CME213/ME339
Lecture 2

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Multicore

Multicore processors. Example: Intel Core i7. Sandy Bridge 3930K 3.8 GHz. 243 Gflops.

Note: the multicore/manycore terminology is not set in stone and its precise meaning varies.

Out-of-order, multiple-instruction issue processor implementing the full x86 instruction set. 4 to 6 cores on each processor operate in parallel. The microprocessor supports hyperthreading with two hardware threads (more parallelism with the "same" hardware). The processor is still designed to maximize the execution speed of sequential programs but parallel codes are required for peak performance.

Manycore

Manycore processors. Focuses more on the execution throughput of parallel applications.

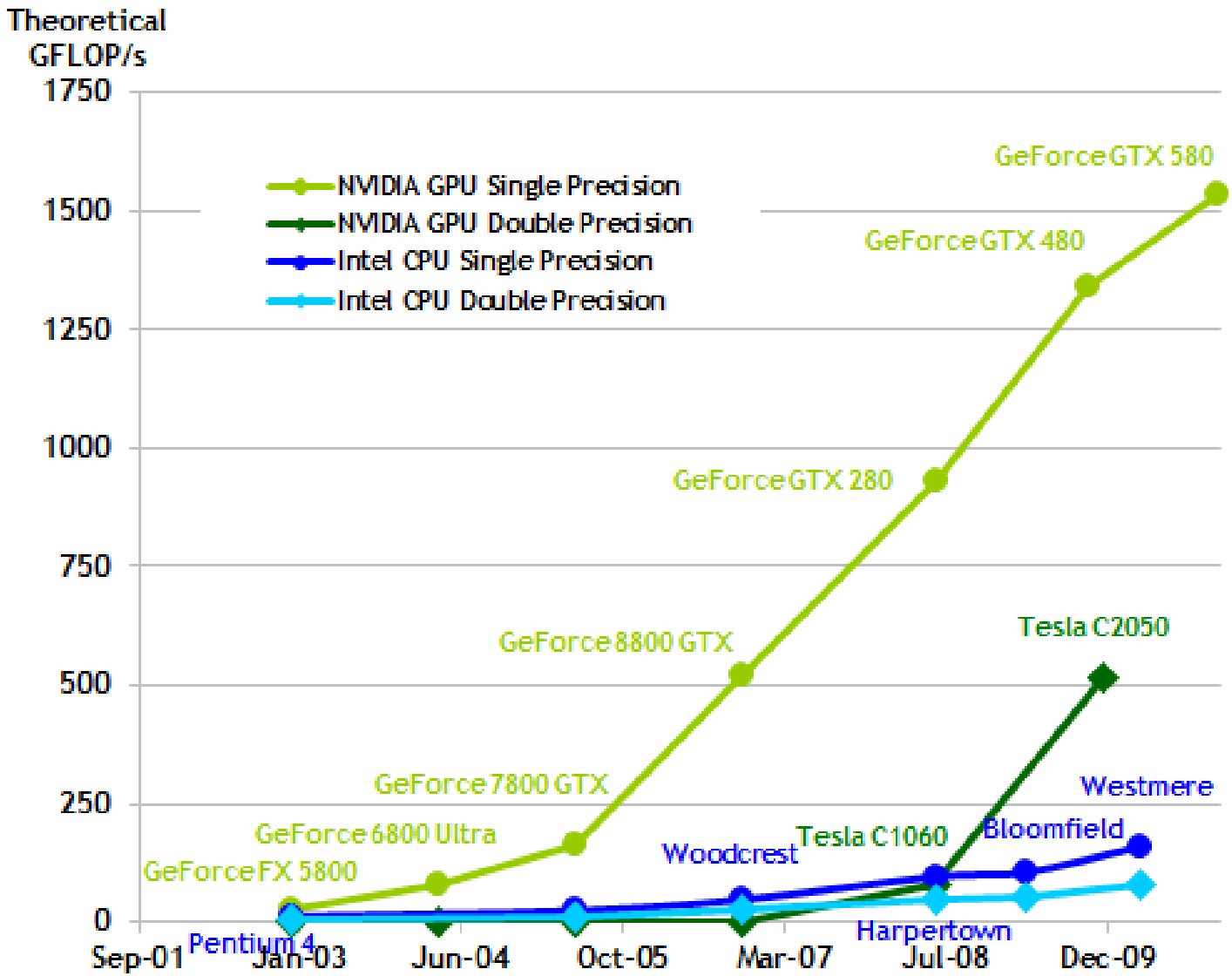
Large number of much smaller cores.

Once again, the number of cores approximately doubles with each generation.

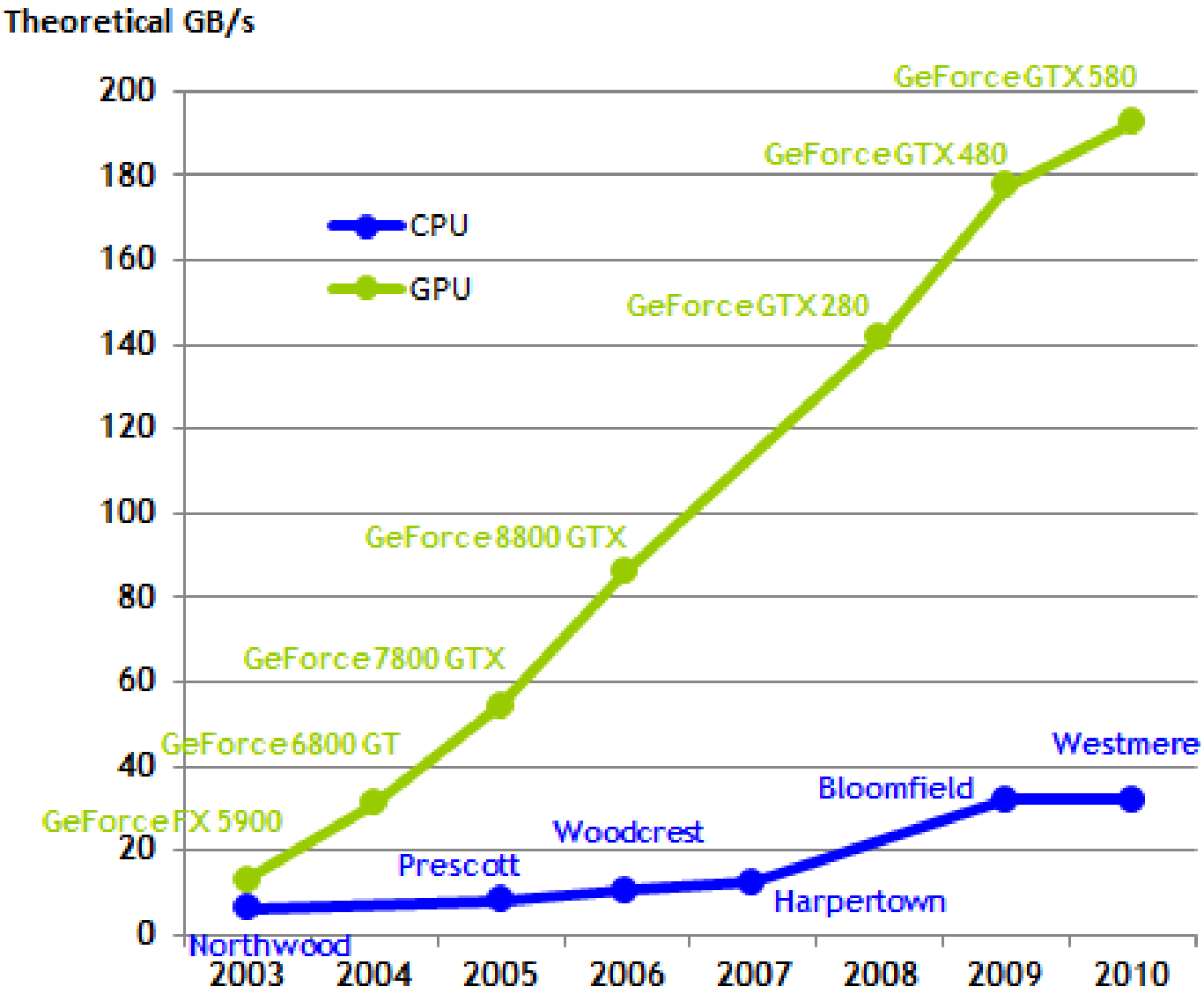
Current exemplar: NVIDIA Tesla M2090 graphics processing unit (GPU) with 512 cores, each of which is a heavily multithreaded, in-order, single-instruction issue processor.

Peak performance: 1,331 Gflops. Maximum number of threads: 24,576. Memory bandwidth: 177 Gbytes/sec. Double precision performance: 665 Gflops. Memory: 6 Gbytes.

Floating-point operations per second

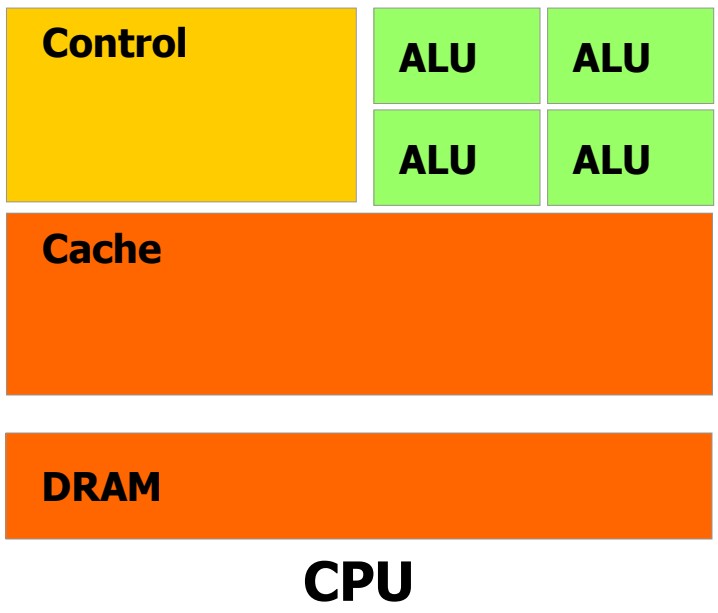


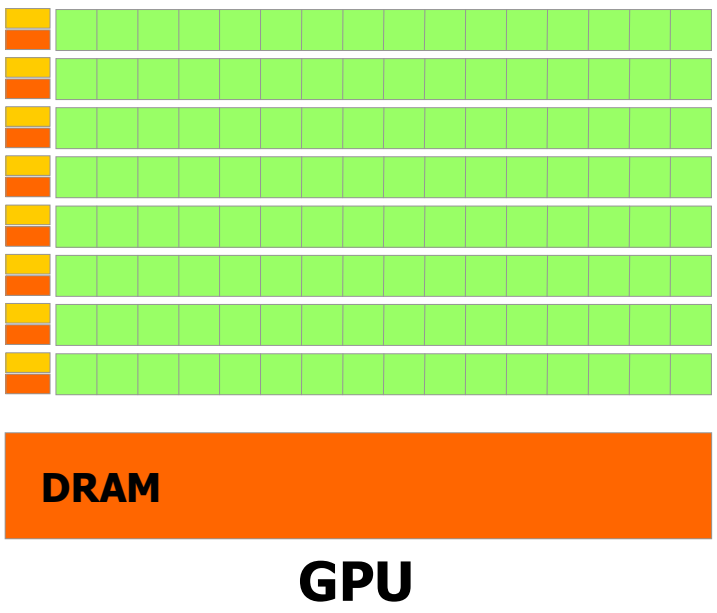
Memory bandwidth



Design philosophy

The GPU devotes more transistors to data processing.





Cost of concurrency

This performance has a cost:

- Applications increasingly need to be concurrent if they want to fully exploit modern hardware
- Efficiency and performance optimization is getting more, not less, important
- Programming languages and systems are increasingly forced to deal well with concurrency

Parallel languages

Many different programming models and languages are available.

This reflects the rapid evolution of technology.

Hopefully, at some point, in the future, we will converge towards a unified language, but it is not clear whether this is possible or whether it will ever happen.

MPI

Message Passing Interface (MPI): used for scalable cluster computing.

Although it can be used on shared memory parallel computers, it was designed with distributed memory computers in mind.

All data sharing and interaction is done through explicit message passing.

OpenMP

Applies to any hardware with a unified memory address space. In practice, it is often used for shared-memory multiprocessor systems.

Limitations: it has not been able to scale beyond a couple hundred computing nodes due to thread management overheads and cache coherence hardware requirements.

CUDA achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements.

OpenCL

Several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a standardized programming model called OpenCL.

We won't discuss OpenCL in this class.

OpenCL is still relatively in its infancy.

The level of programming constructs in OpenCL is at a lower level than CUDA and more tedious to use.

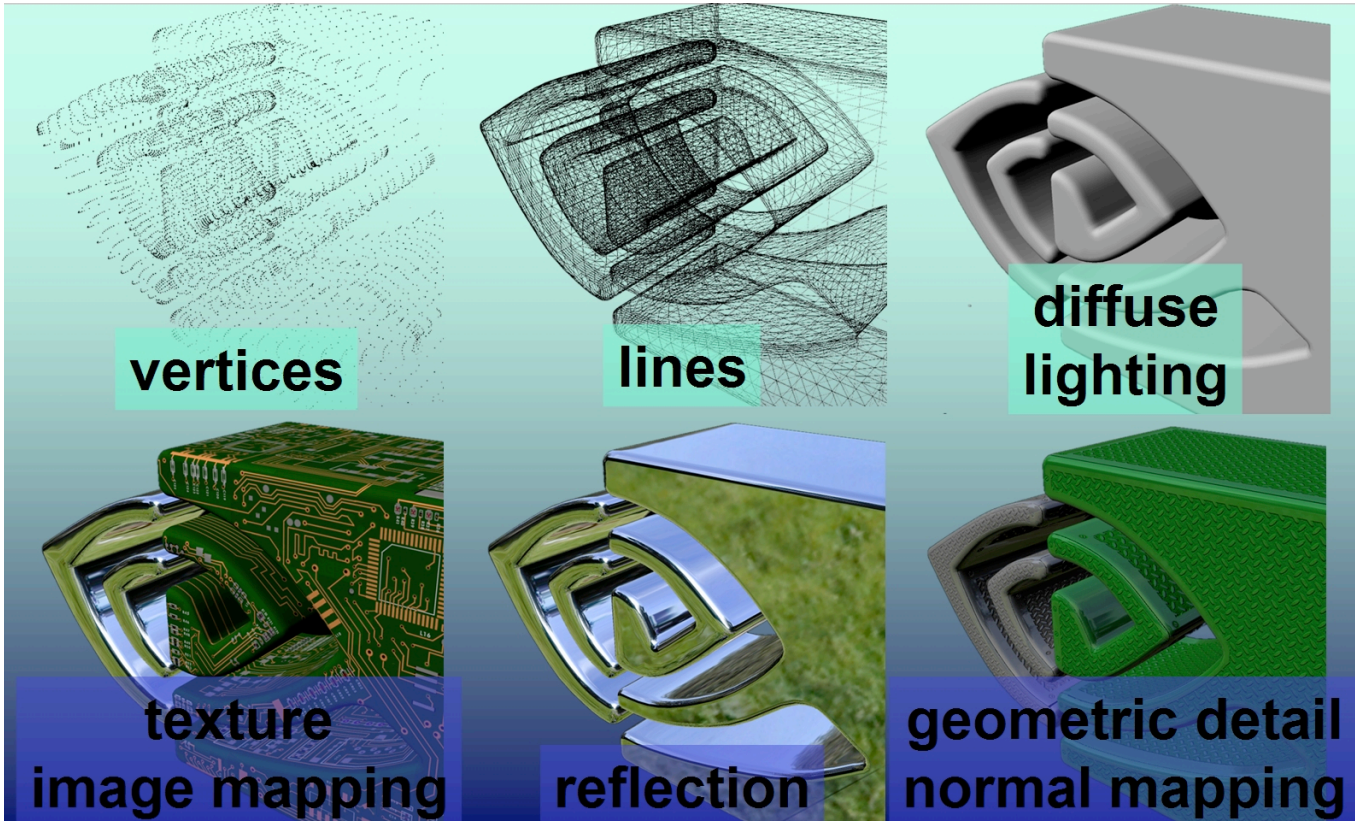
The speed achieved in an application expressed in OpenCL is still much lower than in CUDA on platforms that support both.

OpenCL is not as widespread as CUDA.

Short history of CUDA

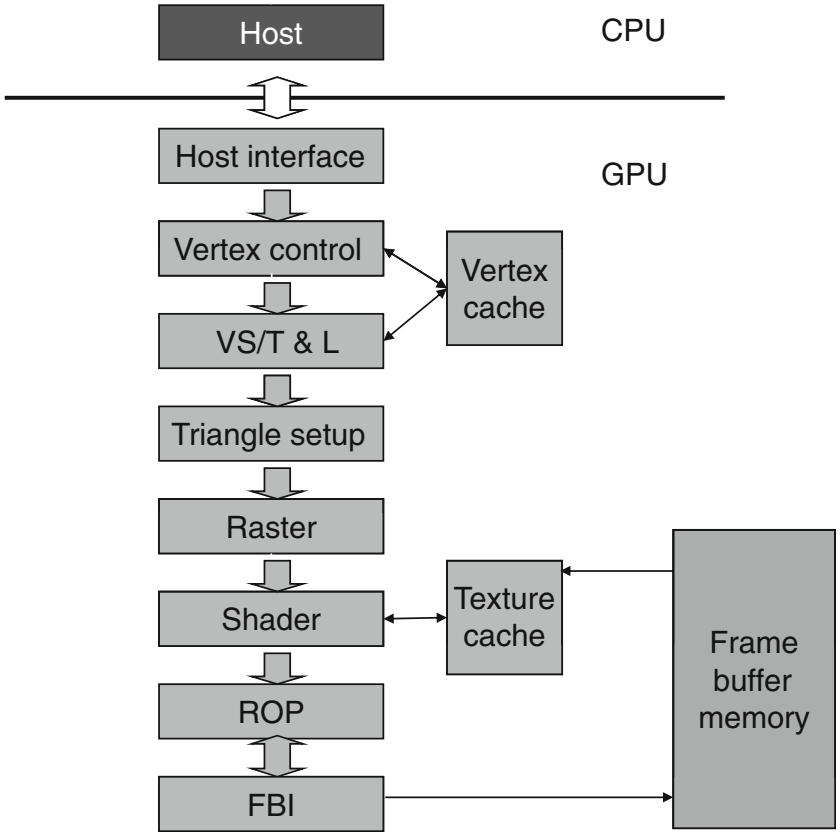
GPU computing started with gaming that requires sophisticated rendering.

Graphics operations: how to describe an object



The graphics pipeline

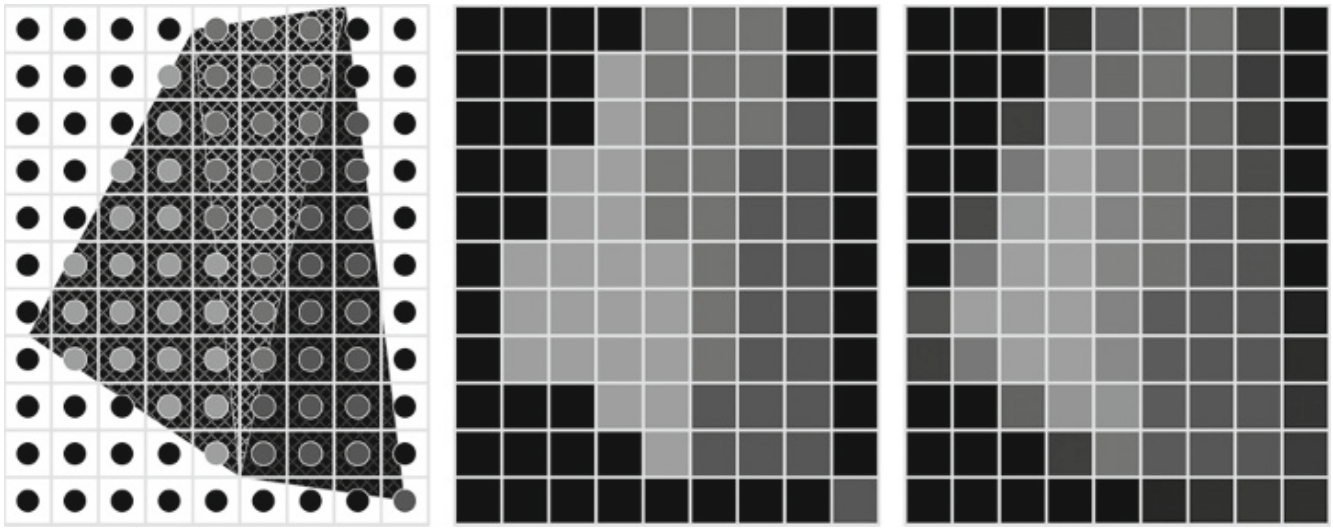
GPUs are based on a graphics pipeline: a technology to enable fast graphics. API (application programming interface): DirectX, OpenGL.



VS/T& L: vertex shading/transform and lightning. At the vertex level.
ROP: raster operations pipeline or render output unit.
FBI: (the other one) frame buffer interface

Graphics is compute intensive

Example: anti-aliasing of an image to improve the rendering.



Triangle geometry | Aliased | Antialiased

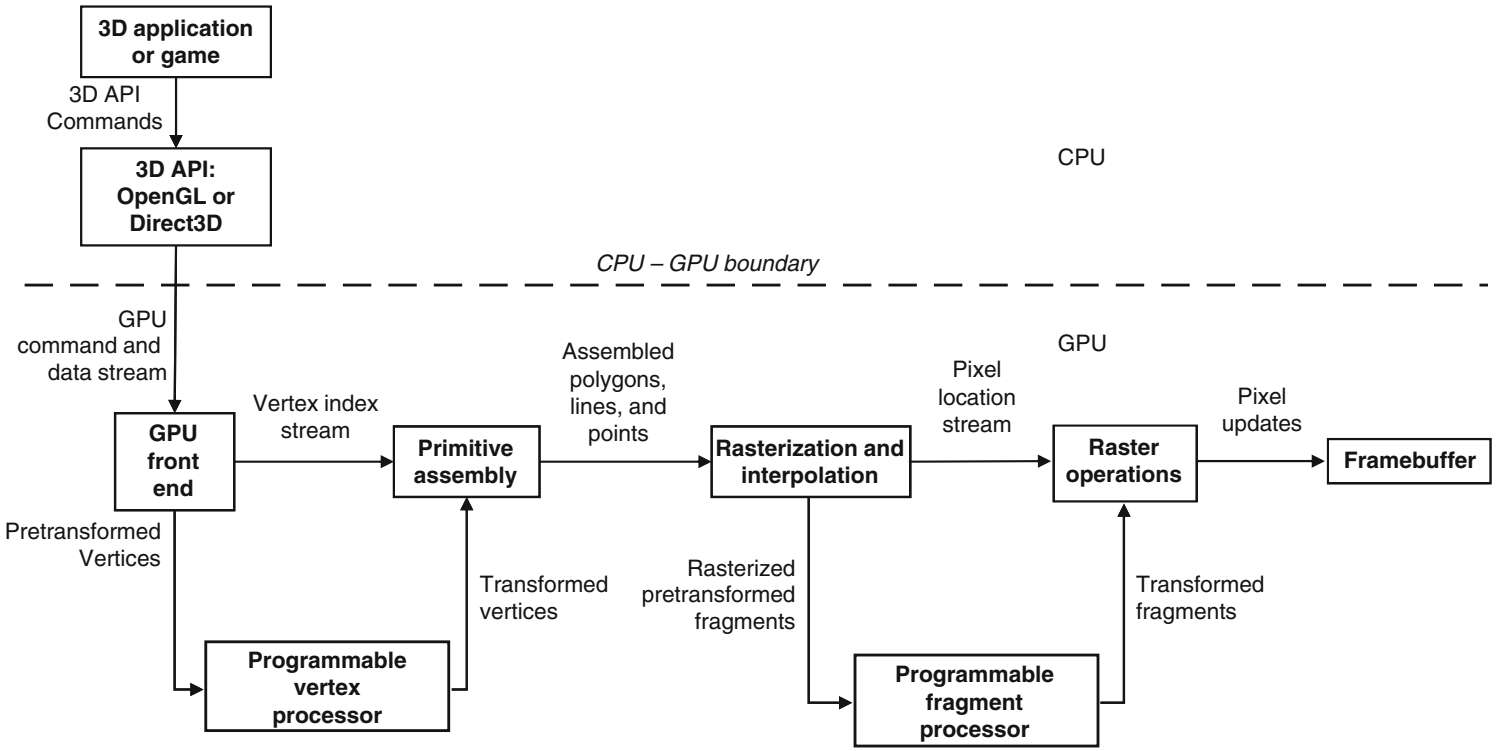
More possibilities

As games became more advanced, programmers wanted more freedom and options to render and shade a scene.

The pipeline became more complicated, to the point where NVIDIA/AMD started introducing general shader programmability.

Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation.

Shader programs calculate the floating-point red, green, blue, alpha color contribution to the rendered image at its pixel sample (x, y) image position.

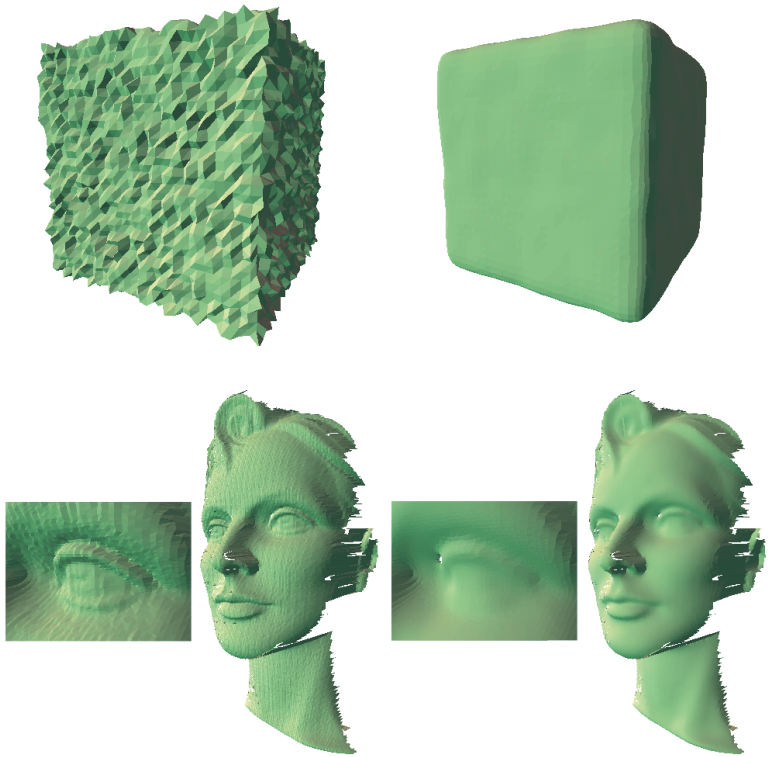


GPGPU

GPGPU was an intermediate step in which GPUs started being used for science and engineering problems.

To access the computational resources, a programmer had to cast his or her problem into native graphics operations so the computation could be launched through OpenGL or DirectX API calls.

To run many simultaneous instances of a compute function, for example, the computation had to be written as a pixel shader.

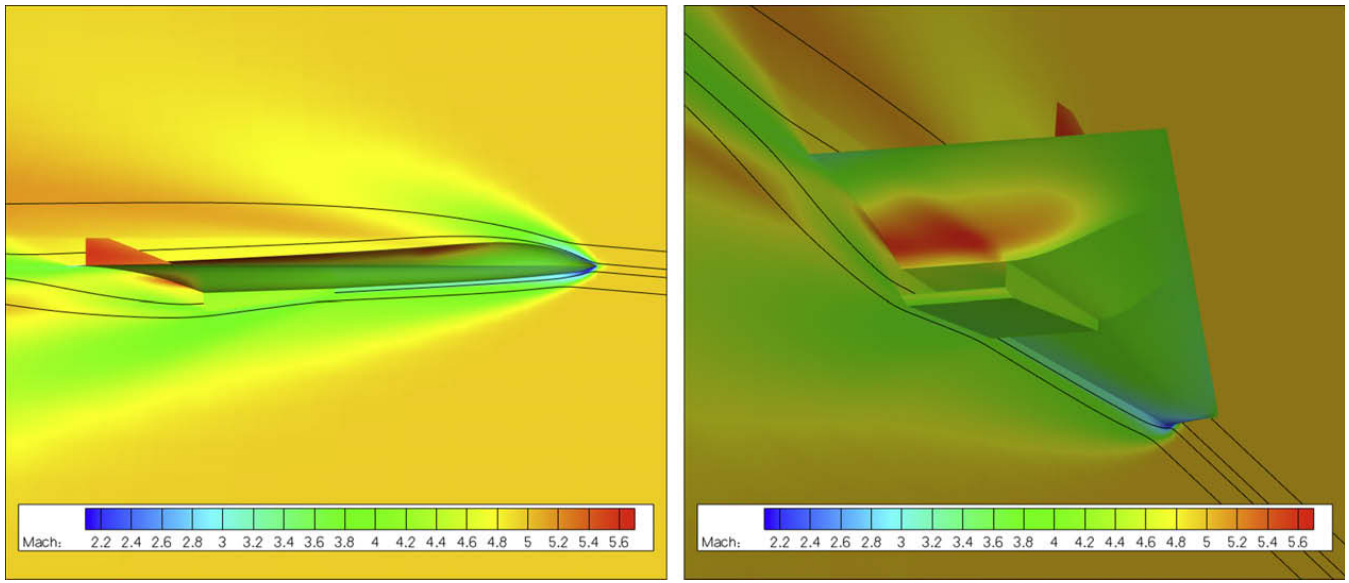


Sparse matrix-vector code running on a GPU for smoothing surfaces.
Bolz, Farmer, Grinspun, Schröder, 2003.

Shader languages: HLSL (High Level Shader Language, Microsoft), Cg (NVIDIA), GLSL (OpenGL shading language).

BrookGPU

A language, created at Stanford, that served as an earlier version of CUDA.

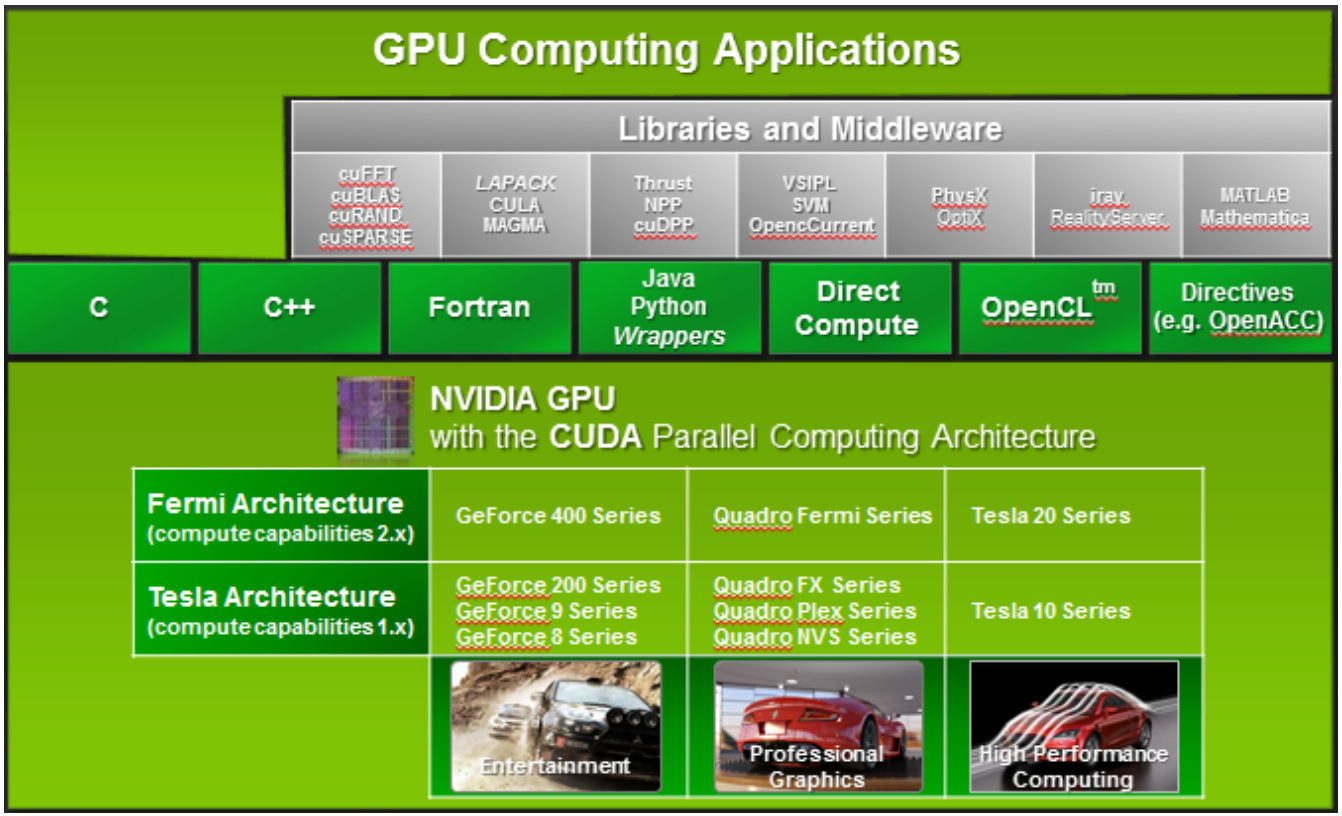


Mach number — side and back views of a hypersonic vehicle.
Erich Elsen, Patrick LeGresley, Eric Darve, 2008

Enters CUDA

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture.

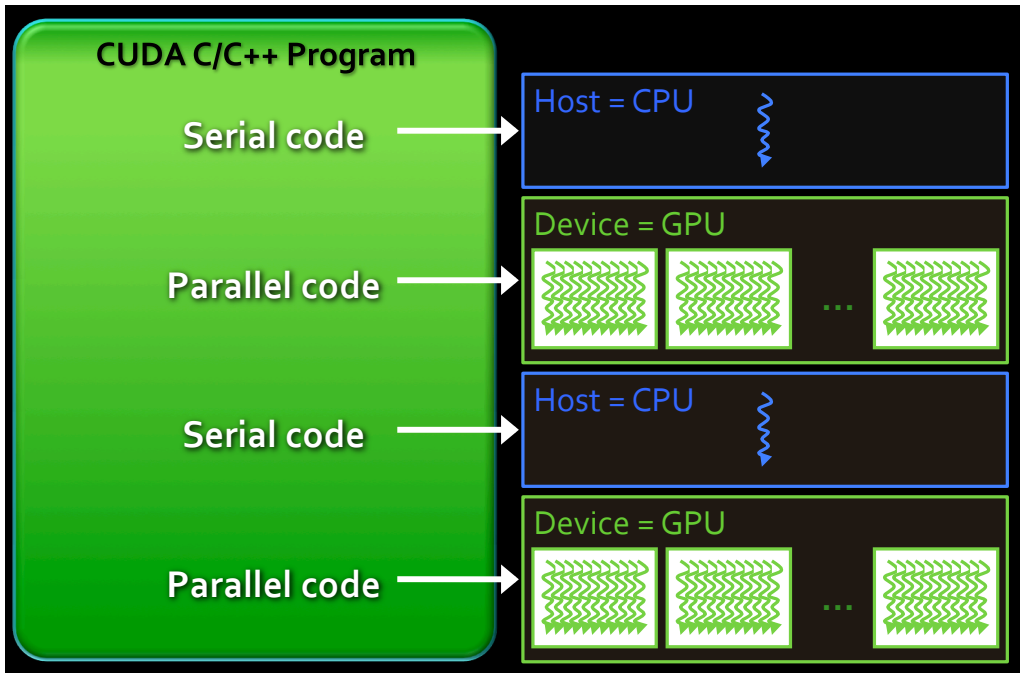
Supports C, C++, FORTRAN, application programming interfaces, directives.



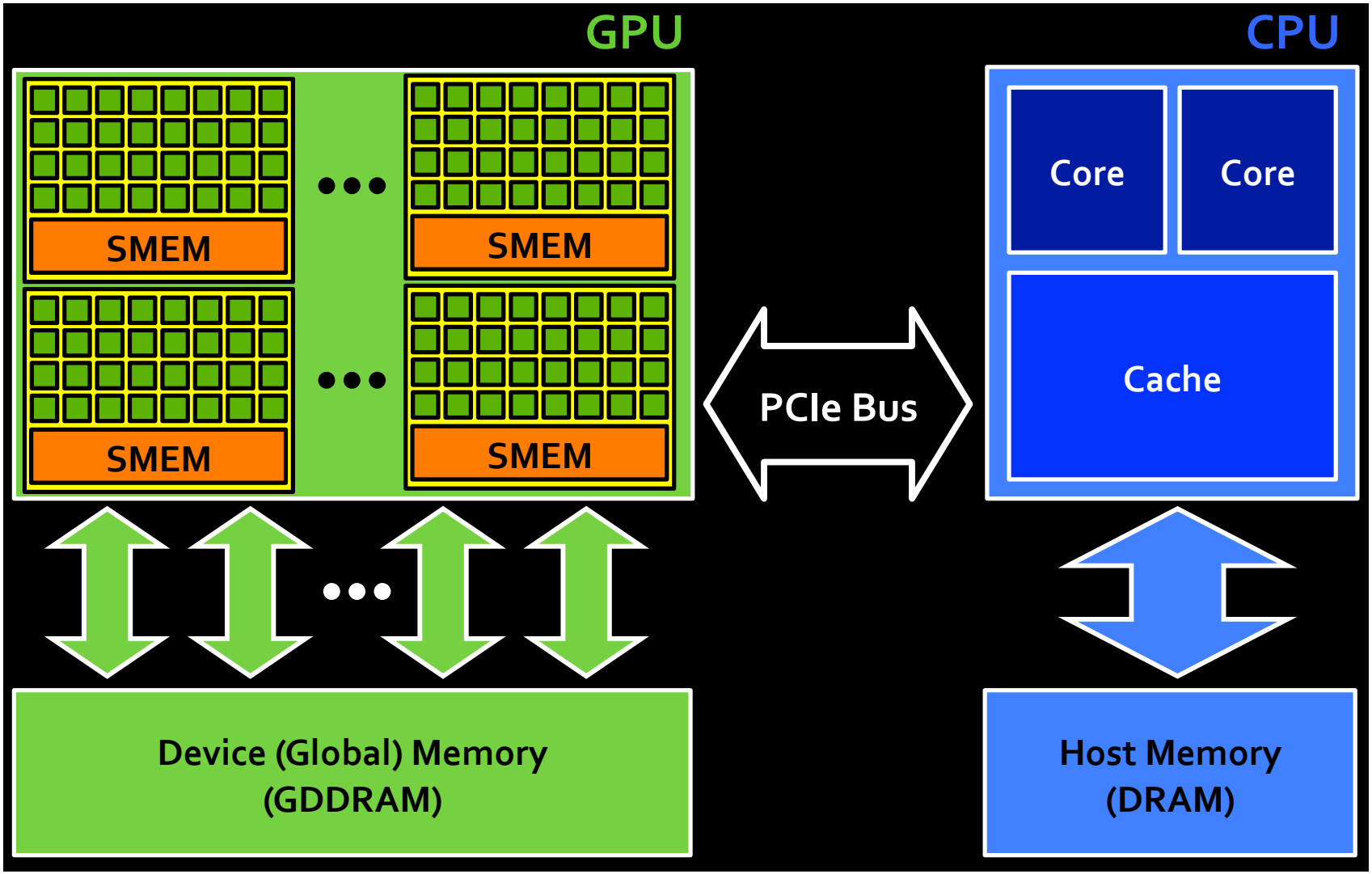
Anatomy of program

Before getting into CUDA, let's look at the overall organization of the code.

- Serial code executes in a host (CPU) thread
- Parallel code executes in many concurrent device (GPU) threads across multiple parallel processing elements



Typical execution flow of CUDA program



A simple example

```
1  // Host code
2  int main()
3  {
4      size_t size = ...;
5      float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
6      // Allocate input vectors h_A and h_B in host memory
7      h_A = (float*)malloc(size);
8      h_B = (float*)malloc(size);
9      h_C = (float*)malloc(size);
10
11     // Initialize input vectors
12     ...
13
14     // Allocate vectors in device memory
15     cudaMalloc(&d_A, size);
16     cudaMalloc(&d_B, size);
17     cudaMalloc(&d_C, size);
18     ...
```

```
1     ...
2     // Copy vectors from host memory to device memory
3     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
4     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
5
6     // Invoke kernel
7     int nthreads = 256;
8     int nblocks = (N + nthreads - 1) / nthreads;
9     VecAdd<<<nblocks, nthreads>>>(d_A, d_B, d_C, N);
10
11    // Copy result from device memory to host memory
12    // h_C contains the result in host memory
13    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
14    ...
15 }
```

```
1  // Device code
2  __global__ void VecAdd(const float* A,
3                         const float* B,
4                         float* C,
5                         int N)
6  {
7      int i = blockDim.x * blockIdx.x + threadIdx.x;
8      if (i < N)
9          C[i] = A[i] + B[i];
10 }
```