Introduction to Computação GPU

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

University of São Paulo / ICMC / SSC – São Carlos
Laboratory of Distributed Systems and Concurrent Programming

Slides based on Patterson, D.A., Hennessy, J.L. Computer organization and design : the hardware/software interface, 5th ed., Elsevier, Amsterdam, 2014,
and
Kirk, D.B., Hwu, W.W., Programming Massively Parallel Processors: a hands-on approach. 2nd ed., Morgan Kaufman, NVIDIA, 2013.

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP

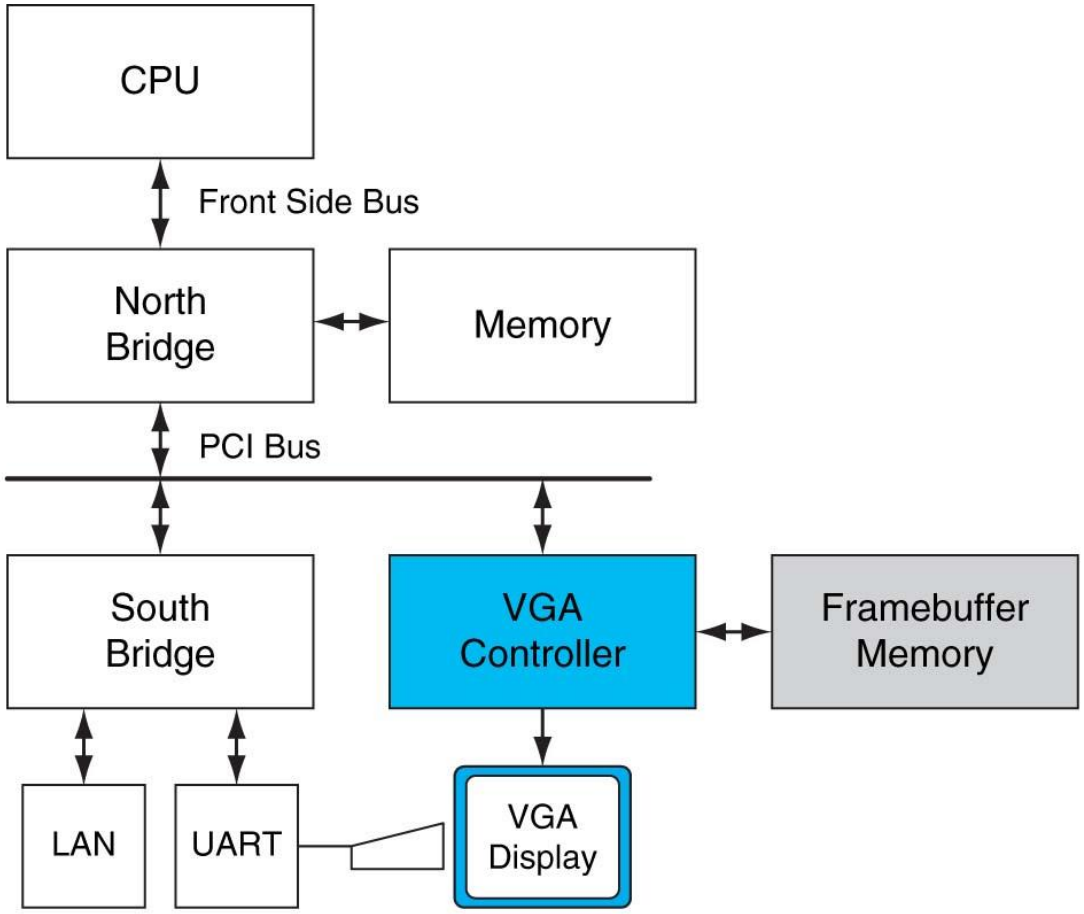


FIGURE A.2.1 Historical PC. VGA controller drives graphics display from framebuffer memory.

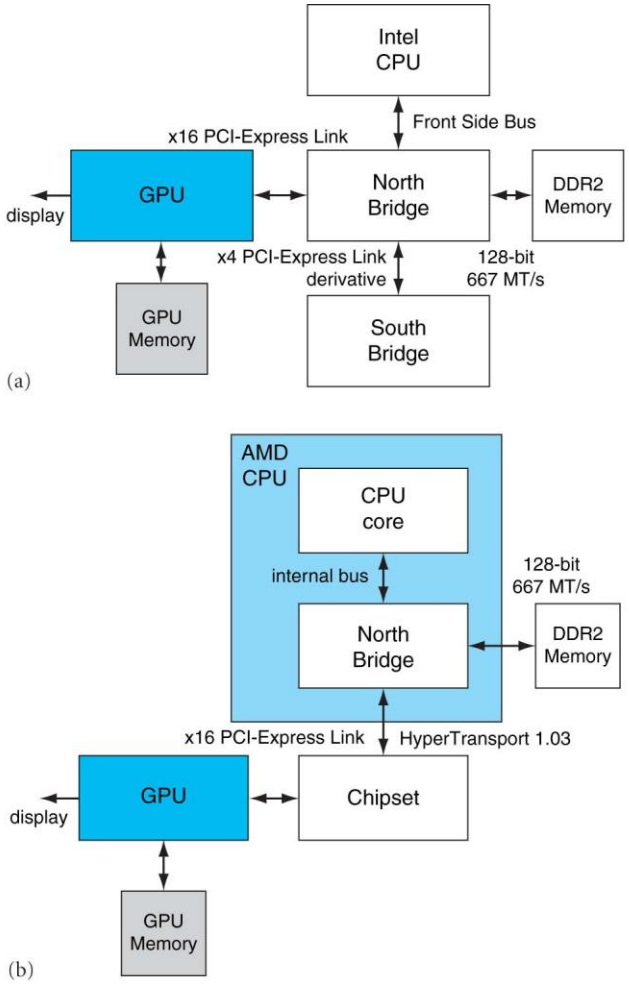


FIGURE A.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure.

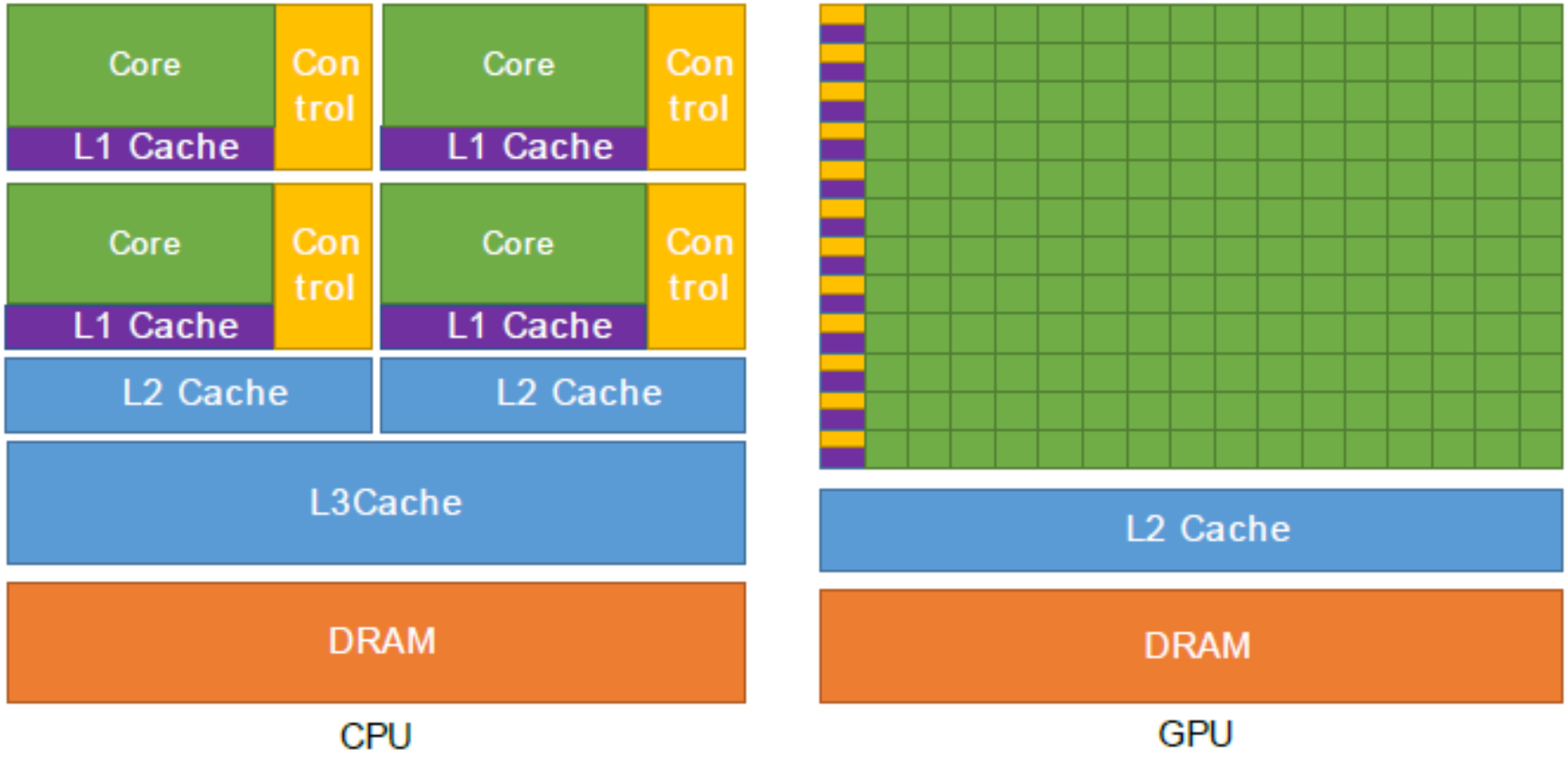


Figure 1 The GPU Devotes More Transistors to Data Processing

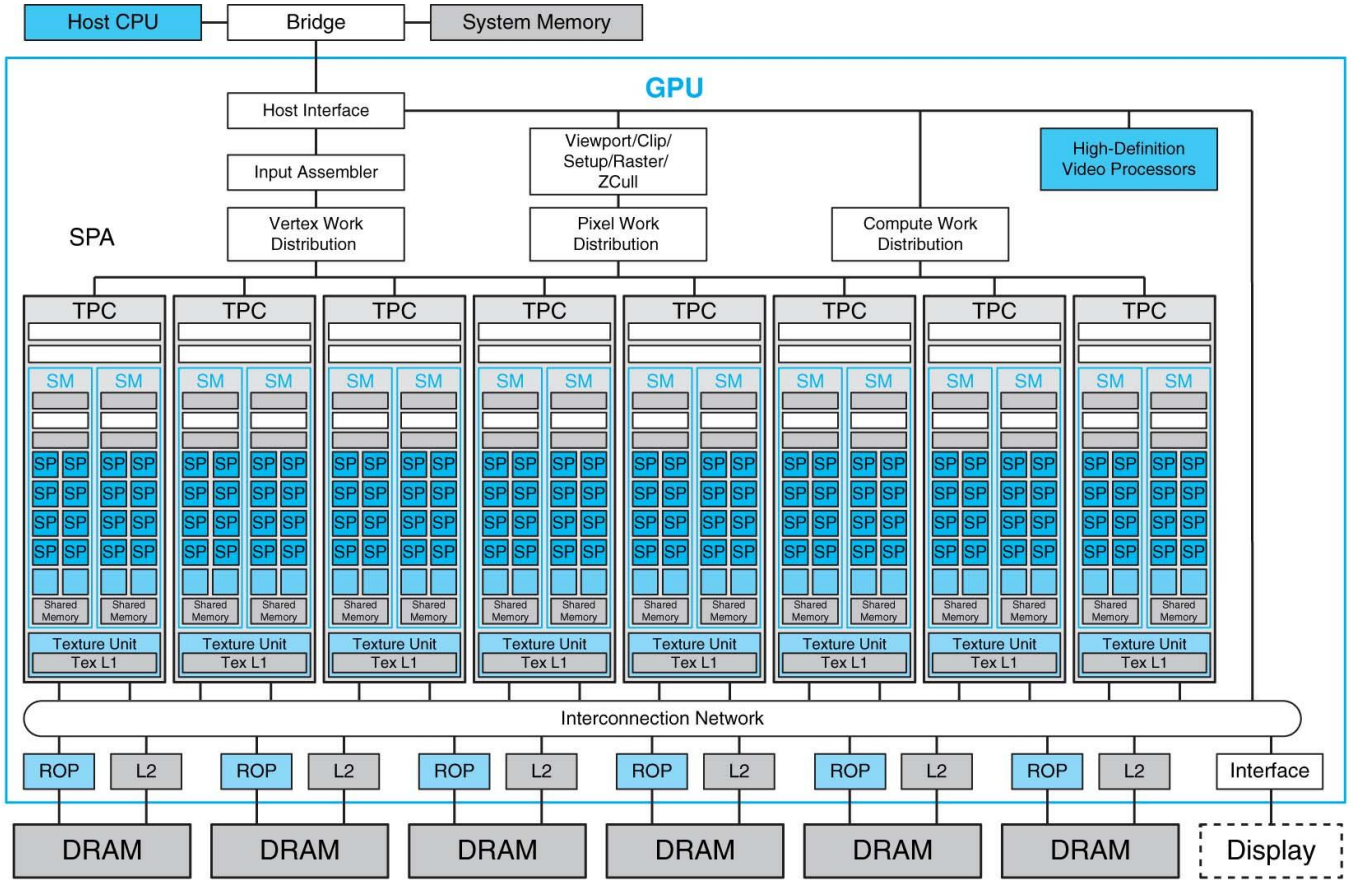


FIGURE A.7.1 NVIDIA Tesla unified graphics and computing GPU architecture. This GeForce 8800 has 128 streaming processor (SP) cores in 16 streaming multiprocessors (SM), arranged in eight texture/processor clusters (TPC). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.

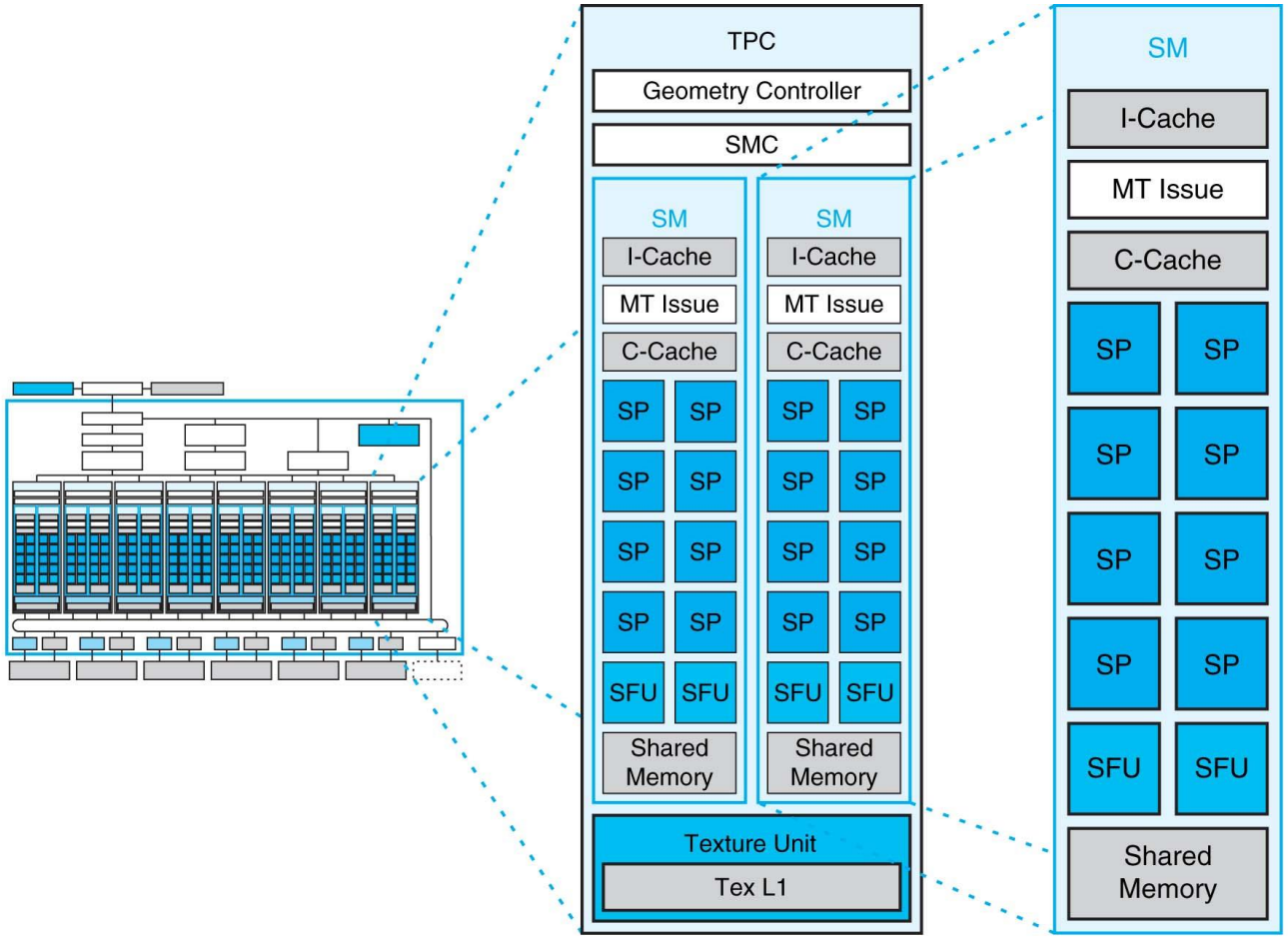


FIGURE A.7.2 Texture/processor cluster (TPC) and a streaming multiprocessor (SM). Each SM has eight streaming processor (SP) cores, two SFUs, and a shared memory.

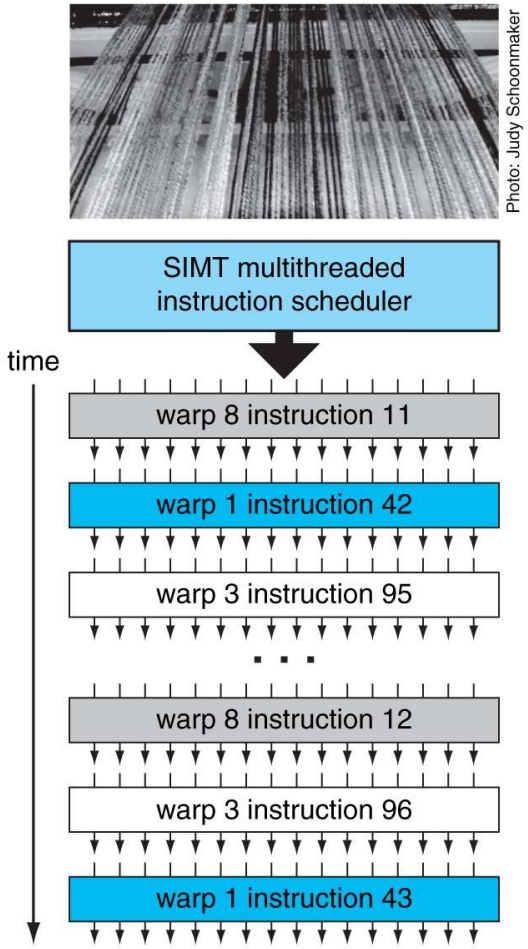
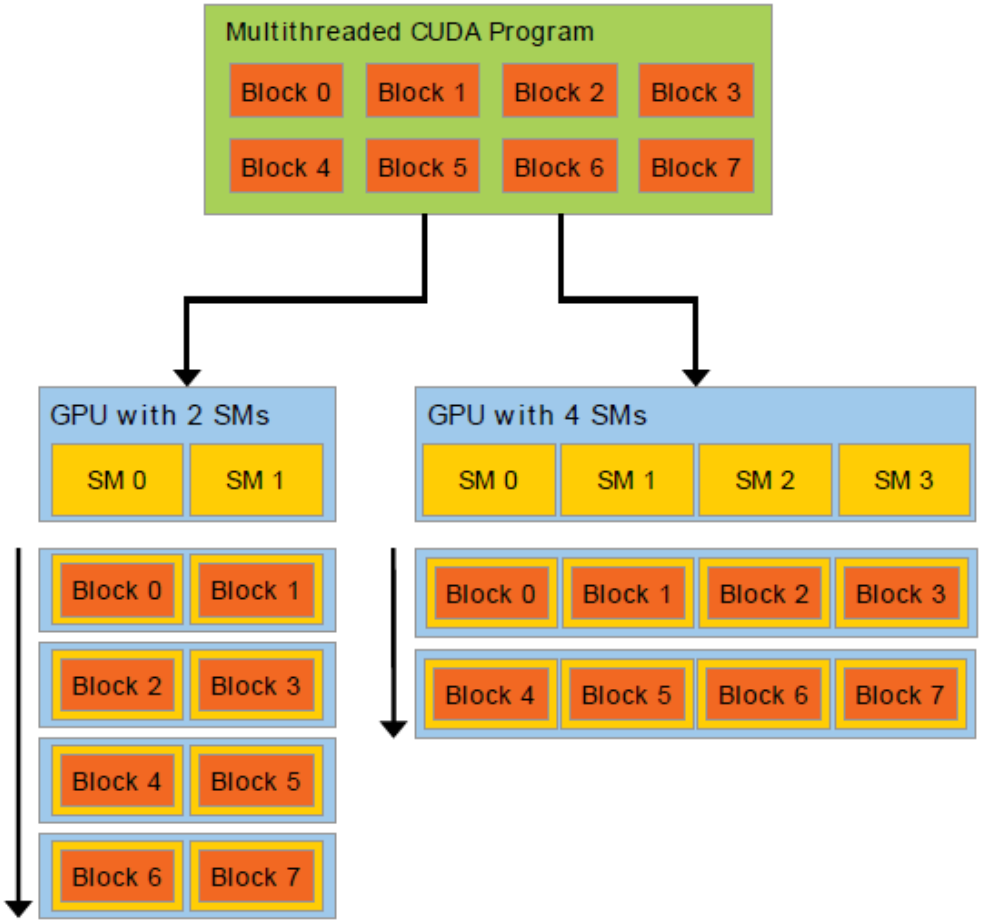


FIGURE A.4.2 SIMT multithreaded warp scheduling. The scheduler selects a ready warp and issues an instruction synchronously to the parallel threads composing the warp. Because warps are independent, the scheduler may select a different warp each time.



A GPU is built around an array of Streaming Multiprocessors (SMs) (see Hardware Implementation for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Figure 3 Automatic Scalability

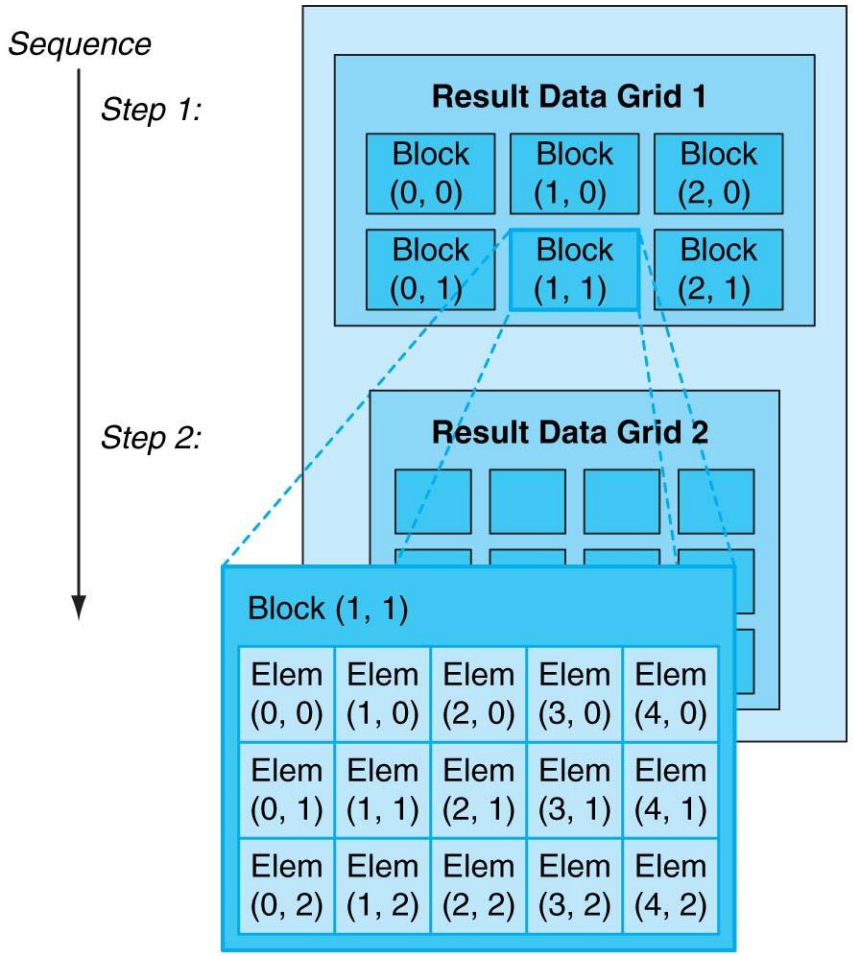


FIGURE A.3.3 Decomposing result data into a grid of blocks of elements to be computed in parallel.

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIGURE A.3.4 Sequential code (top) in C versus parallel code (bottom) in CUDA for SAXPY (see Chapter 7). CUDA parallel threads replace the C serial loop—each thread computes the same result as one loop iteration. The parallel code computes *n* results with *n* threads organized in blocks of 256 threads.

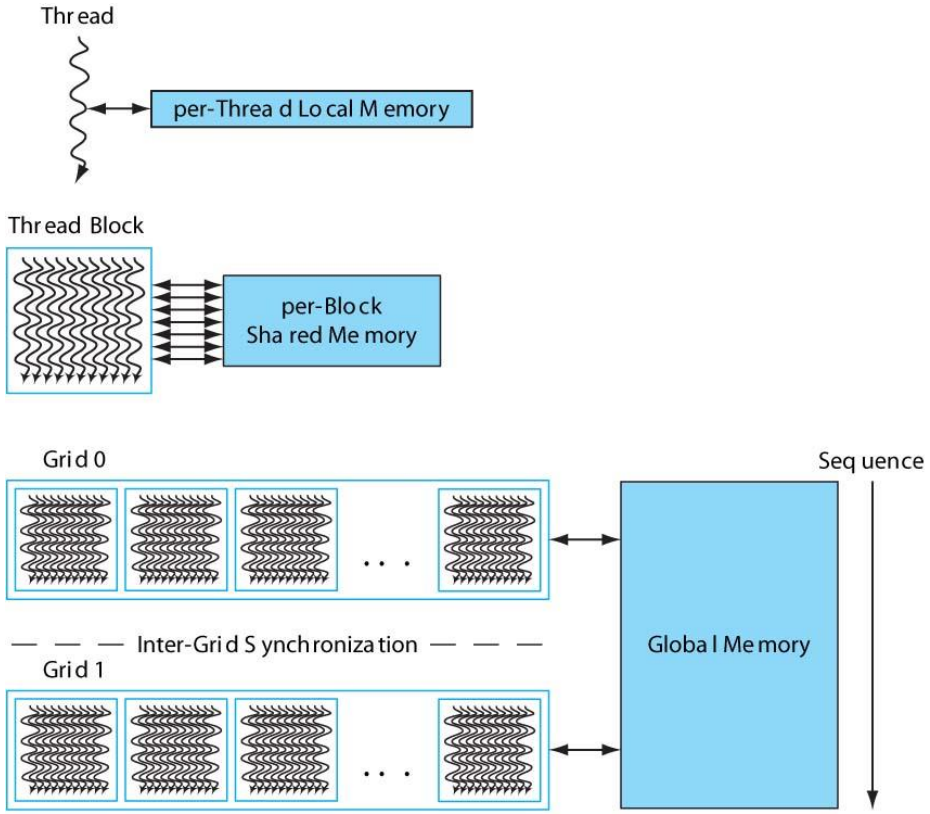


FIGURE A.3.5 Nested granularity levels- thread, thread block, and Book- have corresponding memory sharing levels - loc al. shared and global. Per -thread localmemory is private to the thread. Per block shared memory is shared by all the thread of the block. Per - application global memory is shared by all thread.

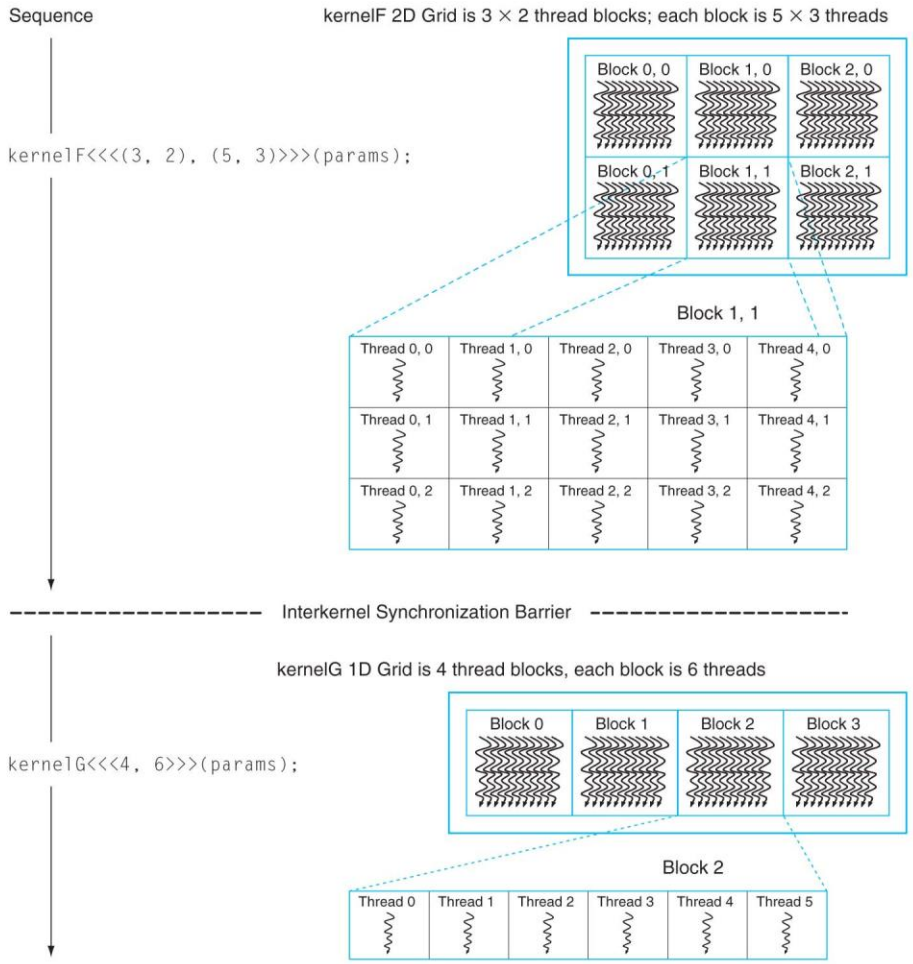


FIGURE A.3.6 Sequence of kernel *F* instantiated on a 2D grid of 2D thread blocks, an interkernel synchronization barrier, followed by kernel *G* on a 1D grid of 1D thread blocks.

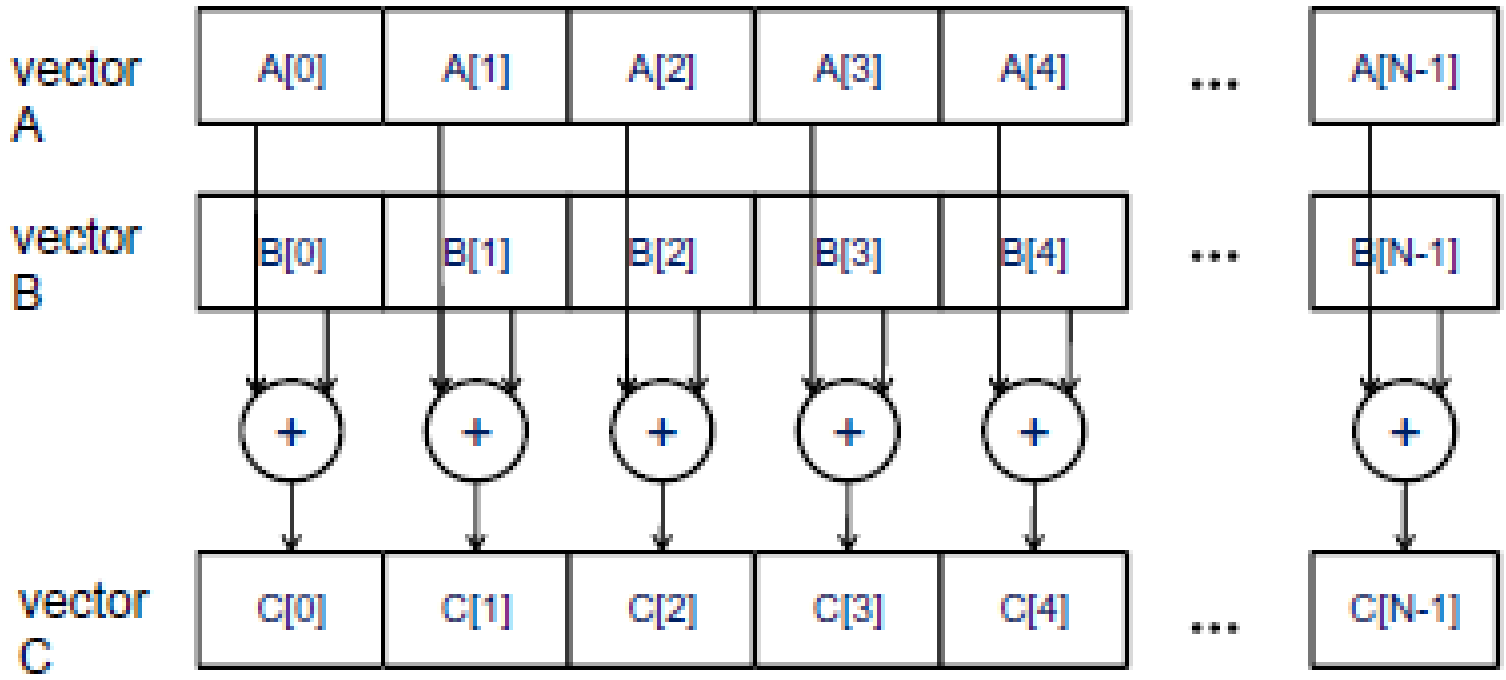


FIGURE 3.1

Data parallelism in vector addition.

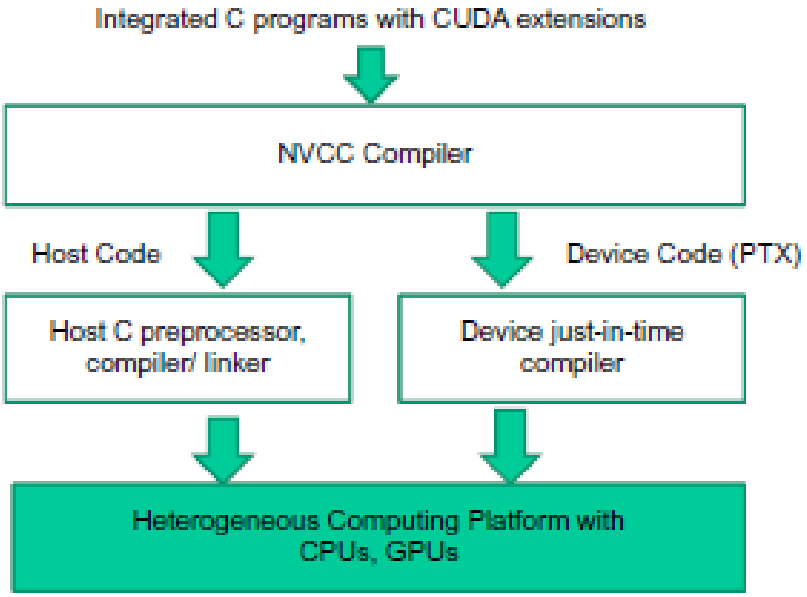


FIGURE 3.2

Overview of the compilation process of a CUDA program.

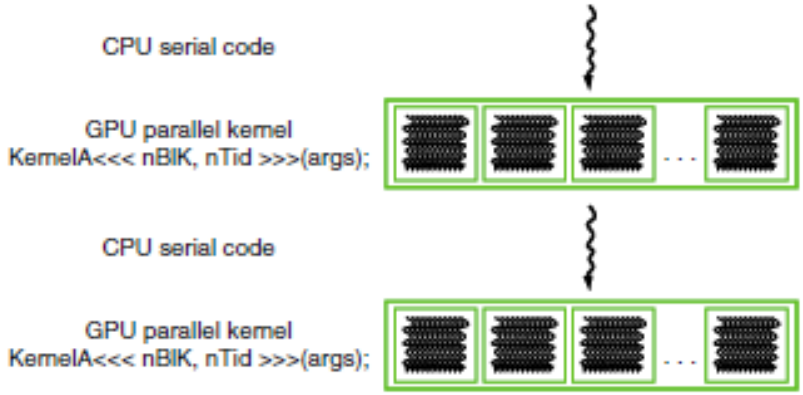


FIGURE 3.3

Execution of a CUDA program.

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

FIGURE 3.4

A simple traditional vector addition C code example.

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
   int size = n* sizeof(float);
   float *A_d, *B_d, *C_d;
   …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

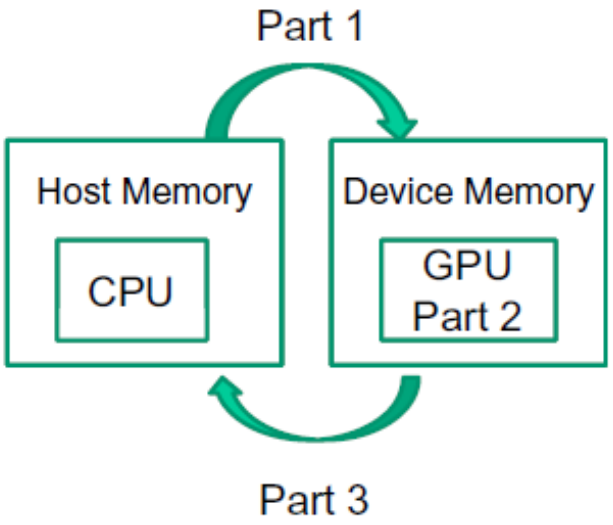


FIGURE 3.5

Outline of a revised `vecAdd()` function that moves the work to a device.

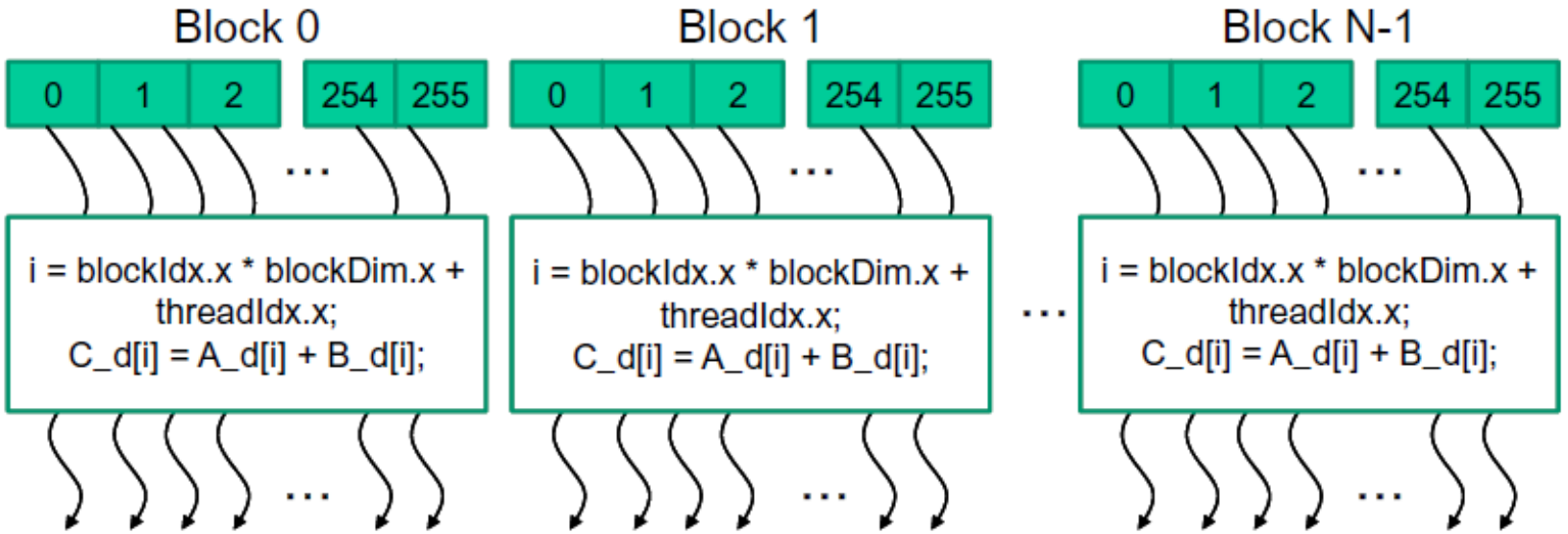


FIGURE 3.10

All threads in a grid execute the same kernel code.

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

FIGURE 3.12

CUDA C keywords for function declaration.