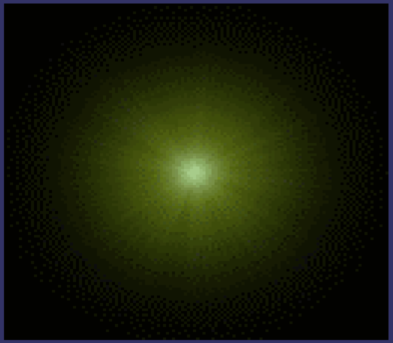
GPUs and GPGPUs

Greg Blanton
John T. Lubia



PROCESSOR ARCHITECTURAL ROADMAP

- Design CPU
  - Optimized for sequential performance

- ILP increasingly difficult to extract from instruction stream
  - Control hardware to check for dependencies, out-of-order execution, prediction logic etc

- Control hardware dominates CPU
  - Complex, difficult to build and verify
  - Takes substantial fraction of die
  - Increased dependency checking the deeper the pipeline
  - Does not do actual computation but consumes power

Move from Instructions per second to Instructions per watt

- Power budget and sealing becomes important
- More transistors on the chip
  - Difficult to scale voltage requirements

PARALLEL ARCHITECTURE

Hardware architectural change from sequential approach to inherently parallel

- Microprocessor manufacture moved into multi-core
- Specialized Processor to handle graphics
  GPU - Graphics Processing Unit

CPU/GPU Parallelism

- Moore's Law gives you more and more transistors
  - What do you want to do with them?
  - CPU strategy: make the workload (one compute thread) run as fast as possible
    - Tactics:
      - Cache (area limiting)
      - Instruction/Data prefetch
      - Speculative execution

      →limited by "perimeter" – communication bandwidth

      …then add task parallelism…multi-core
  - GPU strategy: make the workload (as many threads as possible) run as fast as possible
    - Tactics:
      - Parallelism (1000s of threads)
      - Pipelining

      → limited by "area" – compute capability

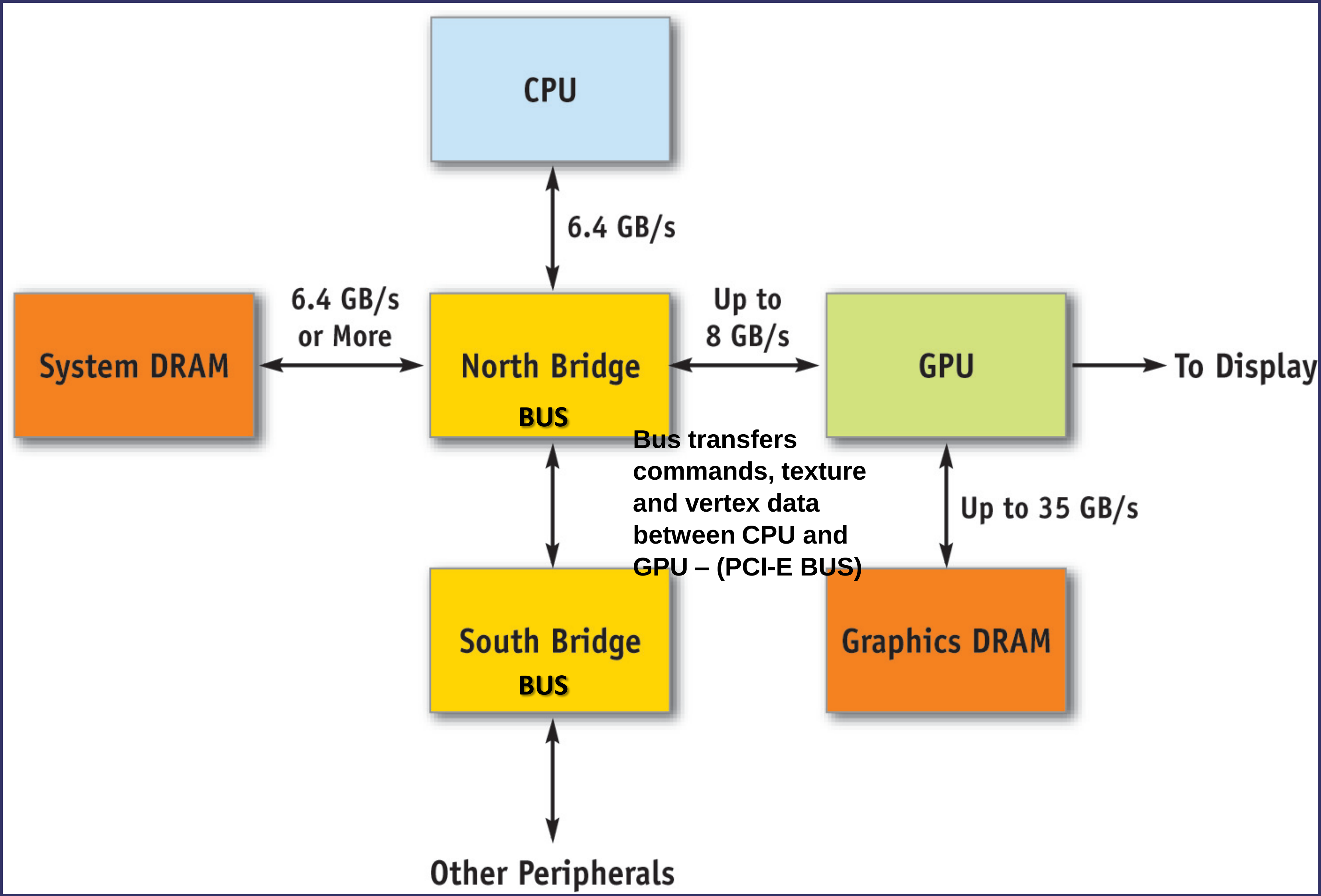
Extracted from NVIDIA research paper

GPU

- Graphics Processing Unit
  - High-performance many-core processors
- Data Parallelized Machine - SIMD Architecture
- Pipelined architecture
  - Less control hardware
  - High computation performance
  - Watt-Power invested into data-path computation

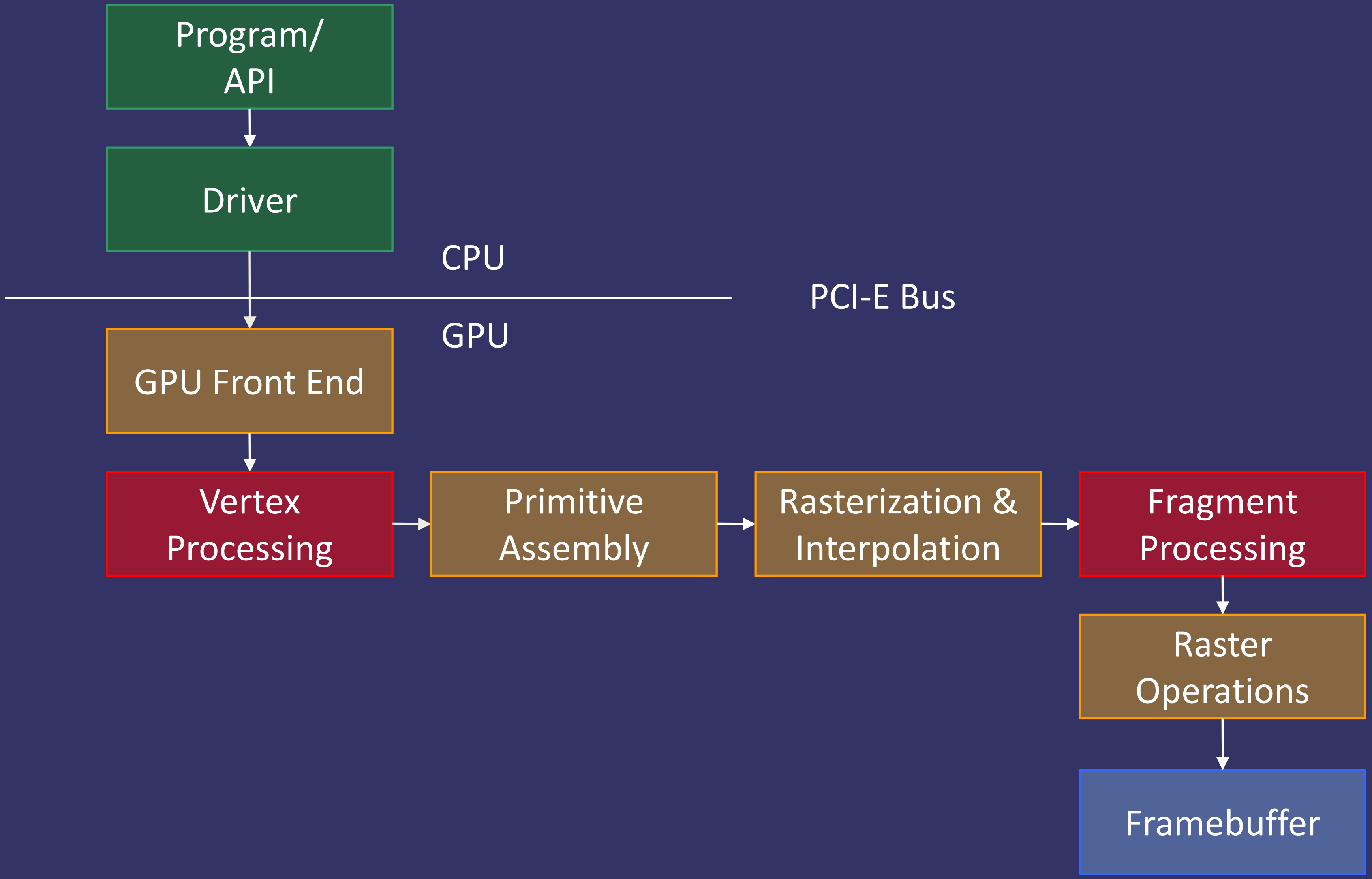
PC ARCHITECTURE

How the GPU fits into the Computer System

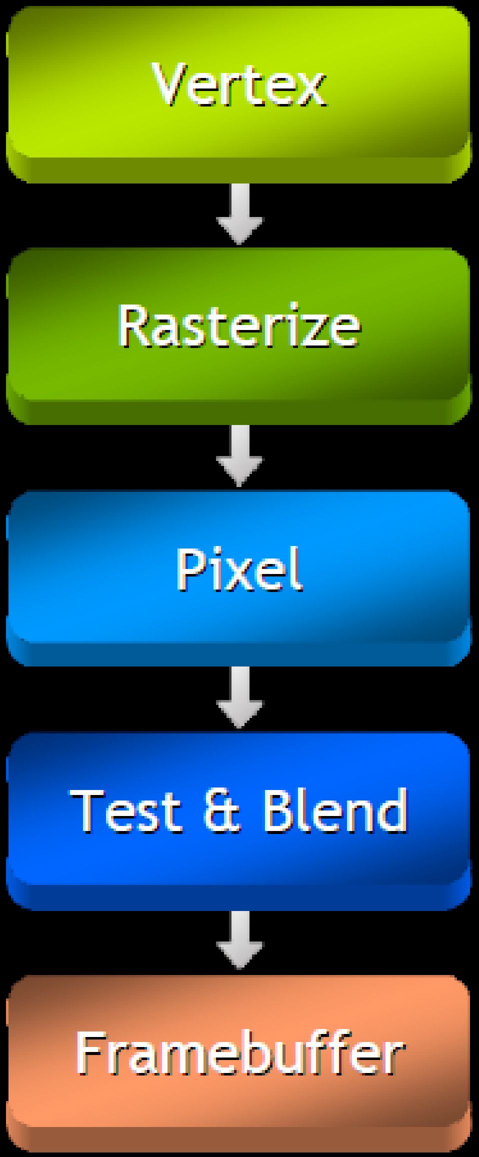


Bus transfers commands, texture and vertex data between CPU and GPU – (PCI-E BUS)

GPU pipeline



The Graphics Pipeline

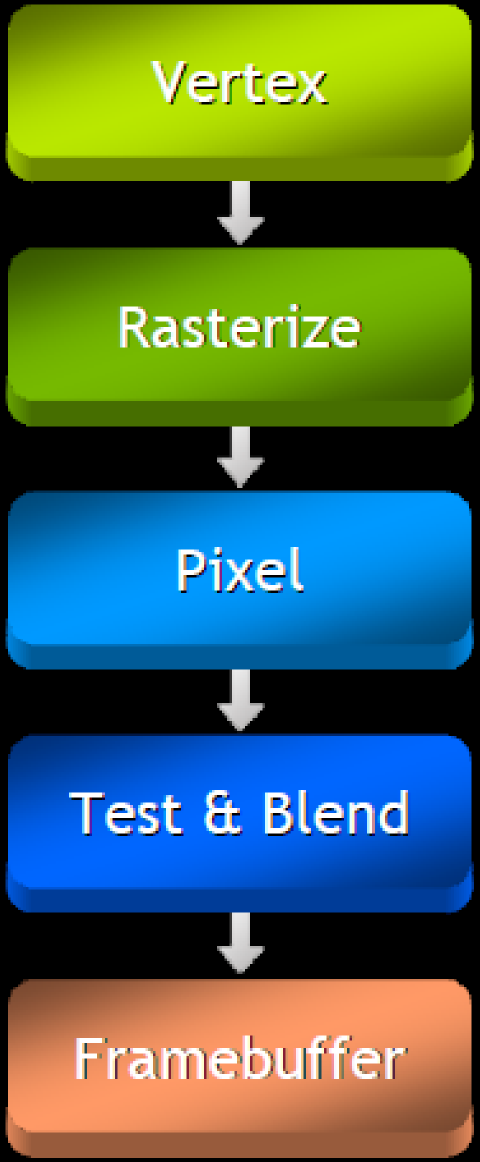


- Key abstraction of real-time graphics
- Hardware used to look like this
- Distinct chips/boards per stage
- Fixed data flow through pipeline

Ideal for graphics processing

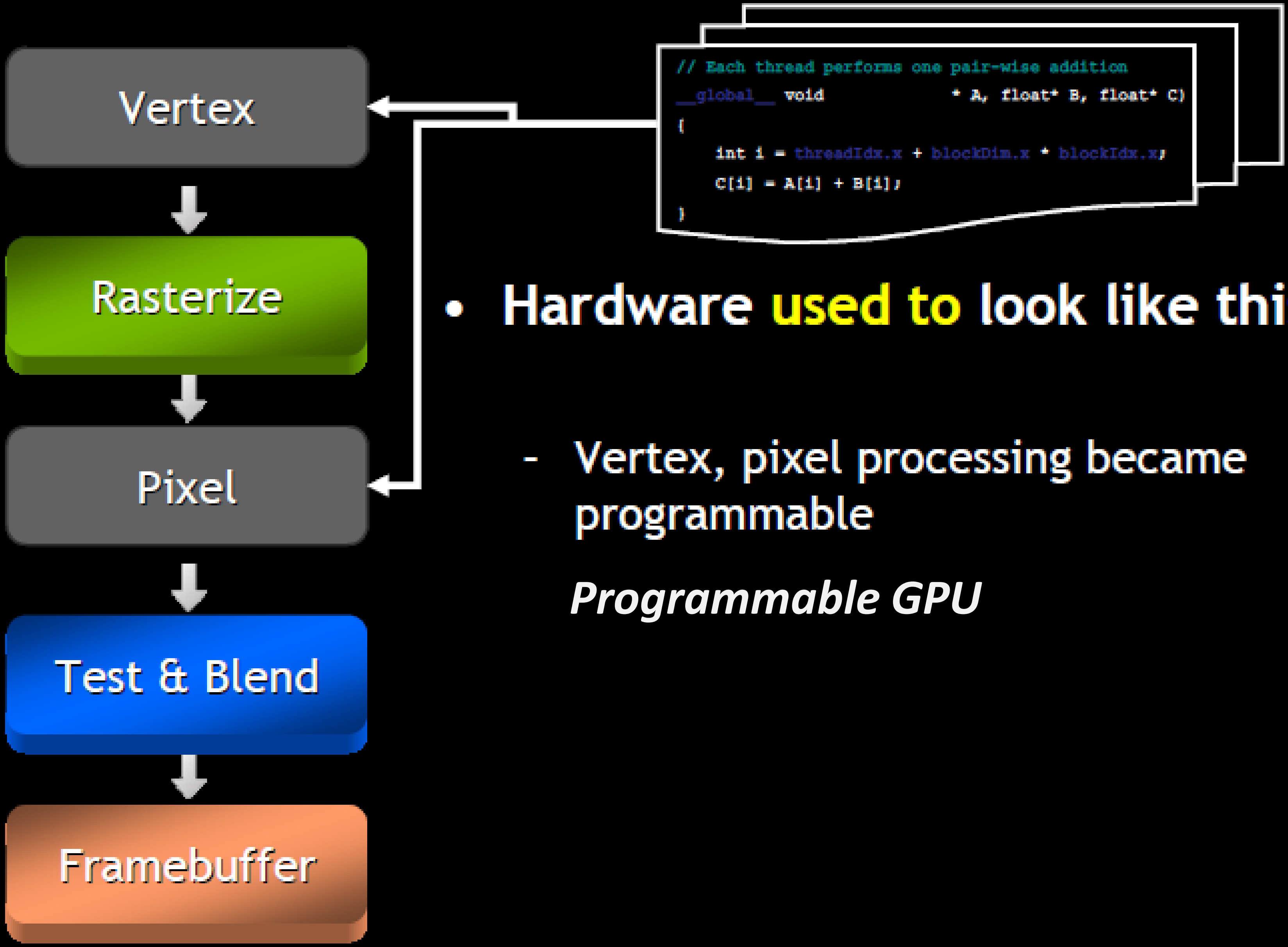
Extracted from NVIDIA research paper

The Graphics Pipeline



- Remains a useful abstraction
- Hardware used to look like this

The Graphics Pipeline



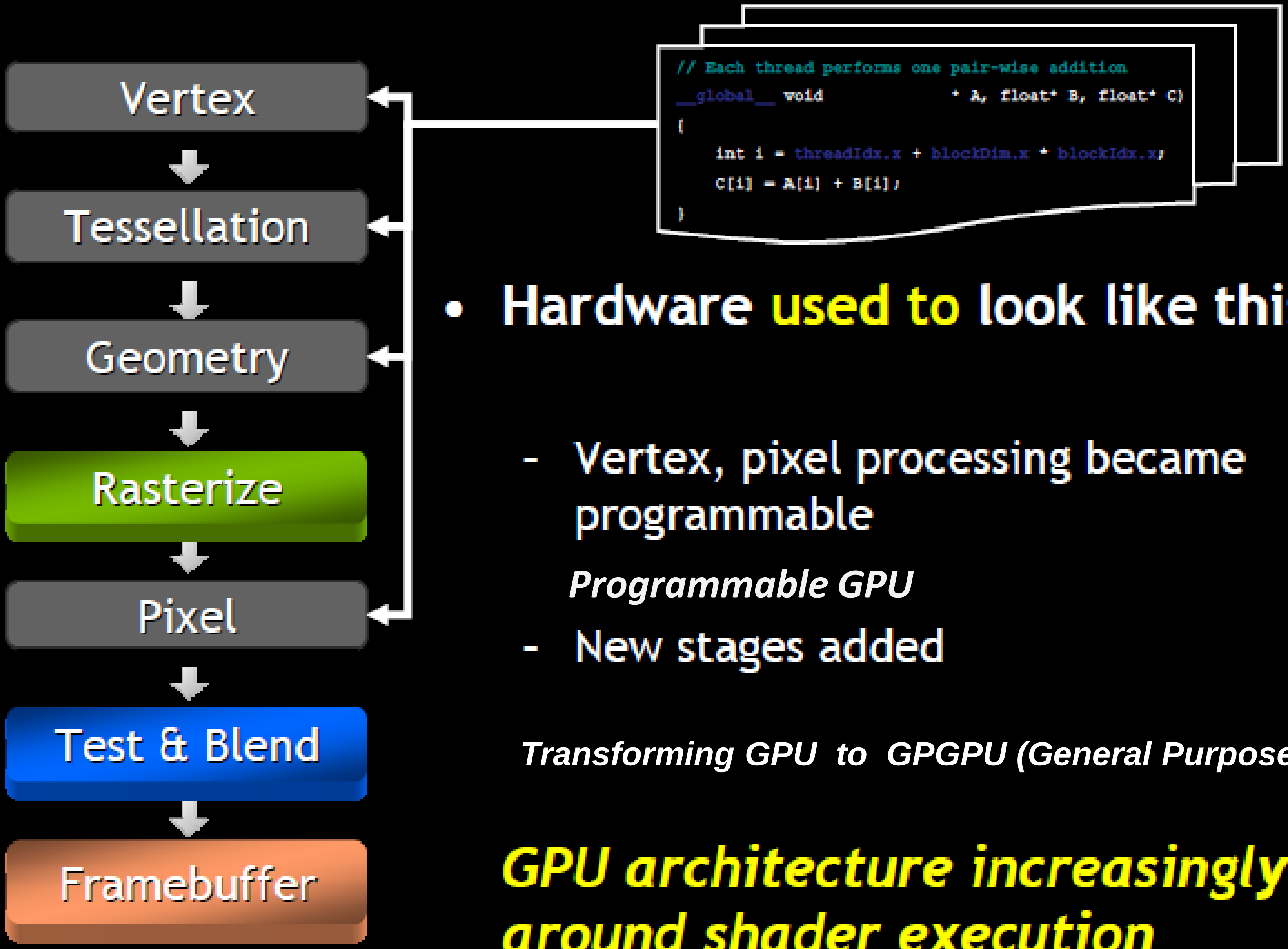
```
// Each thread performs one pair-wise addition
__global__ void                * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Hardware used to look like this:

  - Vertex, pixel processing became programmable

  Programmable GPU

The Graphics Pipeline



```
// Each thread performs one pair-wise addition
__global__ void                  * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Hardware used to look like this
  - Vertex, pixel processing became programmable

    Programmable GPU
  - New stages added

Transforming GPU to GPGPU (General Purpose GPU)

GPU architecture increasingly centers around shader execution

Shaders are execution Kernel of each pipeline stage

Extracted from NVIDIA research paper

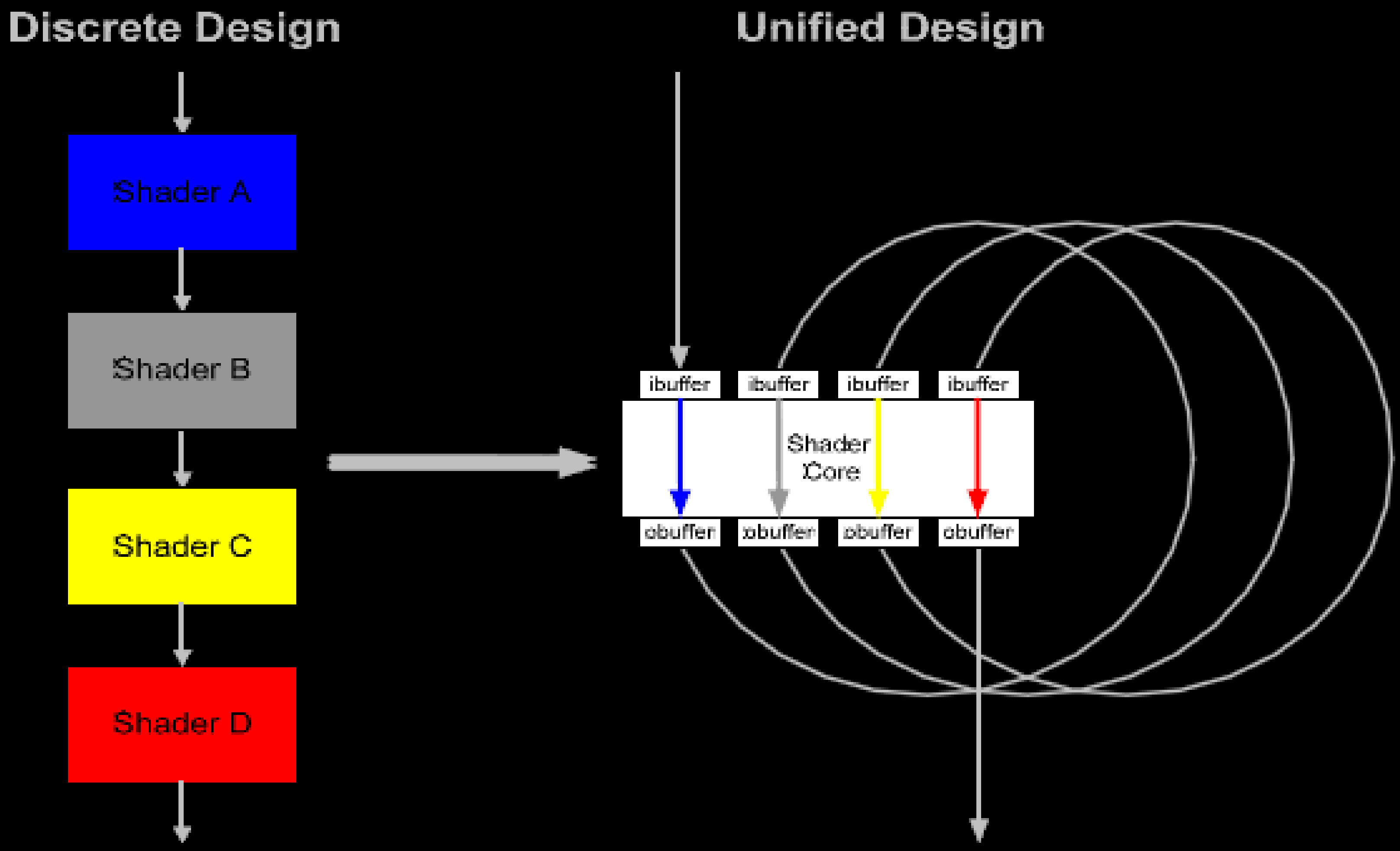
Goal: Performance per millimeter

SIGGRAPH2008

- For GPUs, perfomance == throughput
- Strategy: hide latency with computation not cache
  ➔ Heavy multithreading!
- Implication: need many threads to hide latency
  - Occupancy – typically prefer 128 or more threads/TPA
  - Multiple thread blocks/TPA help minimize effect of barriers
- Strategy: Single Instruction Multiple Thread (*SIMT*)
  - Support SPMD programming model
  - Balance performance with ease of programming

Extracted from NVIDIA research paper

Modern GPUs: Unified Design



Vertex shaders, pixel shaders, etc. become *threads* running different programs on a flexible core

Extracted from NVIDIA research paper

SIMT Thread Execution

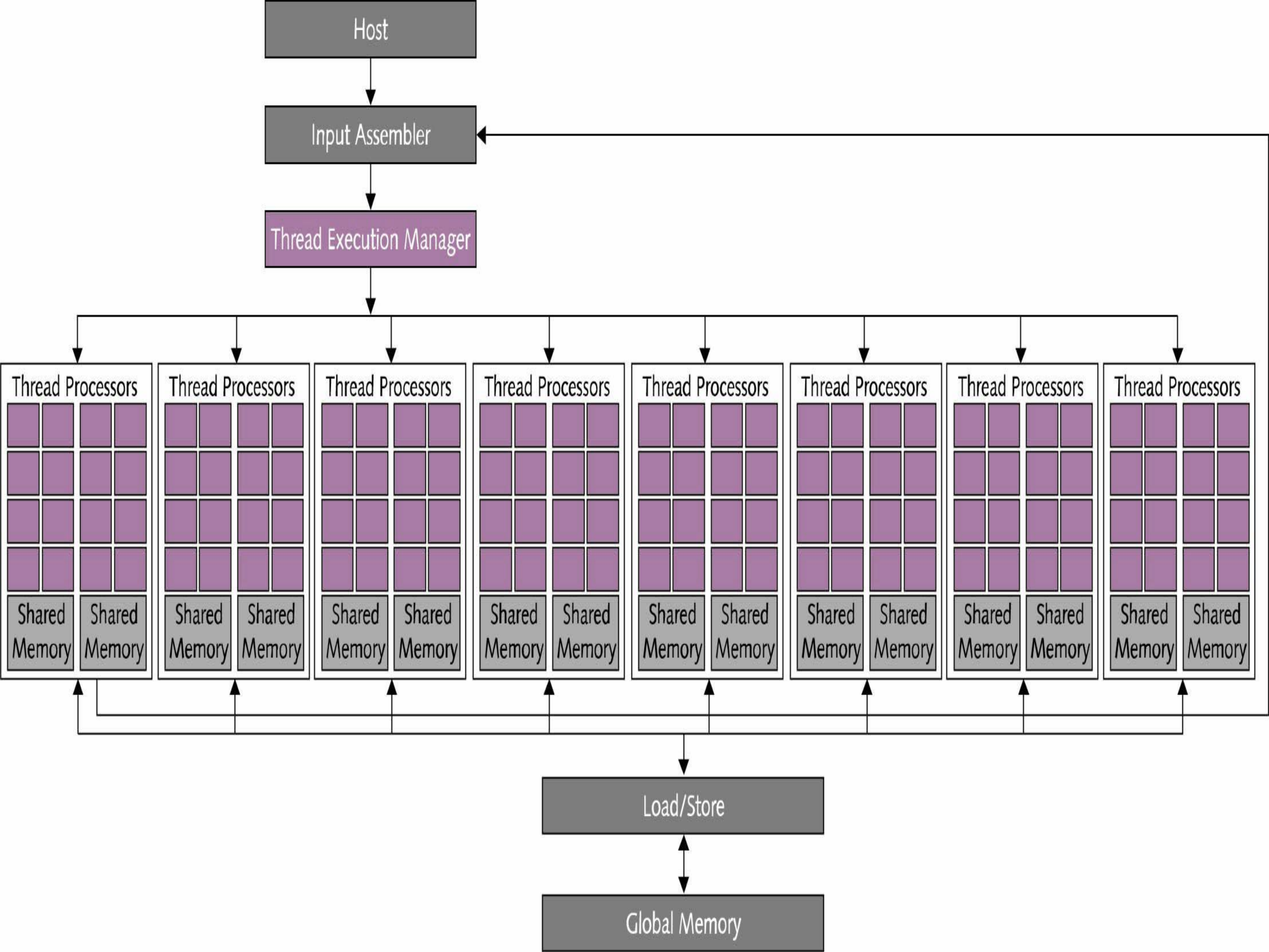
- High-level description of SIMT:
  - Launch zillions of threads
  - When they do the same thing, hardware makes them go fast
  - When they do different things, hardware handles it gracefully

GeForce 8 GPU has 128 thread processors.

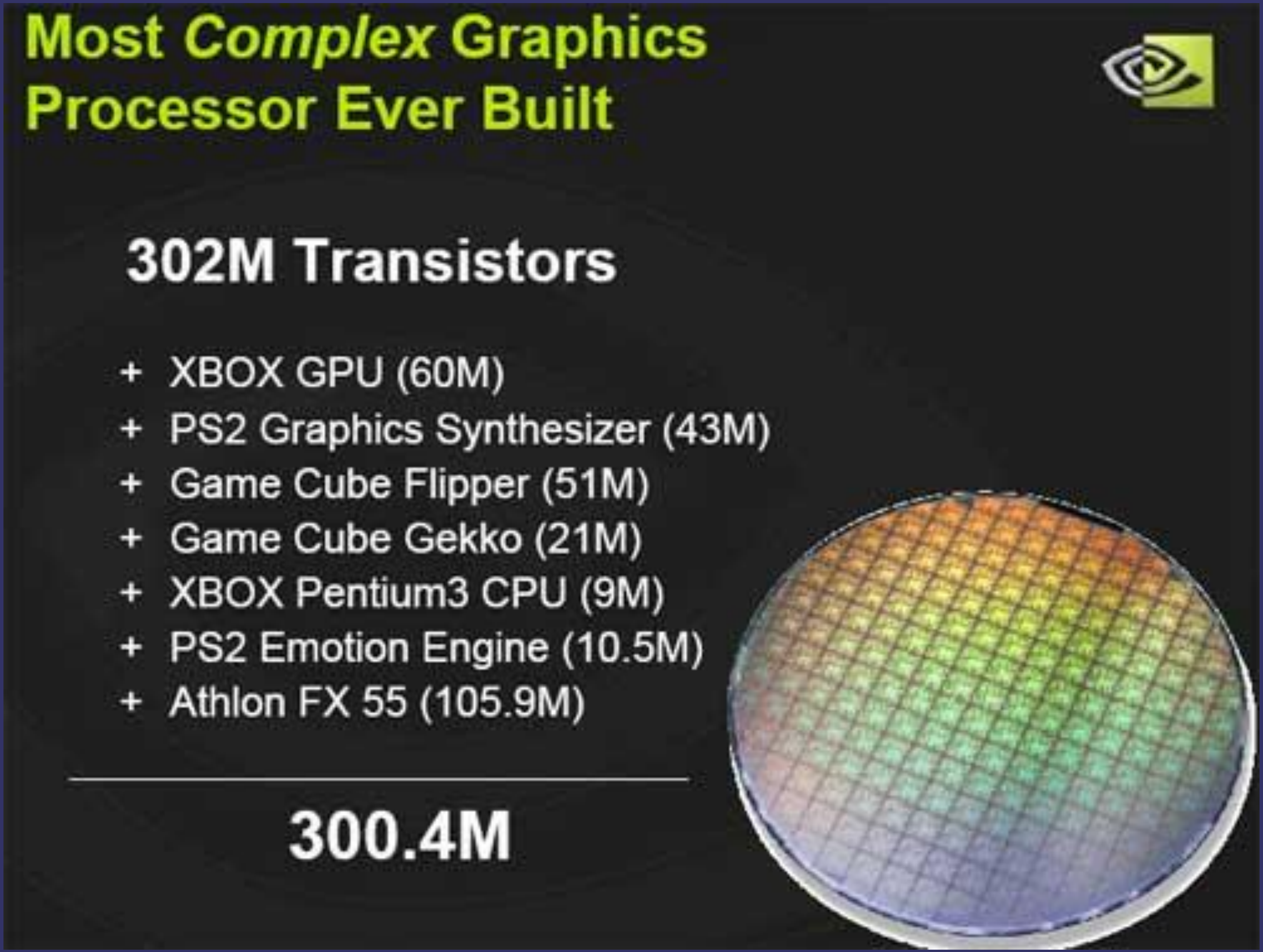
Each thread processor has a single-precision FPU and 1,024 registers, 32
bits wide.

Each cluster of eight thread processors has 16KB of shared local memory supporting parallel data accesses.

A hardware thread-execution manager automatically issues threads to the processors without requiring programmers to write explicitly threaded code.


Host
Input Assembler
Thread Execution Manager
Thread Processors
Shared Memory
Load/Store
Global Memory

GeForce 7800 GTX



Most *Complex* Graphics Processor Ever Built

302M Transistors

- + XBOX GPU (60M)
- + PS2 Graphics Synthesizer (43M)
- + Game Cube Flipper (51M)
- + Game Cube Gekko (21M)
- + XBOX Pentium3 CPU (9M)
- + PS2 Emotion Engine (10.5M)
- + Athlon FX 55 (105.9M)

300.4M

FLOPS: CPU vs GPU

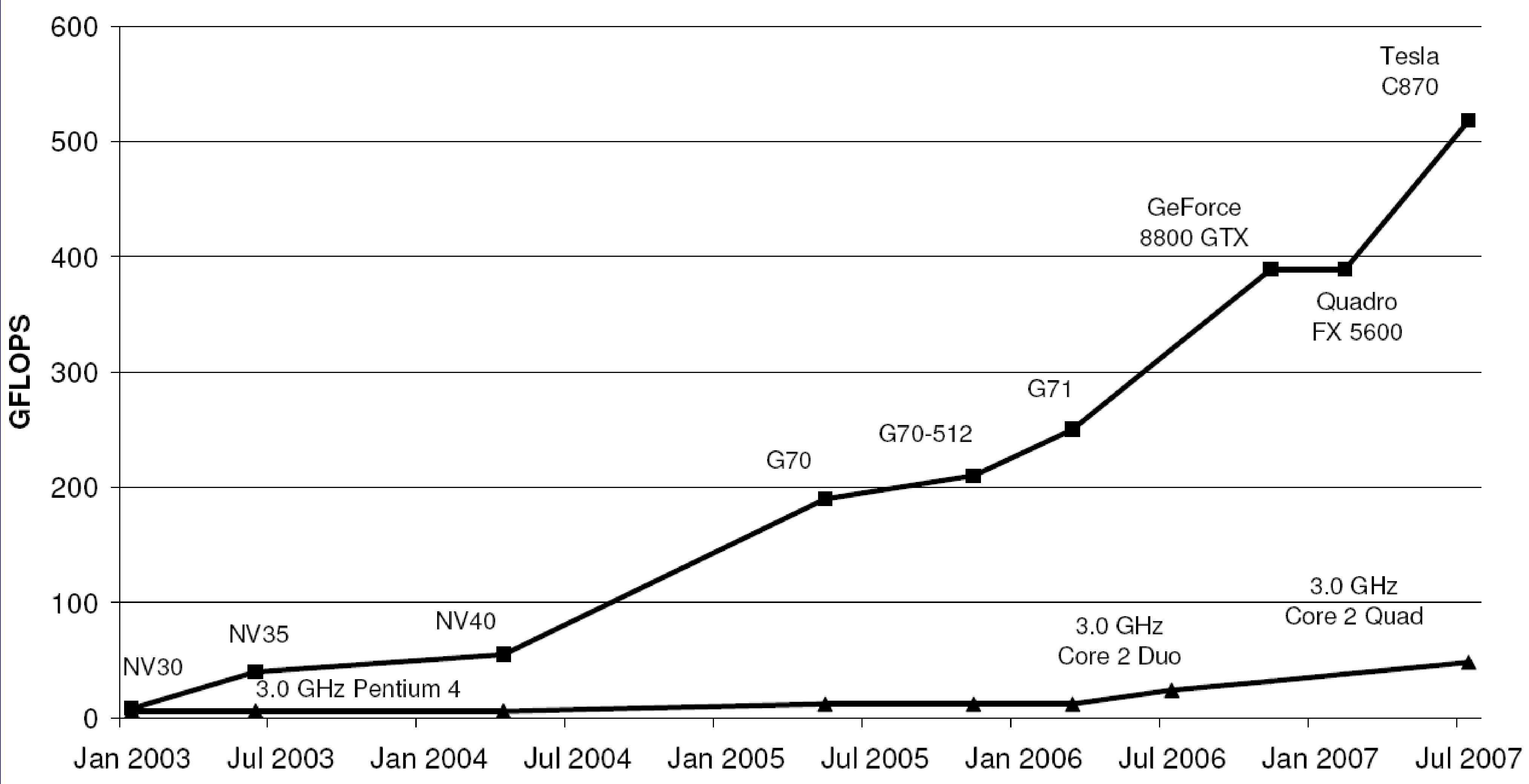


Figure 1.1, Chapter 1, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter1-Introduction.pdf

LATEST NVIDIA DESKTOP GPU

GeForce GTX 295
Release Date: 01/08/2009
Series: GeForce GTX 200
Core Clock: 576 MHz
Shader Clock: 1242 MHz
Memory Clock: 999MHz (1998 DDR)
Memory Bandwidth: 223.776 GB/sec
FLOPS: 1.79 TFLOPS (1788.48 GFLOPS)

Processor Cores: 480 (240 per GPU)

NVIDIA CUDA (Compute Unified Device Architecture)

- Software platform for massively parallel high-performance computing on NVIDIA's powerful GPUs
- Repositioning its GPUs as versatile devices suitable for much more than electronic games and 3D graphics
- Insurance against an uncertain future for discrete GPUs

Mix Code

- NVIDIA now prefers shaders to be called “stream processors” or “thread processor”
- Requires special code for parallel programming, but not to explicitly manage threads in a conventional sense
- GPU code can mix with general-purpose code for the host CPU
- Aims at data-intensive applications that need single-precision floating-point math

Overview of GeForce 8 architecture

- 128 thread processors
- Each capable of managing up to 96 concurrent threads (for a maximum of 12,288 threads)
- Each thread has its own stack, register file, program counter, and local memory.
- Each thread processor has 1024 physical registers, 32 bits wide implemented in SRAM instead of latches.
- NVIDIA can completely redesign this architecture in the next release of GPUs without making the API obsolete or breaking anyone's application software.

CUDA Automatically Manages Threads

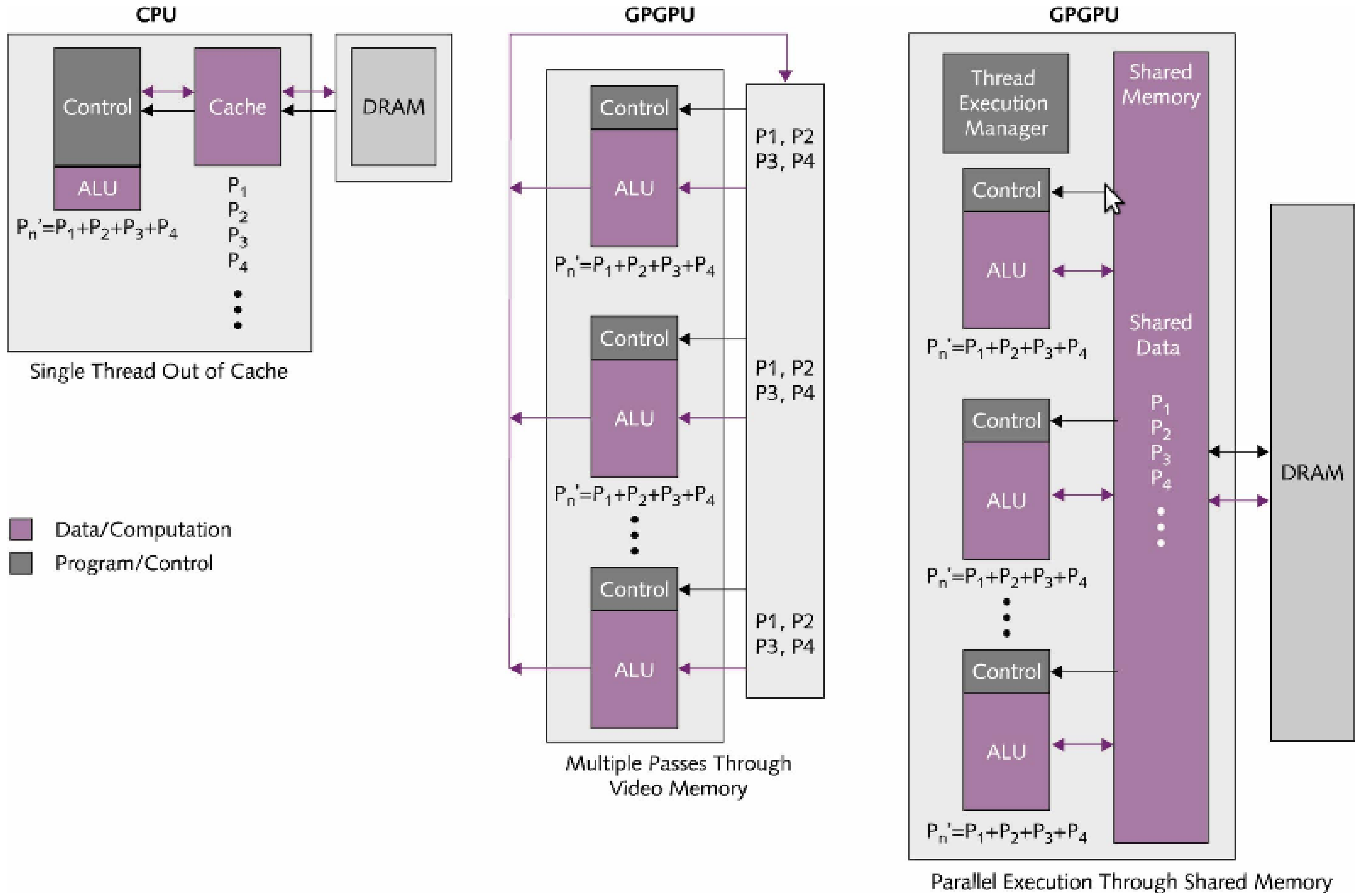
Divides the data set into smaller chunks stored in on-chip memory

Storing locally reduces the need to access off-chip memory, thereby improving performance - latency

Application programmers don't write explicitly threaded code

A Hardware threading manager handles threading automatically

Three different models for High Performance Computing (HPC)



CUDA makes deadlocks among threads impossible (in theory)

- CUDA eliminates deadlocks, no matter how many threads
- Special API call, syncthreads, provides explicit barrier synchronization
- Invokes a compiler-intrinsic function that translates into a single instruction for the GPU
- Barrier instruction blocks threads from operating on data that another thread is using

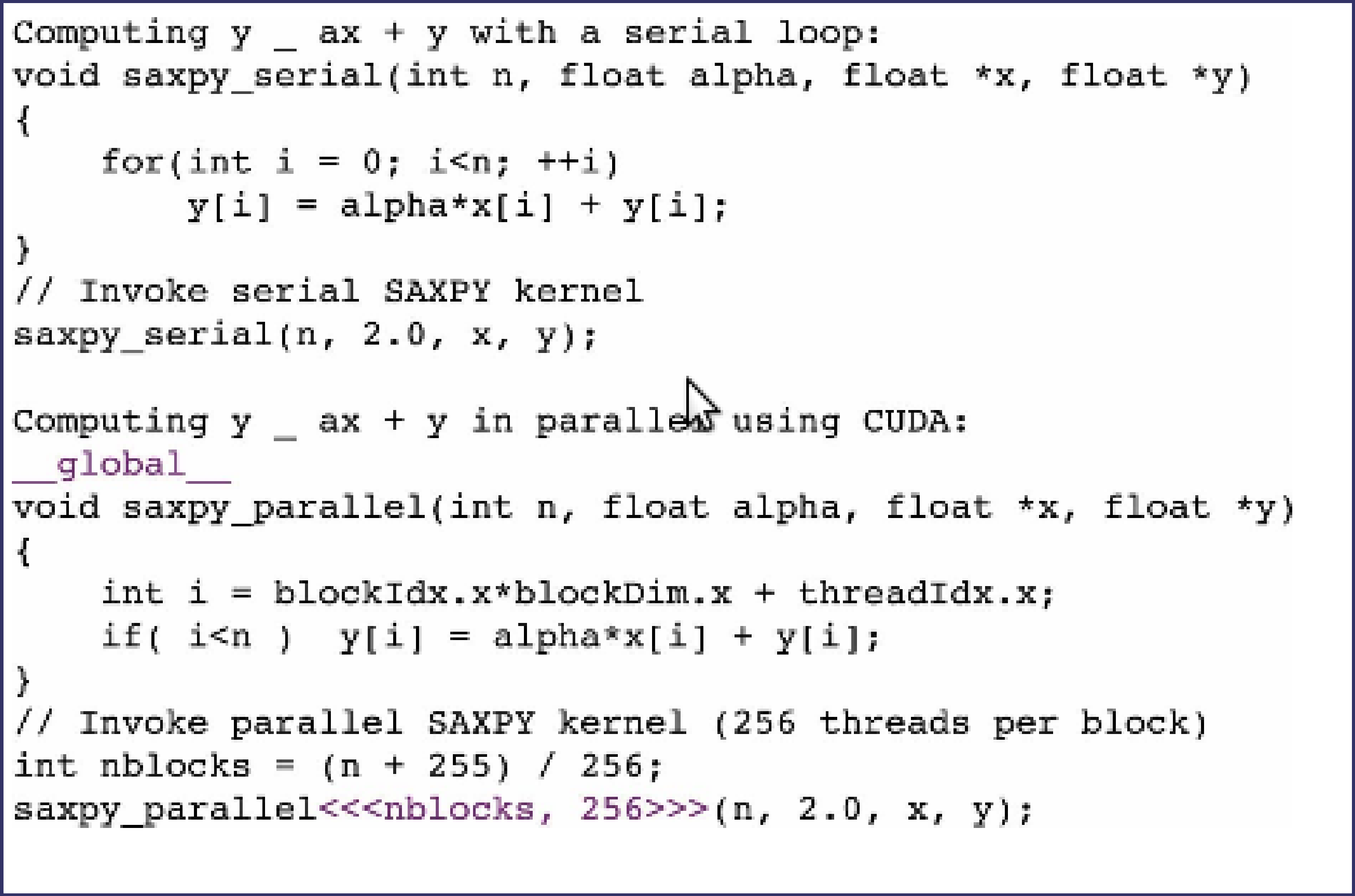
Developers Must Analyze Data

- Problems must be analyzed to determine how best to divide the data into smaller chunks for distribution among the thread processors

- Possible real-world example using CUDA:
  - Scanning network packets for malware
  - One array compared to another (data vs sigs)
  - Dedicate a lightweight thread to each virus sig

- Developers face challenges in analyzing their algorithms and data to find the optimal number of threads and blocks to keep the GPU fully utilized

CUDA programming example



```
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);
```

CONs of CUDA

CONs

- Single-precision floating point is sufficient for consumer graphics, so GPUs don't yet support double precision (planned for future GPU releases)
- Isn't the only option for software development on NVIDIA's GPUs
- Could see resistance from developers
- Tied to one vendor, NVIDIA

Conclusion

Massive amount of power available to users through the use of GPUs and GPGPUs

NVIDIA's approach CUDA

CUDA is limited to just NVIDA's platform

Real-time CUDA Examples

Some Real-time CUDA Demos

- deviceQuery
- histogram64
- nbody
- nbody (emulated)
- fluidsGL
- particles
- oceanFFT
- smoke

References

http://www.nvidia.com

Extracted some slides from work done by David Luebke, NVIDIA Research presentation
http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

http://www.gpgpu.org

Jeff A. Stuart and John D. Owens, Message Passing on Data-Parallel Architectures, Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium

http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf
http://www.youtube.com/watch?v=nlGnKPpOpbE
http://www.nvidia.com/object/product_geforce_gtx_295_us.html
http://www.gpureview.com/GeForce-GTX-295-card-603.html
http://courses.ece.illinois.edu/ece498/al/textbook/Chapter1-Introduction.pdf

http://www.nytimes.com/2003/05/26/business/technology-from-playstation-to-supercomputer-for-50000.html

Special Thanks:
Xuan Wu for the idea of running CUDA demos during the presentation