics API untuk komputasi umum: vertex processor, texture cache, etc.
- data format diubah ke bentuk texture, komputasi pada texture
- CUDA:model yang dikembangkan Nvidia untuk general app programming pada GPU
- Model standar (multi vendor): OpenCL

Arsitektur CUDA

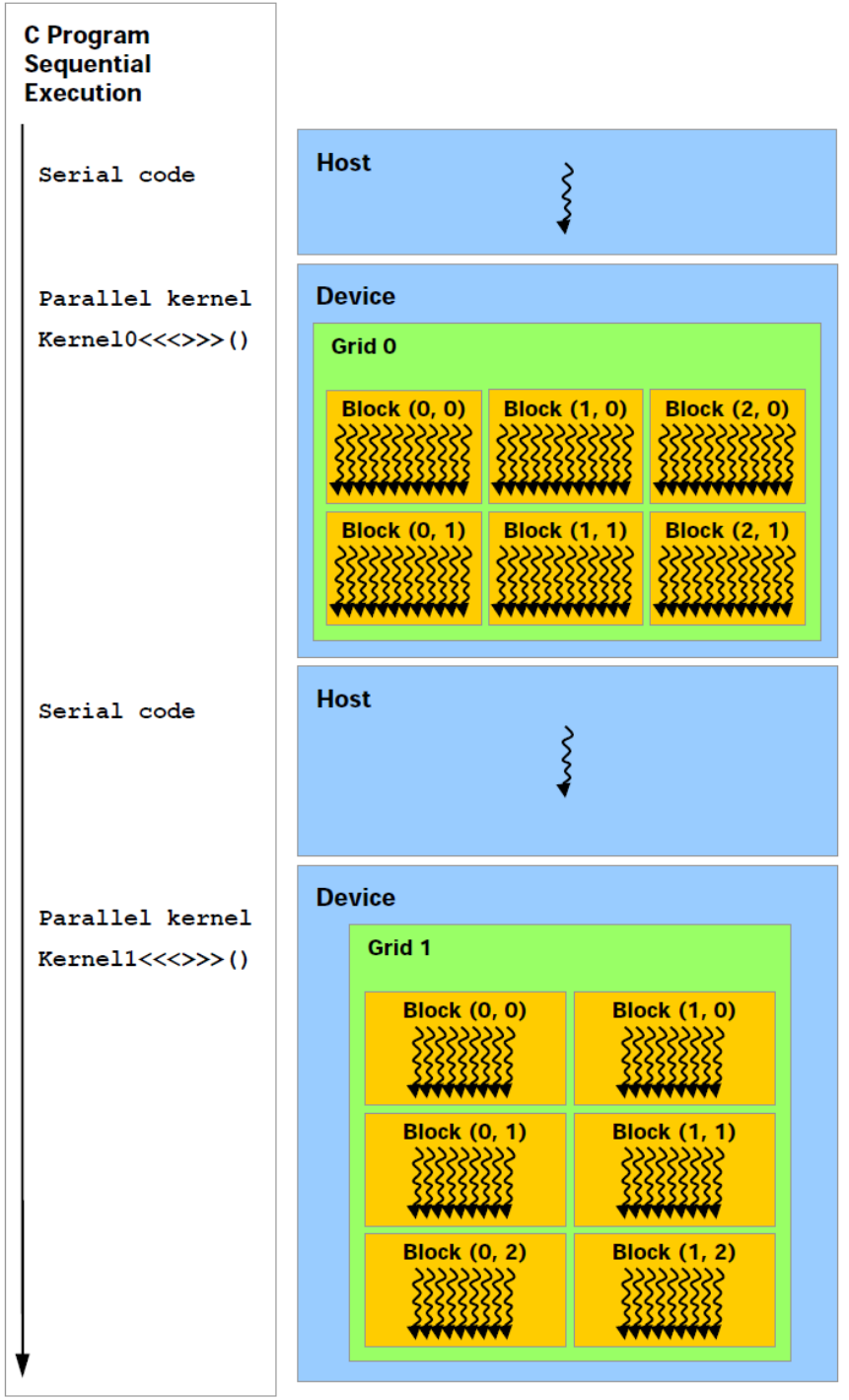


Model Pemrograman CUDA

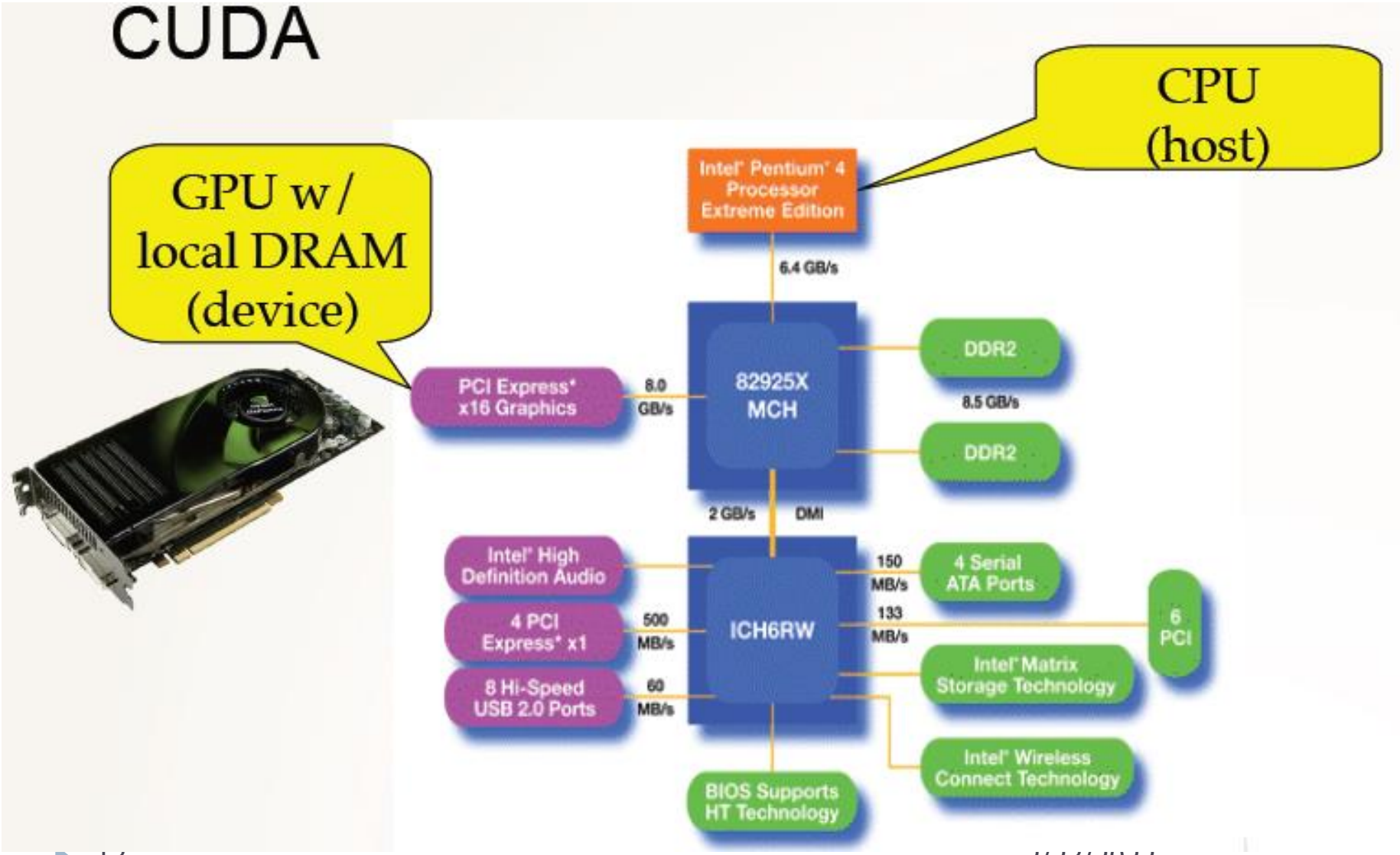
- GPU dapat dilihat sebagai alat komputasi:
  - Sebagai coprocessor untuk CPU (host)
  - Memiliki memori/DRAM sendiri (device memory)
  - Mampu menjalankan banyak threads secara parallel
- Kode/program paralel yang berjalan pada device/GPU disebut **kernel**
- Perbedaan thread GPU dan CPU
  - GPU threads sangat ringan (lightweight)
    - Overhead untuk pembuatan thread sangat kecil
  - GPU memerlukan ribuan threads agar terpakai penuh
    - Multi-core CPU hanya memerlukan sedikit threads

Model Pemrograman CUDA

- Host: komputer
- Device: GPU
- Program berawal dari kode sekuensial yang berjalan pada host
- Saat memerlukan eksekusi pada device, host akan memanggil kernel yang akan dijalankan paralel pada device.
- Hasil komputasi dikirim ke host untuk di proses lebih lanjut



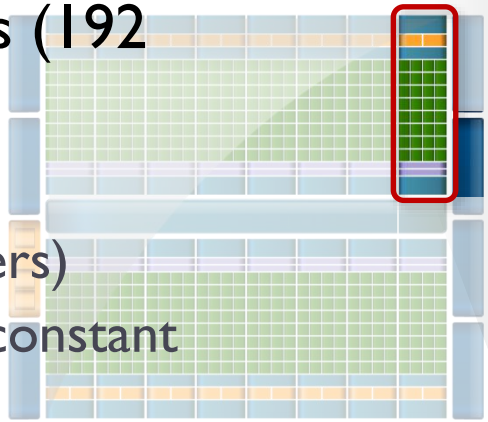
Bus/interkoneksi antar decice dan host

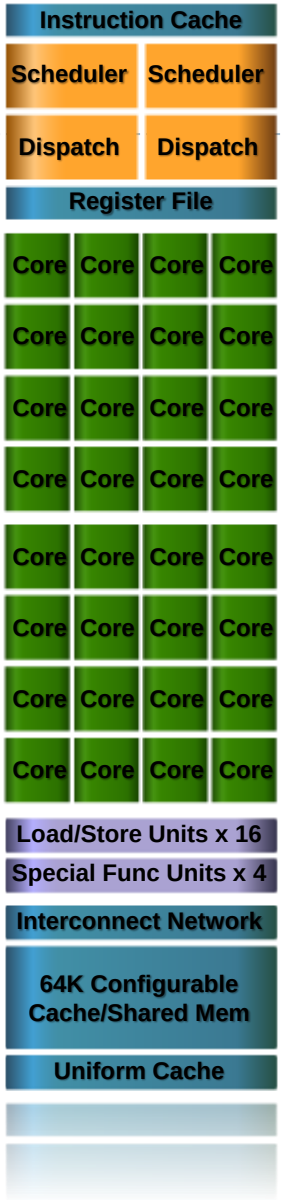


SM (Streaming Multiprocessor)

- GPU terdiri atas blok yang disebut Streaming Multiprocessor (SM) (12 SM pada GTX 780)

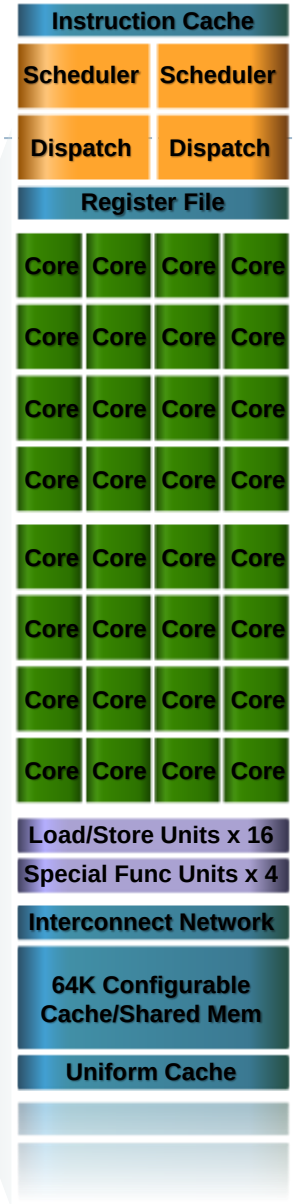
- 1 SM terdiri atas multiple cores (192 core/SM pada GTX 780)
- Setiap SM memiliki
  - ribuan registers (e.g. 64 K registers)
  - caches (shared memory(64KB), constant cache, texture cache, L1 cache
  - Warp/thread scheduler





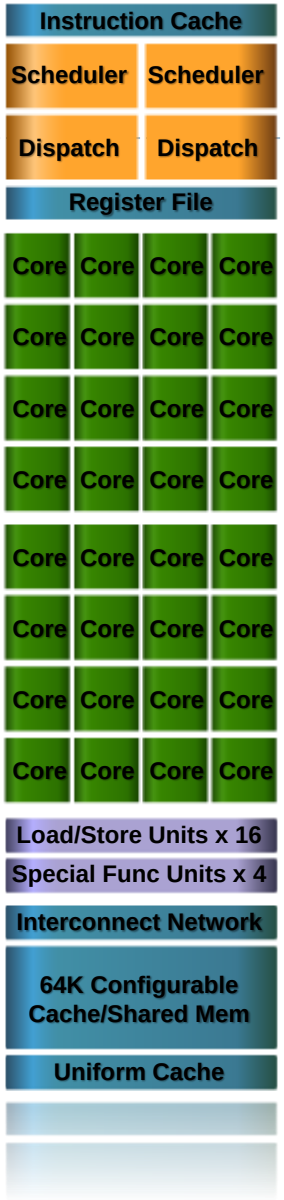
SM (Streaming Multiprocessor)

- Direct load/store to memory
  - Usual linear sequence of bytes
  - High bandwidth (Hundreds GB/sec)
- 64KB of fast, on-chip RAM
  - Software or hardware-managed
  - Shared amongst CUDA cores
  - Enables thread communication



Key Architectural Ideas

- SIMT (Single Instruction Multiple Thread) execution
  - threads berjalan dalam grup/blok berukuran 32 disebut warps
  - threads dalam satu warp menjalankan setiap instruction unit (IU) bersama
  - HW secara otomatis menangani perbedaan eksekusi antar threads dalam 1 warp
- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW menggunakan threads untuk menyembunyikan latency
- Threads have all resources needed to run
  - any warp not waiting for something can run
  - context switching is (basically) free



Perbedaan istilah pada CUDA vs OpenCL

Table 9.1 Hardware terminology: A rough translation

Host	OpenCL	AMD GPU	NVIDIA/CUDA	Intel Gen11
CPU	Compute device	GPU	GPU	GPU
Multiprocessor	Compute unit (CU)	Compute unit (CU)	Streaming multi-processor (SM)	Subslice
Processing core (Core for short)	Processing element (PE)	Processing element (PE)	Compute cores or CUDA cores	Execution units (EU)
Thread	Work Item	Work Item	Thread	
Vector or SIMD	Vector	Vector	Emulated with SIMT warp	SIMD

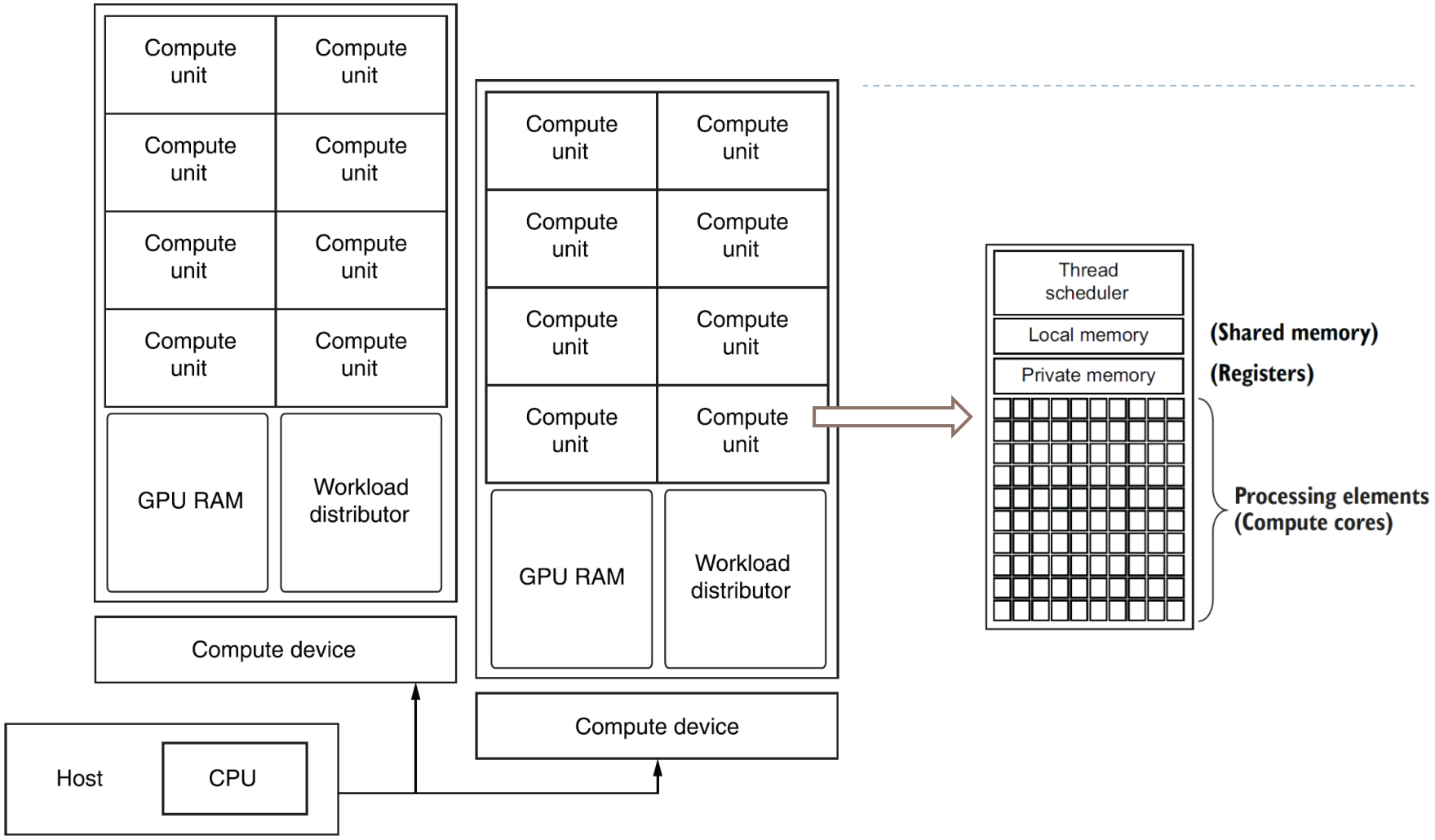
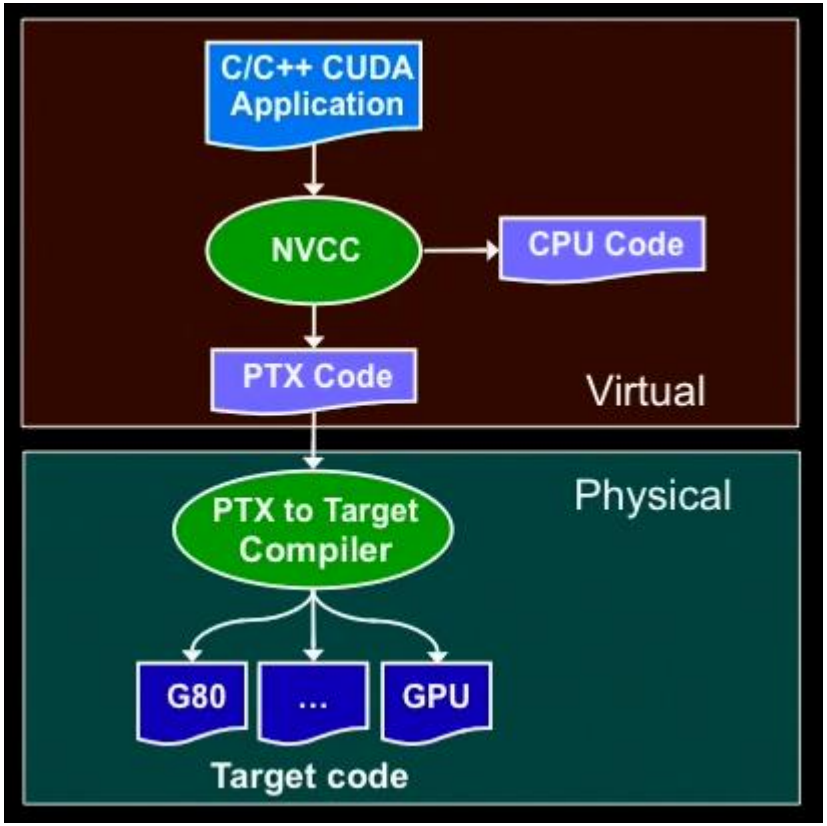


Figure 9.2 A simplified block diagram of a GPU system showing two compute devices, each having separate GPU, GPU memory, and multiple compute units (CUs). The NVIDIA CUDA terminology refers to CUs as streaming multiprocessors (SMs).

Kompilasi


C/C++ CUDA Application
NVCC
CPU Code
PTX Code
Virtual
Physical
PTX to Target Compiler
G80
...
GPU
Target code

NVCC tool

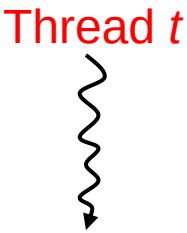
- nvcc <filename>.cu [-o <executable>]
  - build release mode
- nvcc -g <filename>.cu
  - build debug mode
- nvcc -deviceemu <filename>.cu
  - build device emulation
- nvcc -deviceemu -g <filename>.cu
  - build device emulation debug mode

Key Parallel Abstractions in CUDA

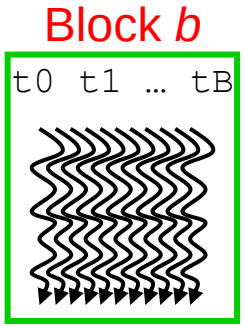
- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program



- Threads are grouped into thread blocks
  - threads in the same block can cooperate



- Threads/blocks have unique IDs

CUDA: Extended C

- **DecIspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

C for CUDA

- Philosophy: provide minimal set of extensions necessary to expose power

- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

- Variable qualifiers:

```
__constant__ float my_constant_array[32];
__shared__   float my_shared_array[32];
```

- Execution configuration:

```
dim3 grid_dim(100, 50);  // 5000 thread blocks
dim3 block_dim(4, 8, 8); // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void __syncthreads(); // Thread synchronization
```

Example: vector_addition

Device Code

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // elided initialization code
    ...
    // Run N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: vector_addition

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

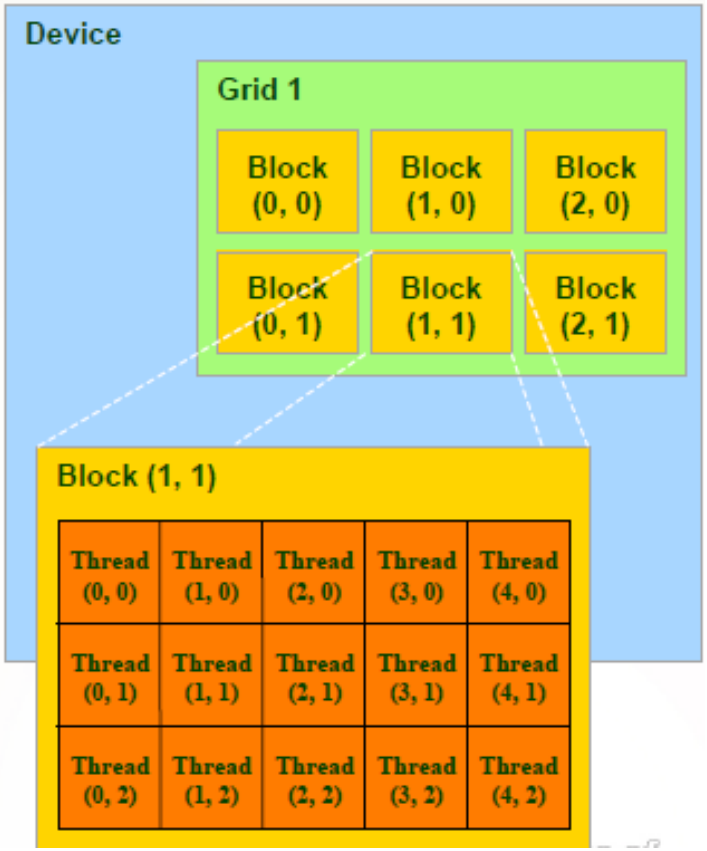
```
int main()
{
    // elided initialization code
    ...
    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Initialization code for vector_addition

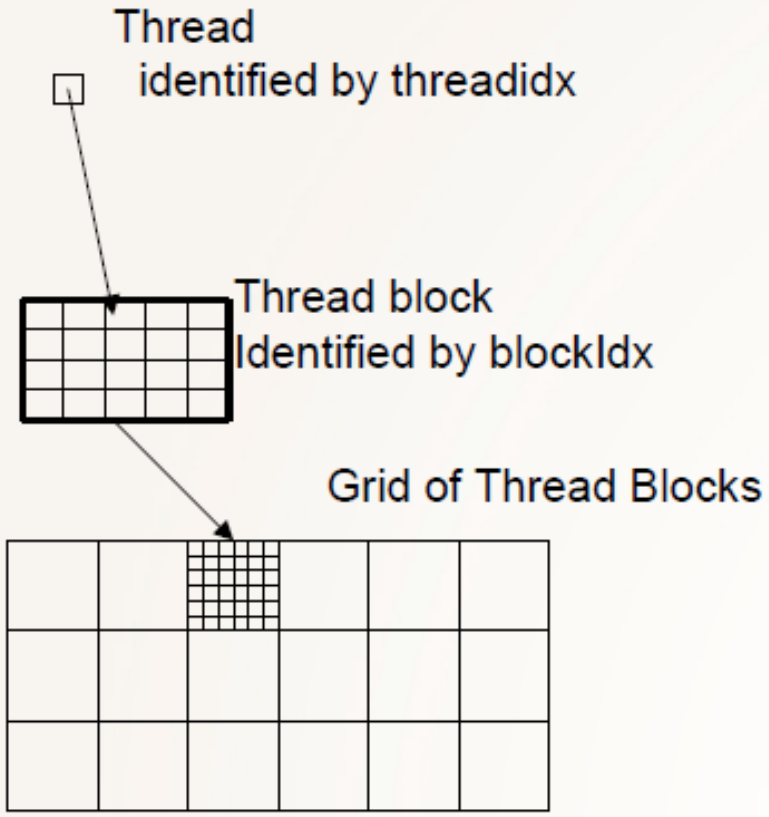
```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice
  );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice
  );

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(d_A, d_B, d_C);
```

Programming Model

- A kernel is executed as a grid of thread blocks
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Execution Model



Multiple levels of parallelism
-Thread block
    -Up to 512 threads per block
    -Communicate through shared memory
    -Threads guranteed to be resident
    -threadIdx, blockIdx
    -__syncthreads()
-Grid of thread blocks
    -F <<< nblocks, ntrheads >>> (a, b, c)

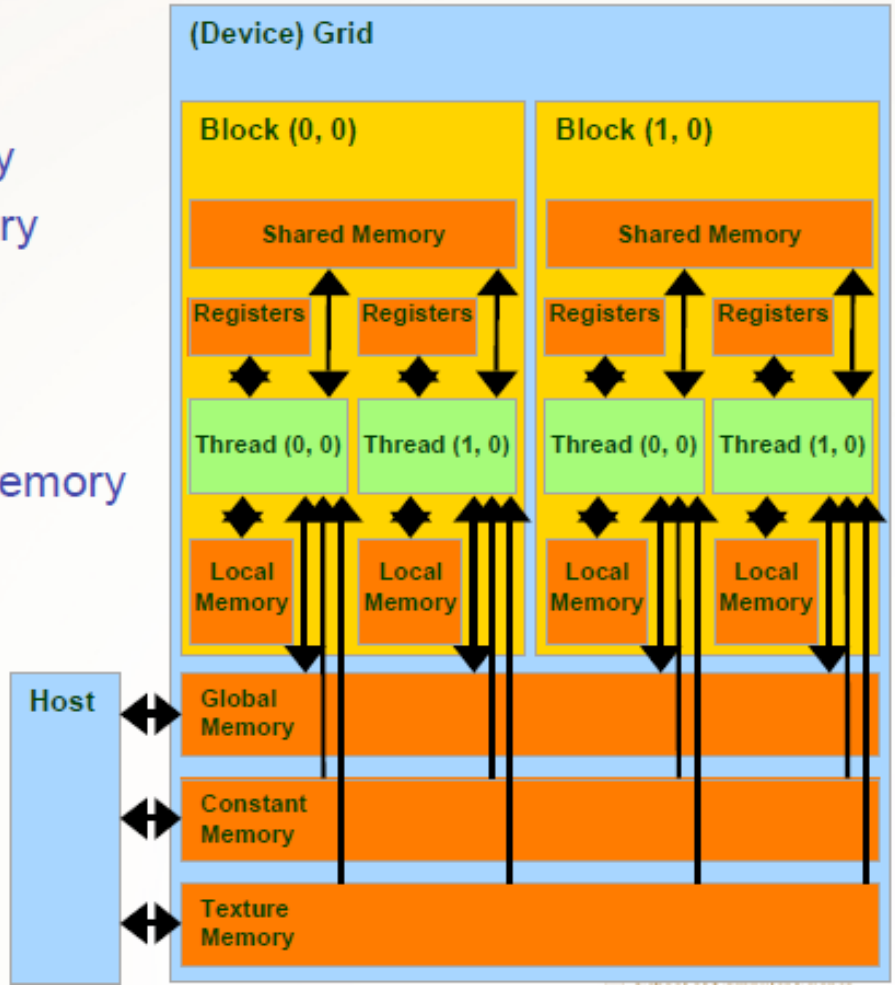
Code executed on GPU

- C/C++ with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
  - No dynamic polymorphism

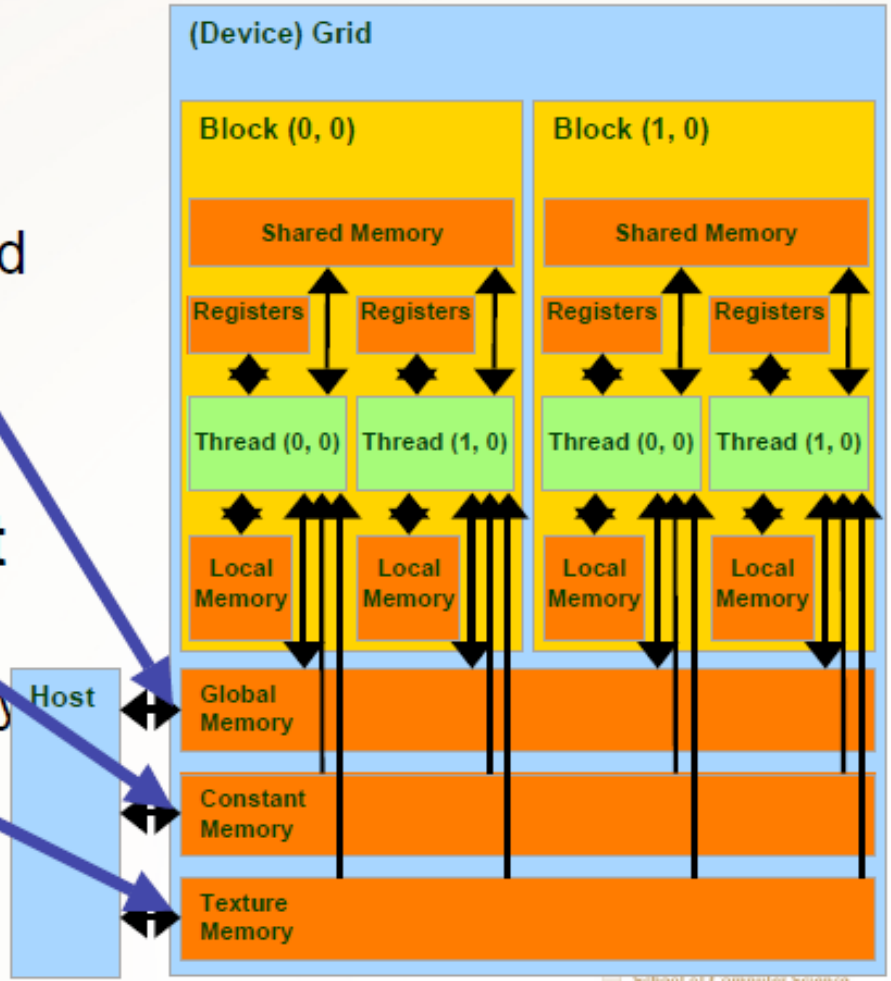
- Must be declared with a qualifier:
  - __global__ : launched by CPU,
    cannot be called from GPU must return void
  - __device__ : called from other GPU functions,
    cannot be called by the CPU
  - __host__ : can be called by CPU
  - __host__ and __device__ qualifiers can be combined
    - sample use: overloading operators

Memory Model

- Each thread 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
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories

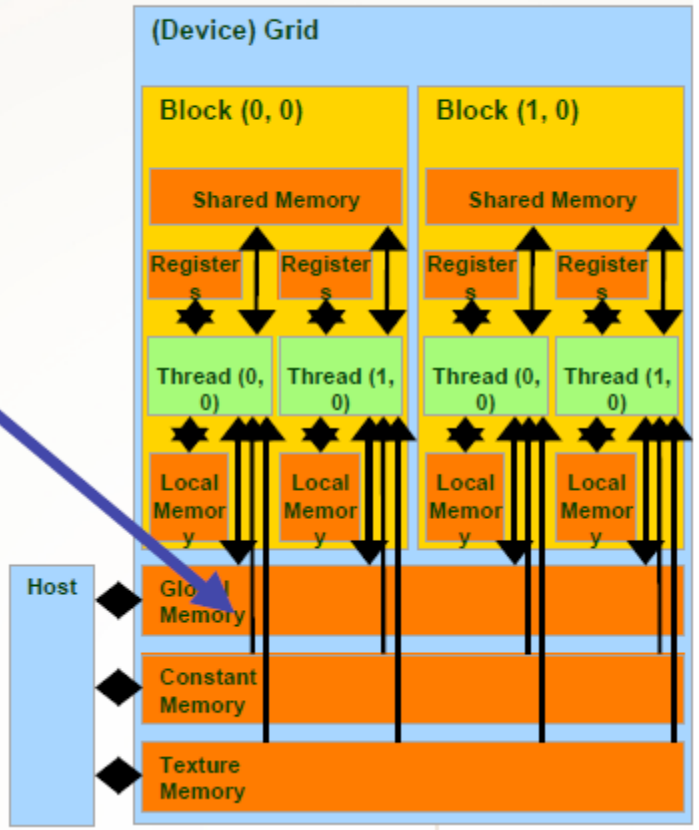


- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Device Memory Allocation

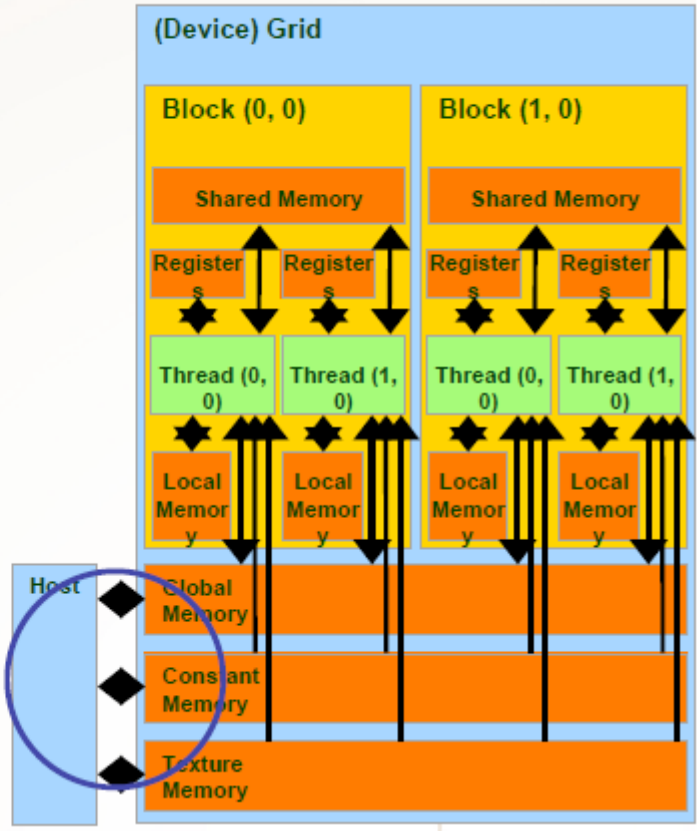
- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous in CUDA 1.0



- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md.elements, M.elements, size,
    cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
    cudaMemcpyDeviceToHost);
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Example: Shuffling Data

```
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int* new_array, int*
   indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);
}
```

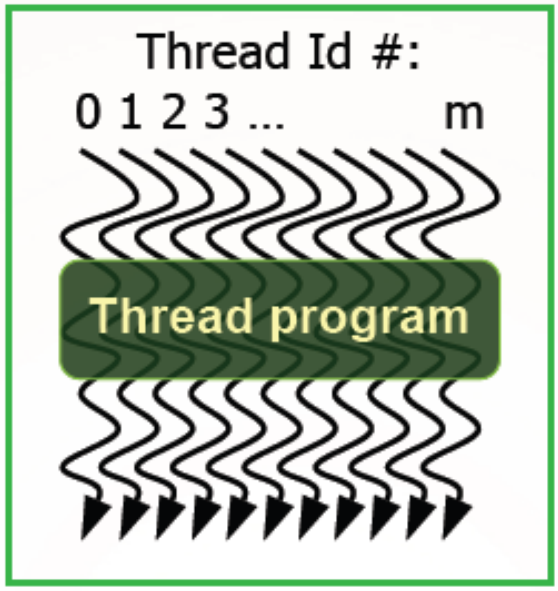
Host Code

CUDA Thread Block

- Programmer declares (Thread) Block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads

- All threads in a Block execute the same thread program
- Threads have thread id numbers within Block
- Threads share data and synchronize while doing their share of the work
- Thread program uses thread id to select work and address shared data

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

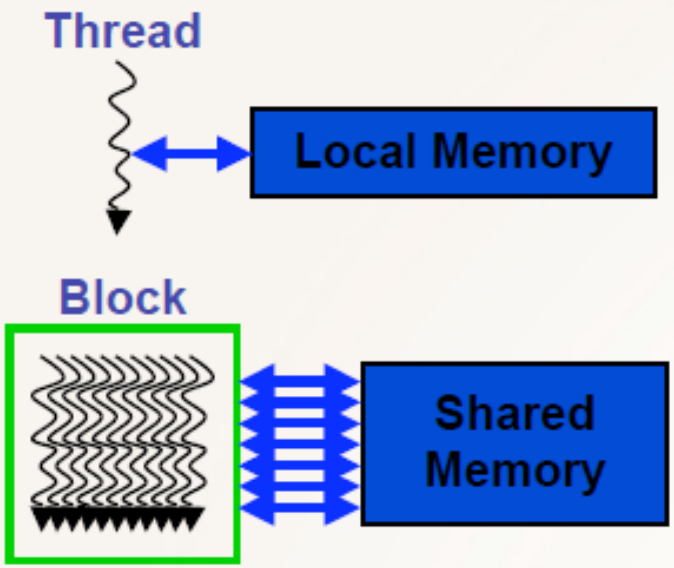
```
__global__ void mykernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    mykernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Blocks must be independent

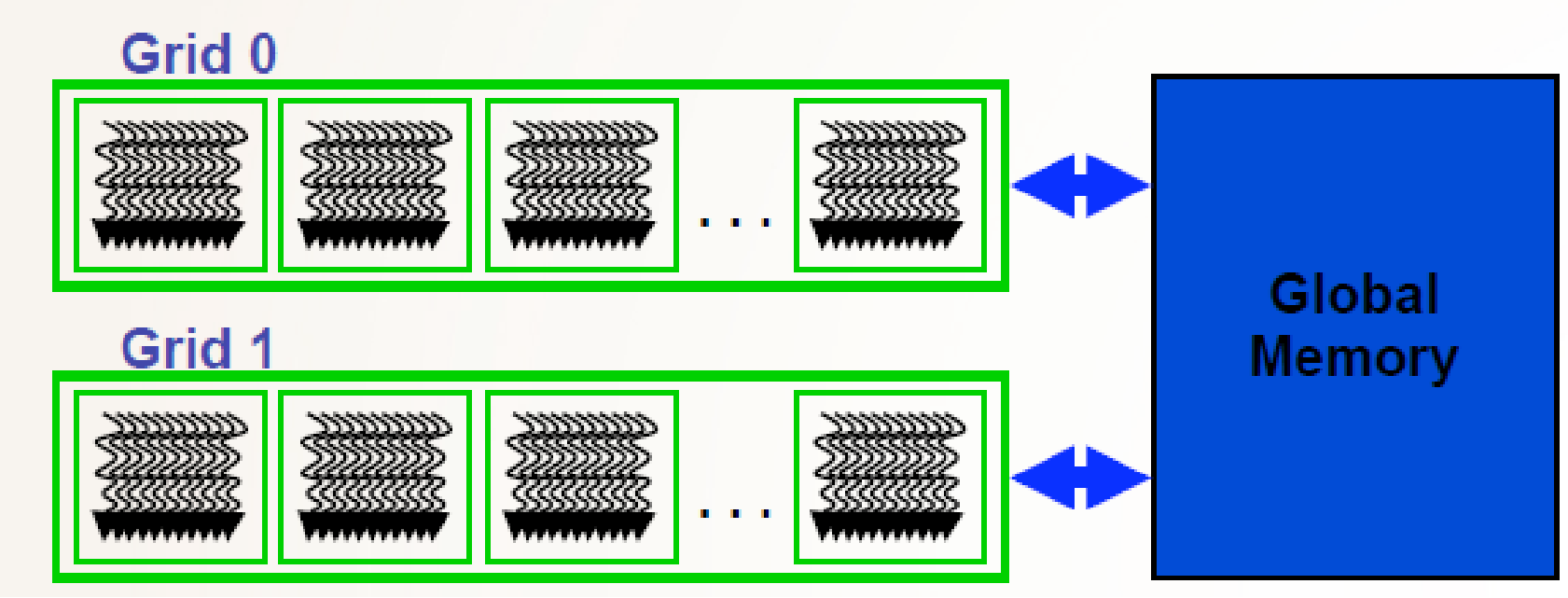
- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Shared Memory



- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

Shared Memory



- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per
   block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

Square Matrix Multiplication Example

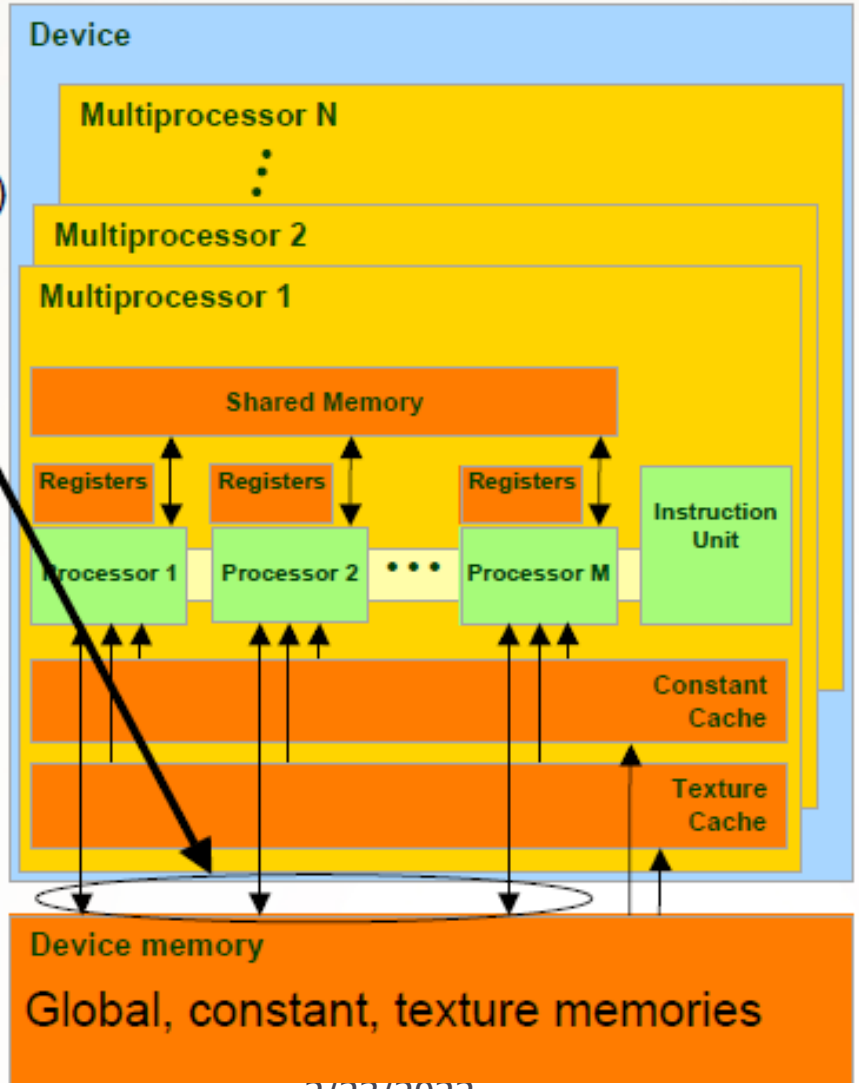
- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory

```
// 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)
{
    // 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((void**)&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((void**)&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((void**)&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);
}
```

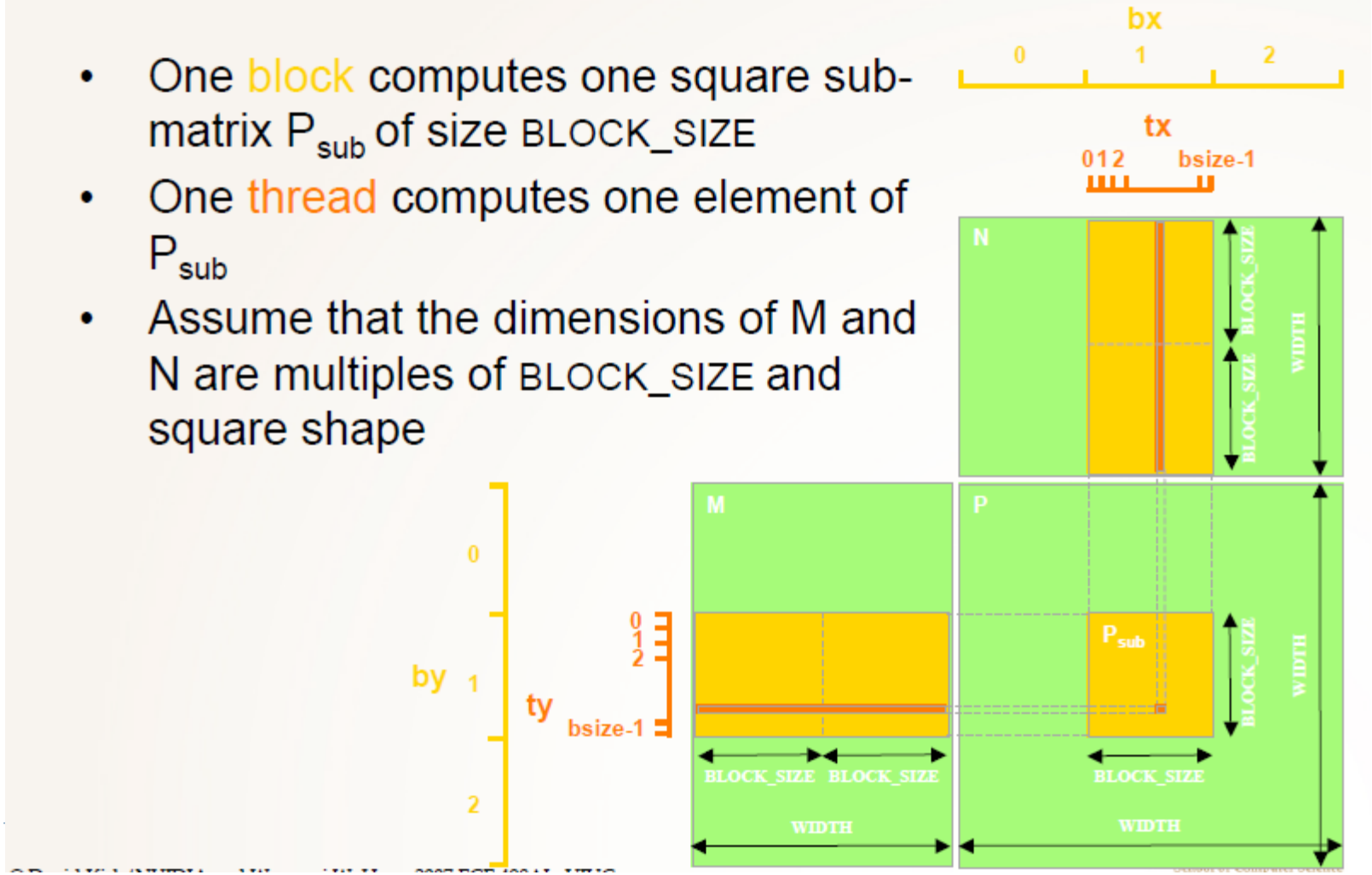
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * 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;

}
```

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code should run at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



Tiled Matrix Multiply

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape



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

// 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;
}

// 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)
{
    // Load A and B to device memory
    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((void**)&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((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&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, d_C.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)
{
    // 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);
}
```

Sumber

- NVIDIA Cuda Programming Guide