

CS 6023 - GPU Programming

Introduction to Parallel Programming Frameworks

31/01/2019

Agenda

- Challenges and options for parallel programming
- Basic structure and syntax of a CUDA program

Acknowledgements

Nvidia teaching kit for Deep Learning

What are the ways in which GPU architecture differs from CPU architecture?

- Focus on latency vs throughput
- Have many small cores vs few complicated cores
- Support extreme SIMD (32 or more ALUs per F/D) vs few (4 or 8)
- Partition execution contexts to support many threads with very small context-switching costs
- Support for much higher memory bandwidth

Challenges with parallel programming

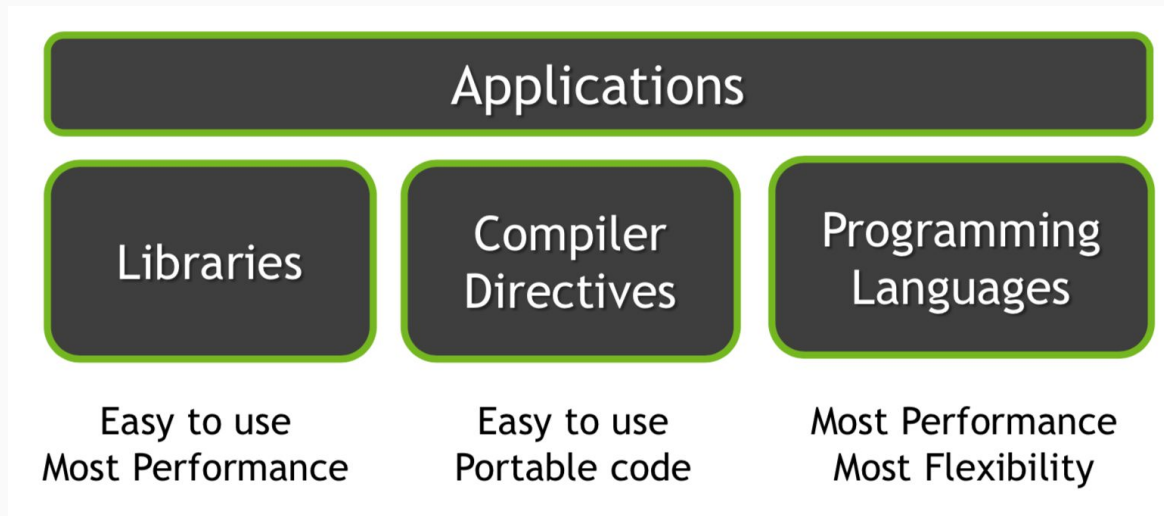
*If **performance** is not a constraint, then do not do parallel programming*

- Designing parallel programs with the same level of complexity as sequential programs (i.e., limit overheads)
- Avoiding being memory bound by increasing operational intensity by optimizing memory fetching and reuse
- Dealing with erratic, variable, and unstructured input data characteristics
- Efficiently converting recurrent algorithms to parallel versions
- Avoiding race conditions, synchronization errors
- Efficiently utilizing the cores/resources (load balancing or efficient distribution of work)

Quiz 1 next week

- 14th Feb (as per Institute calendar) - usual class time slot
- Syllabus: till whatever is covered until tomorrow (8th Feb)
- Multiple choice quiz
- +1 for correct answers and -0.5 for wrong answers
- Closed book
- One A4 sheet paper allowed

How to accelerate applications



- Ease of use: Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- “Drop-in”: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications

GPU accelerated libraries

Linear Algebra

FFT, BLAS,
SPARSE, Matrix



CULA|tools



CUSP

Numerical & Math

RAND, Statistics



ArrayFire



Data Struct. & AI

Sort, Scan, Zero Sum



Visual Processing

Image & Video



Deep Learning



Thrust library - Quick example

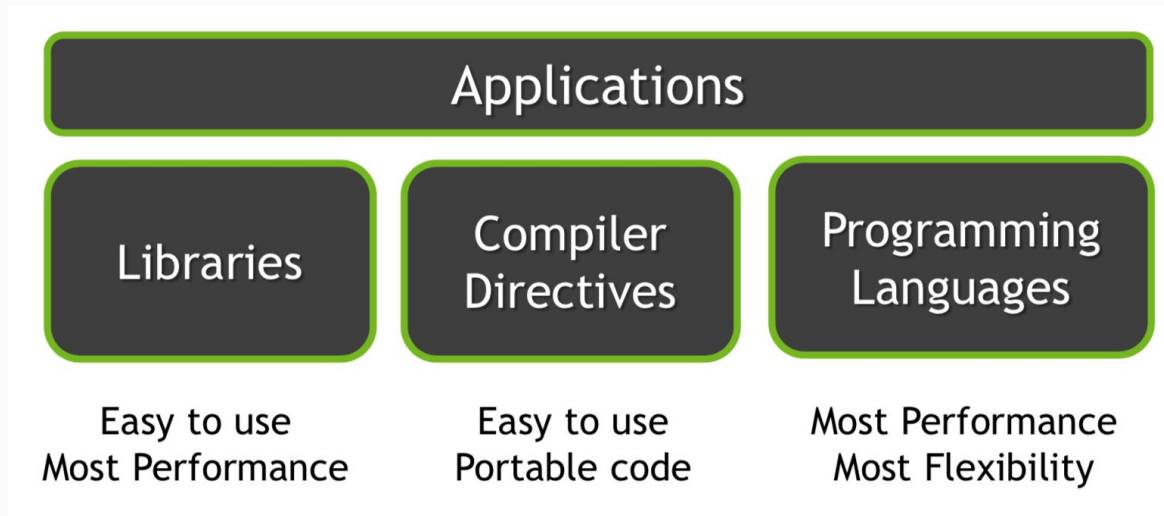
- **Thrust** 2010: Vector containers (think generics) on host and device
- Also efficient algorithms that operate on vectors

```
//initialize random values on host
thrust::host_vector<int>h_vec(1000);
thrust::generate(h_vec.begin(),h_vec.end(),rand);
//copy values to device
thrust::device_vector<int>d_vec=h_vec;
//compute sum on host
int h_sum=thrust::reduce(h_vec.begin(),h_vec.end());
//compute sum on device
int d_sum=thrust::reduce(d_vec.begin(),d_vec.end());
```

- **MPI:** Message Passing Interface for efficient communication in distributed memory (single NUMA machine or multiple machines)

```
/** send_recv.c */
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv) {
    int my_rank, numbertoreceive, numbertosend=77;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank==0){
        MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        printf("Number received is: %d\n", numbertoreceive);
    }
    else if(my_rank == 1)
        MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

How to accelerate applications



Compiler directives

- Ease of use: Compiler takes care of details of parallelism management and data movement
- Portable: The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- Uncertain: Performance of code can vary across compiler versions

OpenMP: Quick example

- **OpenMP** (2005): For shared memory multiprocessors.
 - C/C++, Fortran. Thread based
 - Explicit parallelism (programmer adds compiler directives)

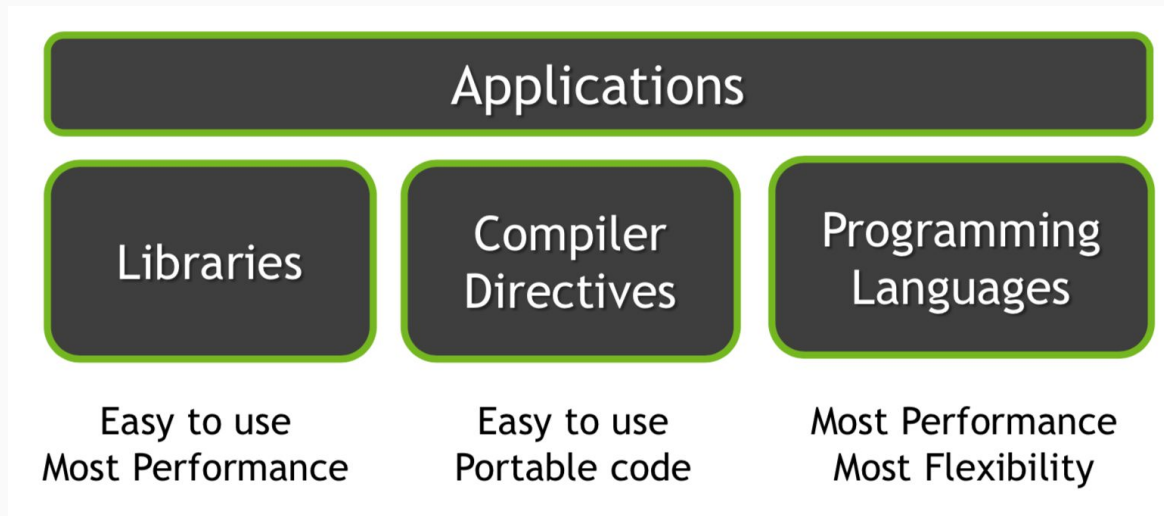
```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main() {
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("Hello (%d)\n", ID);
        printf(" world (%d)\n", ID);
    }
}
```

Set # of threads for OpenMP
In csh setenv OMP_NUM_THREADS 8

Compile: g++ -fopenmp hello.c
Run: ./a.out

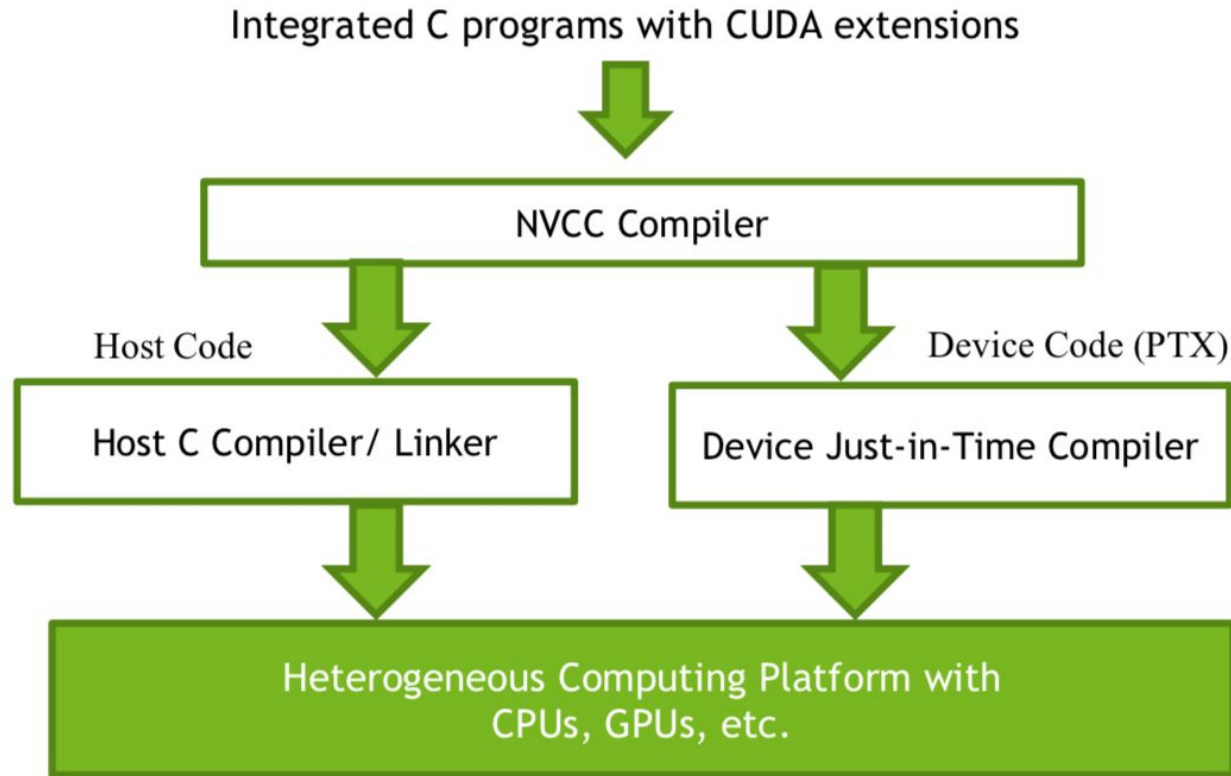
How to accelerate applications



Programming languages

- Performance: Programmer has best control of parallelism and data movement
- Flexible: The computation does not need to fit into a limited set of library patterns or directive types
- Verbose: The programmer often needs to express more details

Compilation of CUDA C



- **Host** - CPU executing ANSI C
- **Device** - GPU (highly parallel) executing extended ANSI C
- Host and device have separate memories
- Single **CUDA program** contains both host and device code
- **Kernel** - Data parallel function
 - Runs on the device with (large number of) lightweight threads
 - Creation, scheduling, and management of threads on device
 - Think of kernel as motivated by operations on the graphics pipeline

Basic kernel

```
// Kernel Definition
```

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

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

Execution of CUDA program

CPU Serial Code

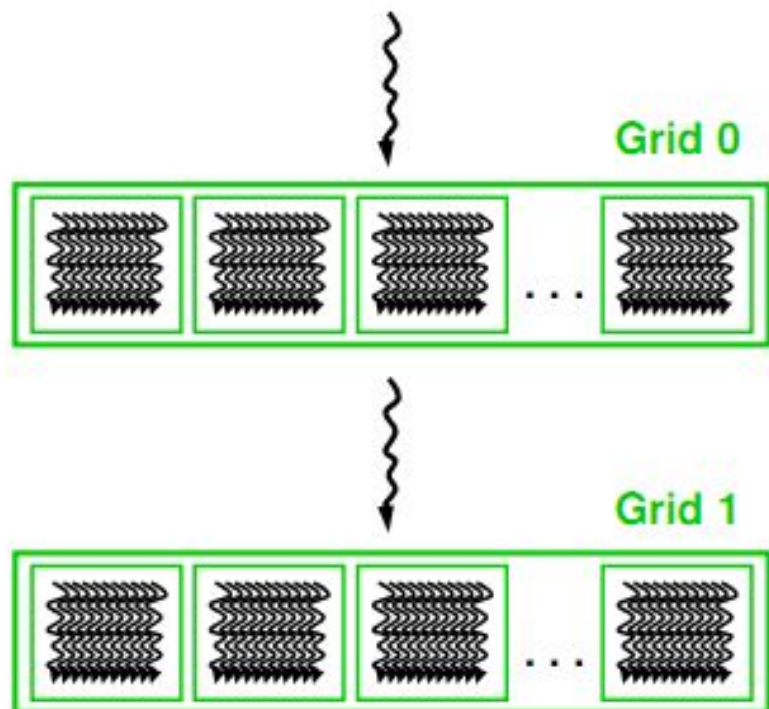
GPU Parallel Kernel

```
KernelA<<< nBlk, nTid >>>(args);
```

CPU Serial Code

GPU Parallel Kernel

```
KernelB<<< nBlk, nTid >>>(args);
```



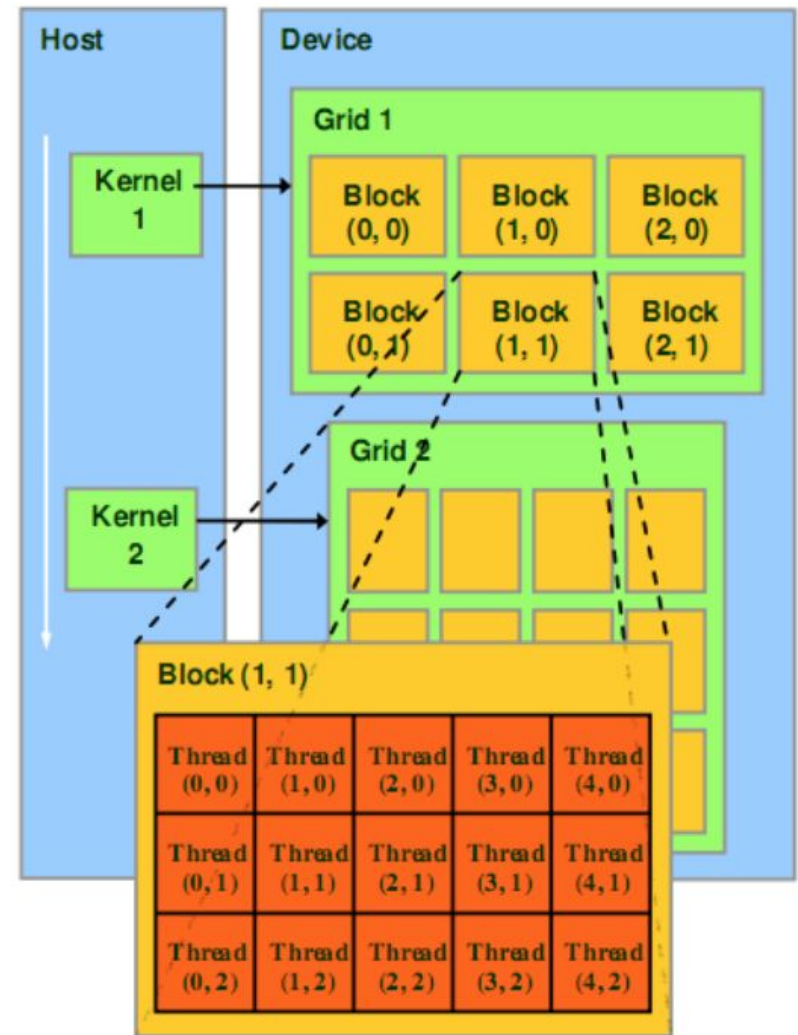
- **Thread** - Each independent unit of work (one invocation of the kernel)
- **Block** - A collection of threads
 - Arranged in 1D, 2D, or 3D
 - Unit defined by the ability of threads to
 - share memory
 - synchronize
- **Grid** - A collection of blocks which all share the same kernel
 - Arranged in 1D or 2D
 - Each block in a grid has the same number of threads

Thread hierarchy

Kernel function corresponds to a grid
which is a collection of blocks and
each block is a collection of threads

Why these multiple hierarchies?

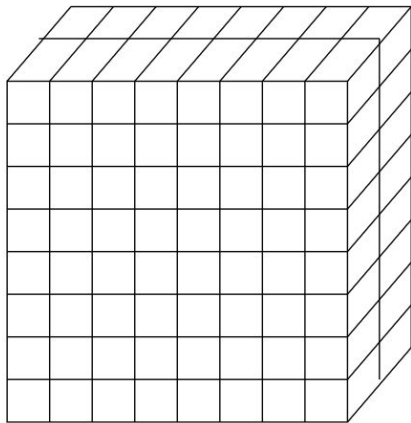
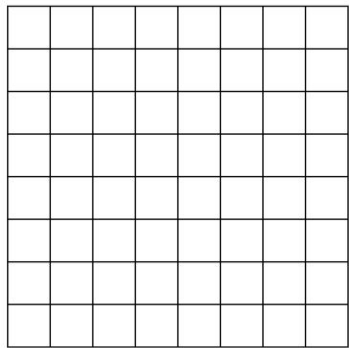
*To map to typical high-dimensional
data structures and multiple loops in
algorithms*



Thread hierarchy

Data can be multidimensional
vector, matrix, volume/ tensor

Arrange threads in a block in 1D, 2D, 3D



Loops can be multidimensional
6 nested loops for deconvolution
Arrange blocks in a grid in 1D, 2D

```
1: procedure DECONVOLUTION
2:   for  $i_c = 0$  to  $I_C - 1$  do
3:     for  $i_h = 0$  to  $I_H - 1$  do
4:       for  $i_w = 0$  to  $I_W - 1$  do
5:         for  $o_c = 0$  to  $O_C - 1$  do
6:           for  $k_h = 0$  to  $K - 1$  do
7:             for  $k_w = 0$  to  $K - 1$  do
8:                $o_h \leftarrow S \times i_h + k_h - P$ 
9:                $o_w \leftarrow S \times i_w + k_w - P$ 
10:               $\text{out}[o_c][o_h][o_w] \leftarrow (\text{in}[i_c][i_h][i_w]$ 
                 $\times \text{kernel}[o_c][i_c][k_h][k_w])$ 
```

Thread hierarchy

```
// Kernel Definition
__global__ void matrixAddition(const float* A, const float* B, float*
C)
{
    int idx = threadIdx.y * blockDim.x + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}

int main()
{
    // ....
    // Kernel invocation with one block of N * N * 1 threads
    int blocks = 1;
    dim3 threadsPerBlock(N, N);
    vectorAdd<<<blocks, threadsPerBlock>>>(A, B, C);
}
```

Thread hierarchy

```
// Kernel Definition
__global__ void vectorAdd(const float* A, const float* B, float* C, int N)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = y * N + x;
    C[idx] = A[idx] + B[idx];
}

int main()
{
    // ....
    // Kernel invocation with computed configuration
    dim3 threadsPerBlock(16, 16);
    dim3 blocks(N / threadPerBlock.x, N / threadPerBlock.y);
    vectorAdd<<<blocks, threadsPerBlock>>>(A, B, C);
}
```


Thread Hierarchy - Example

Matrix addition of two matrices 256 x 256

threadIdx.x / y		
blockDim.x / y		
blockIdx.x / y		

```
int x = blockIdx.x * blockDim.x + threadIdx.x;  
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

Thread Hierarchy - Example

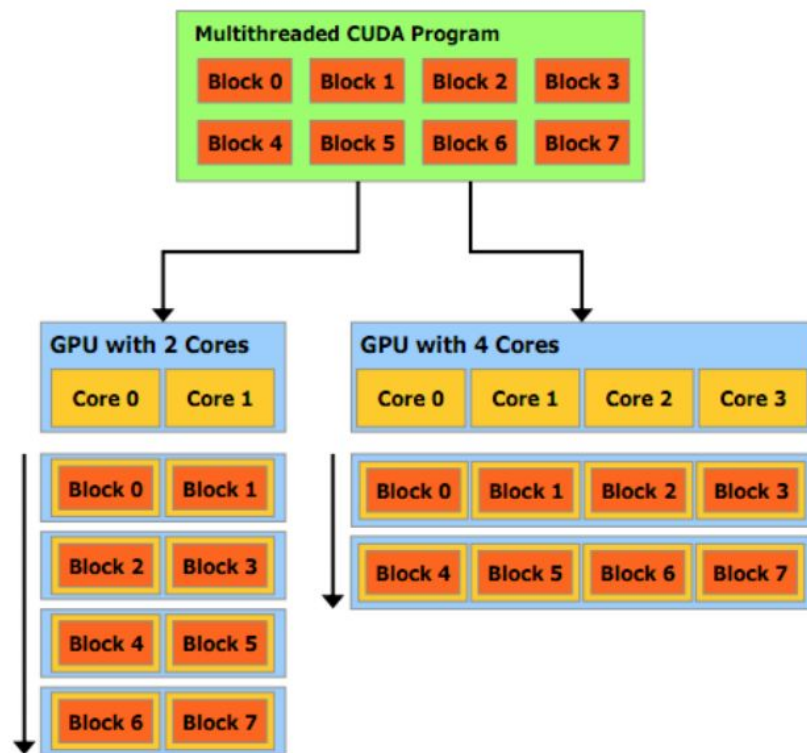
Matrix addition of two matrices 256 x 256

threadIdx.x / y	[0-15]	[0-127]
blockDim.x / y	16	128
blockIdx.x / y	[0-15]	[0-1]

Choosing a better configuration depends on hardware properties

```
int x = blockIdx.x * blockDim.x + threadIdx.x;  
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

But the implementation on cores is a hardware decision



A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability