GPGPU

General-Purpose computation on
Graphics Processing Units

OUTLINE

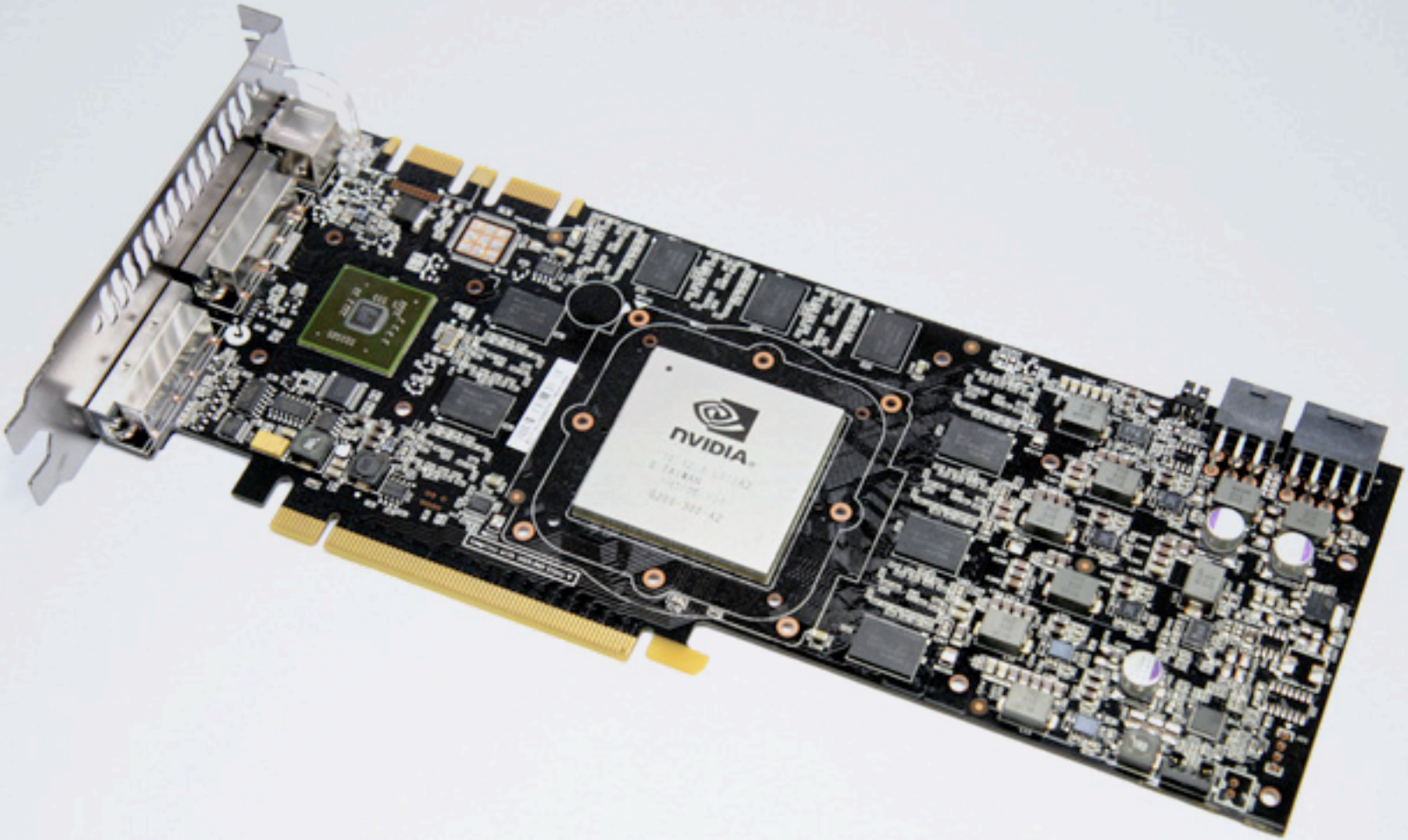
- The development of GPUs: *From GPU to GPGPU*
- GPU hardware: *NVIDIA 200-Series case study*
- GPU software: *A look into the driver*
- GPGPU software: *Programming GPUs today*
  NVIDIA CUDA case study
- Ongoing and expected developments: *The future of GPGPU (?)*

GPUS FOR GRAPHICS

The leading cause for innovation in GPU architecture

GROWTH

- GPU market grows despite recession (*Tom's Hardware*)
  - About 6% rate, backing 50G$ video game market (*ars technica*)
- Super-exponential transistor count growth
  - NVIDIA GeForce GTX 280 - 1400 million
  - Terra-FLOP capacity
- Not just for Games: Multimedia, Physics, HPC, ... (?)



A GRAPHICS CARD

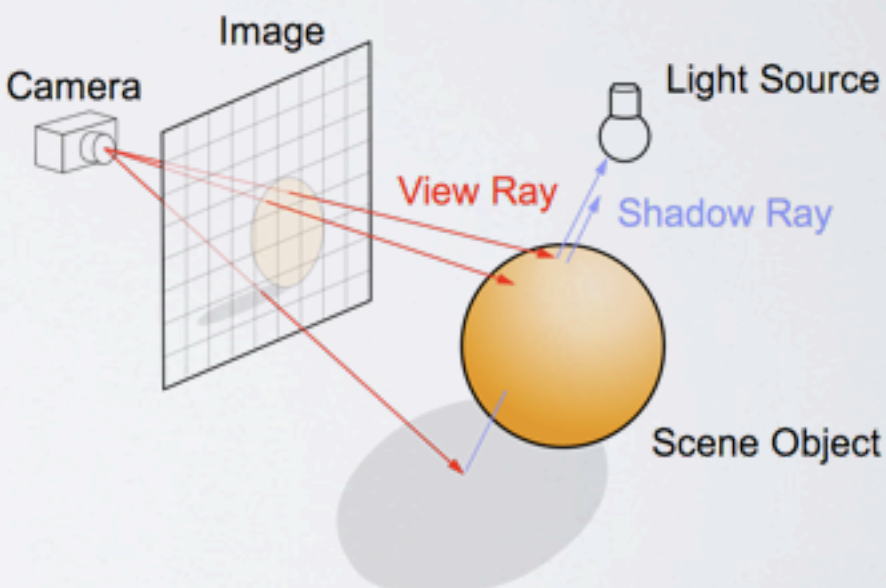
NVIDIA GTX280, 1GB GDDR3

A COMPLEX GFX SCENE

in-game screenshot from
GRID (Codemasters)

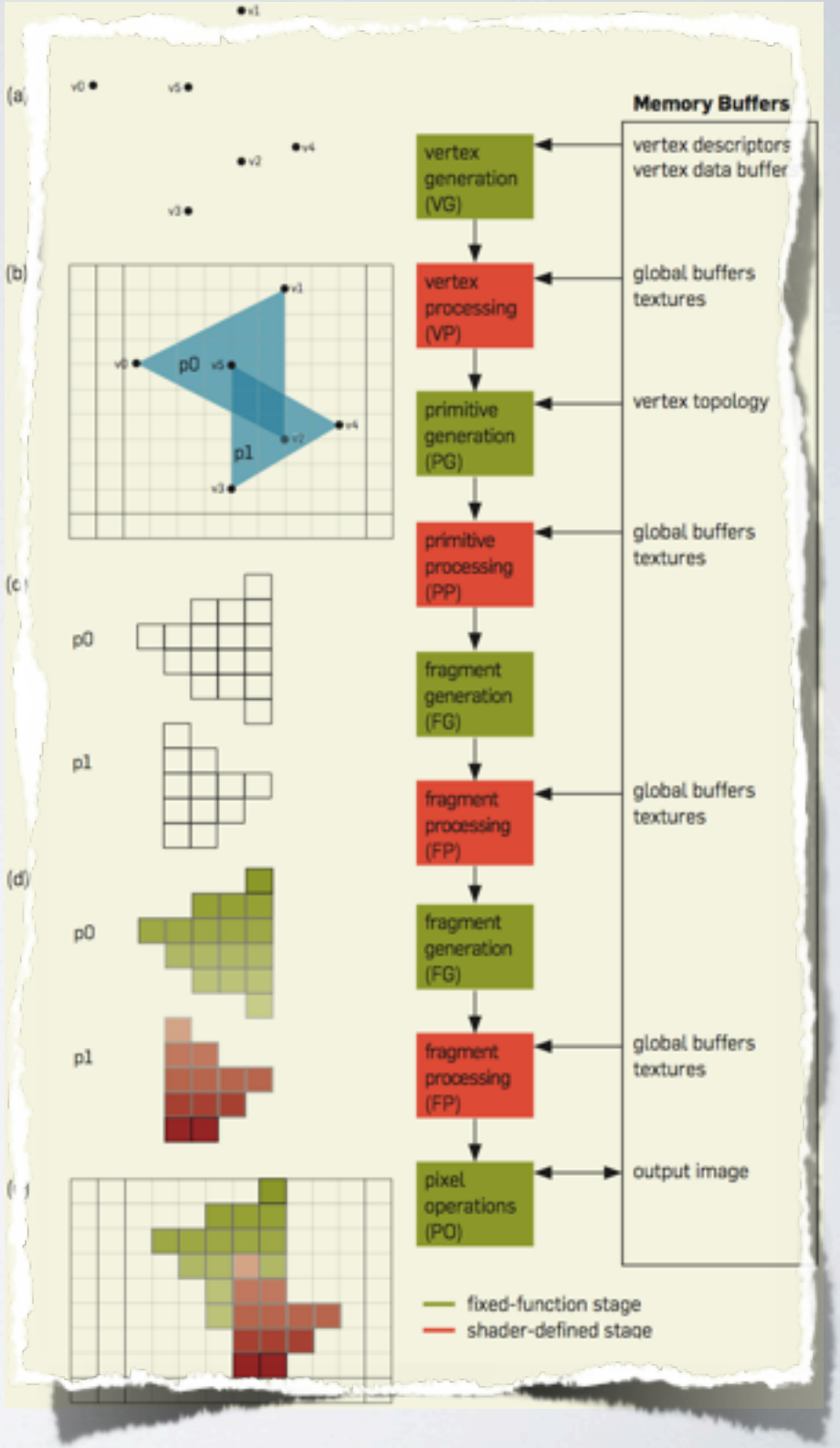
3D RENDERING

- Modeling ➠ Animation ➠ Rendering
- Ray-tracing: Algorithm for photorealistic 2D representation of 3D scenes
- Rasterization: Algorithm for efficient real-time 3D-rendering



GRAPHICS PIPELINE

Mapping 3D-world to screen through a GFX-API



SHADER PROGRAMMING

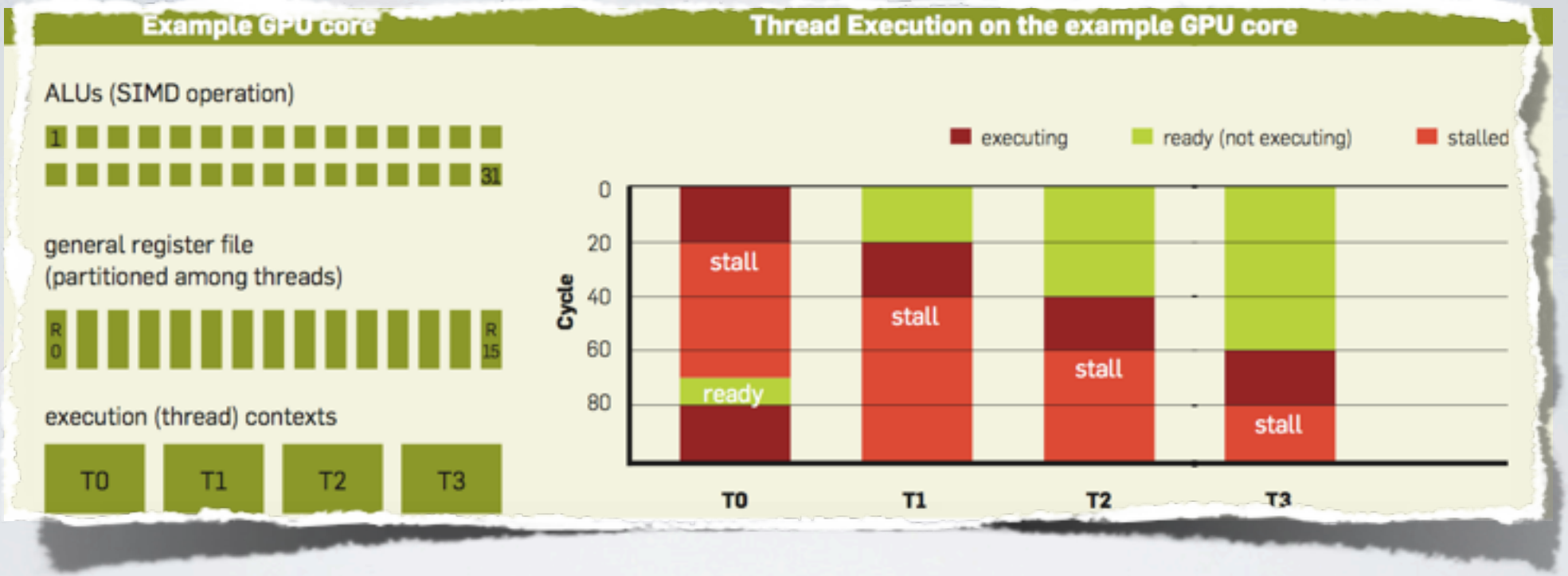
- Graphics-API-specific language implementation (Cg/GLSL/HLSL)
- Full control flow support
- Operate on read-only textures (or output of previous stage)
- Shader model 4.0 : Vertex / Geometry / Pixel
  - Directx11 introduces two more shaders (programmable stages)
- Ship intermediate language "binaries"

GFX API CHALLENGES

- Task (function) / Data parallelism ➡ GFX Pipeline / many-core architecture
- Scene-dependent workload ➡ Unify shaders
- Bottlenecks ➡ fixed function units
- Common operations ➡ SIMD-like execution
- Massive memory accesses with/out patterns ➡ Memory access/ hierarchy innovation

GPU CHARACTERISTICS

- SIMD: Single Instruction, Multiple Data
  - Example: RGBA, XYZW quadruples
- Many-core + Wide SIMD = Lots of ALUs
  - SIMD control flow = predicated execution
- Hardware multithreading = high processor utilization
  - Needs compile-time, static information to be realized



SAMPLE SHADER EXECUTION

Execute a 32-wide SIMD instruction / thread
switch to another on every 1 to 4 cycles

GPU MEMORY SYSTEM

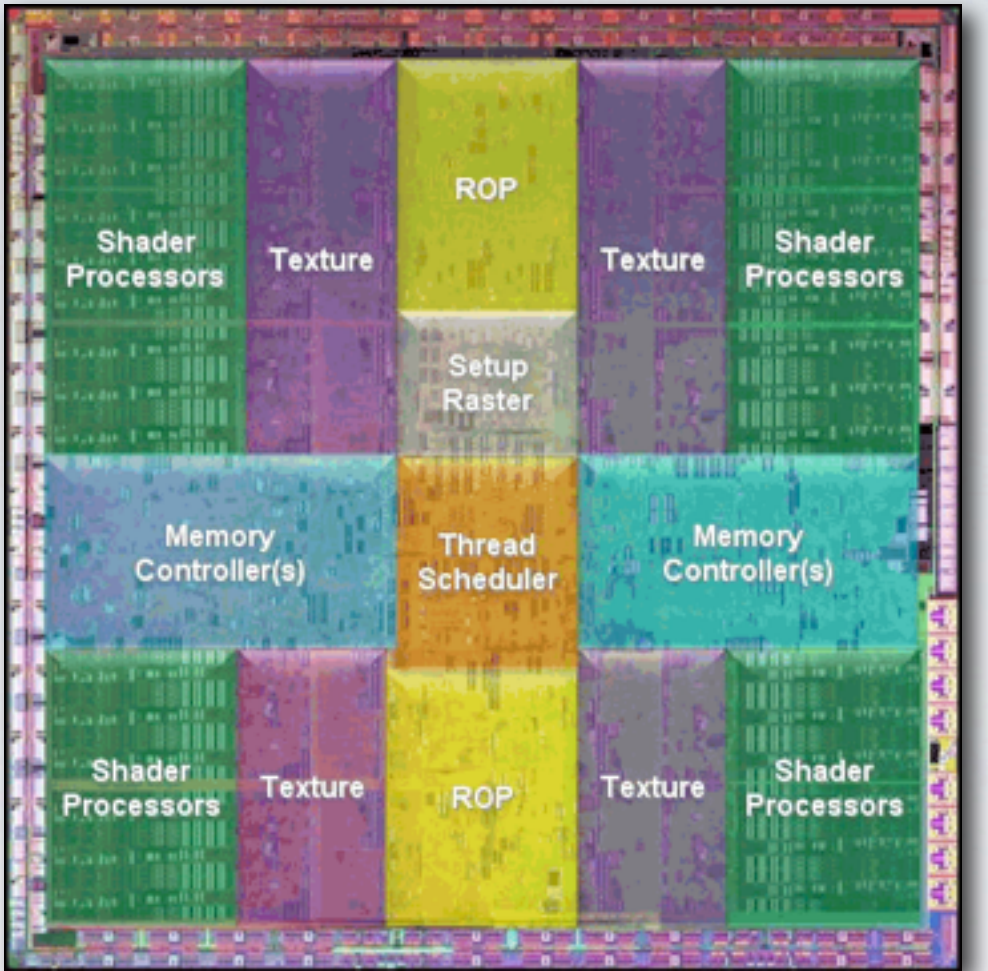
- Focus on high bandwidth rather than low latency
  - need to transfer massive textures between host and device
- Flat but exposed memory hierarchy
  - Limited or programmer-manageable caching
  - Big SIMD-wide register files
- Memory access coalescing necessary for performance

GPU EXECUTION SYSTEM

- Schedule and assign threads to maintain pipeline flow
- Resize and manage buffers and reorder memory access to avoid costly collisions
- Enable control flow with least possible overhead
- Load balancing under pipeline limitations/ enhancements:
  - ex. occluded fragments shading
  - ex. bottlenecks, reseeding possibilities

GPUS FOR GPGPU

How GPU architecture transforms to enable General Purpose programming



CASE STUDY: NVIDIA GTX280

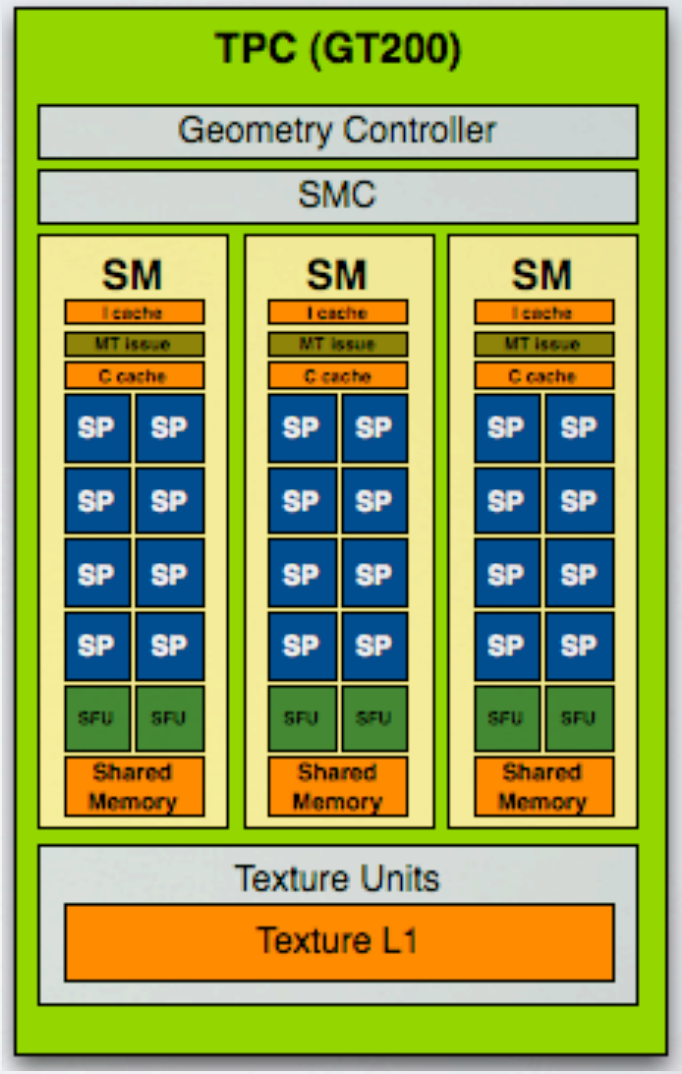
The NVIDIA 200-Series representative,
built with GPGPU in mind

GT200 CHARACTERISTICS

- Monolithic die
- 240-Stream-Processor Array
- 80/80 Texture Address / Filtering
- 32 ROPs
- 602MHz Core Clock
- 1296MHz Shader Clock
- 1107MHz Memory Clock
- 512-bit Memory Bus Width
- 1GB GDDR3 Frame Buffer
- 1.4B Transistor Count
- TSMC 65nm Manufacturing Process
- 350$ Price

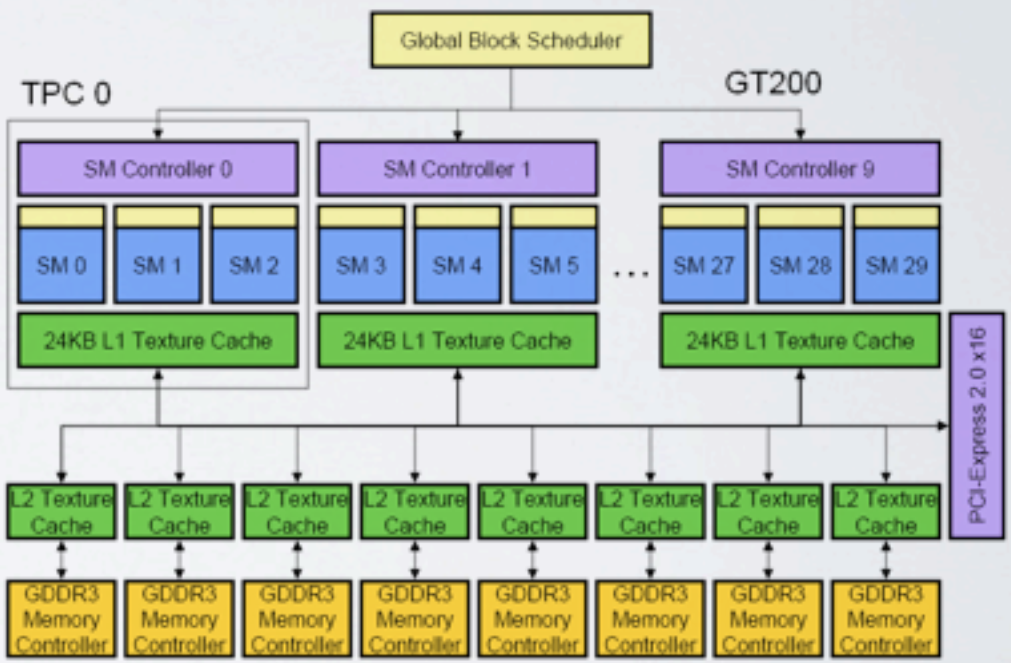
BUILDING BLOCKS

- 10 x TPC: Texture/Processor Cluster
  - 3 x SM: Streaming Multiprocessor (aka Thread Processor Arrays)
    - 8 x SP: Streaming Processors (aka Shader Cores or Thread Processors)



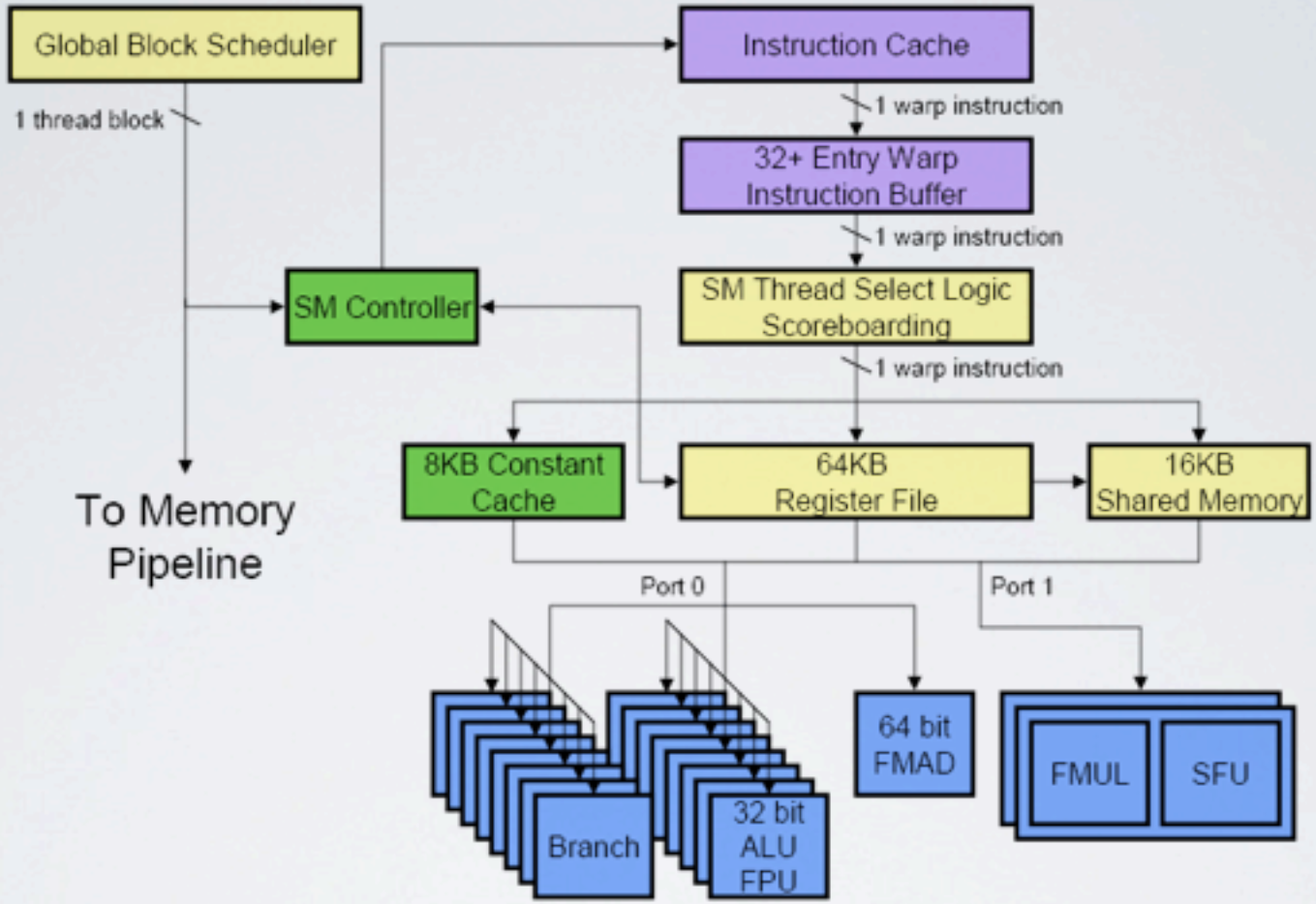
OVERVIEW OF PROCESSING UNITS

- Block scheduler issues *blocks of threads* in round robin fashion to SMs
  - real-time accounting for load balancing and resource exploitation
- SMs fetch instructions
- SPs execute



SOME TERMINOLOGY

- *Core*: SP
- Multiprocessor: SM
- *Thread*: think of every instruction on a SIMD unit as a different thread (ex. 8 quads of XYZW pixels = 32 threads)
- *Warp*: a set of 32 threads
- *Scoreboarding*: A simpe technique to issue data-independent instructions dynamically, out of order



STREAMING MULTIPROCESSOR

Key components

SM AND THREADS

- Highly threaded, single-issue processor with 8-wide SIMD
- 1024 threads concurrently: 32 threads in 32 warps in flight
- Scheduled blocks of no more than 512 threads
- One to two entries per warp in-flight in the instruction buffer
- Issue warps to SPs using scoreboard (no full renaming)
  - Issue logic prioritizes in close proximity to ICache

SM AND MEMORY

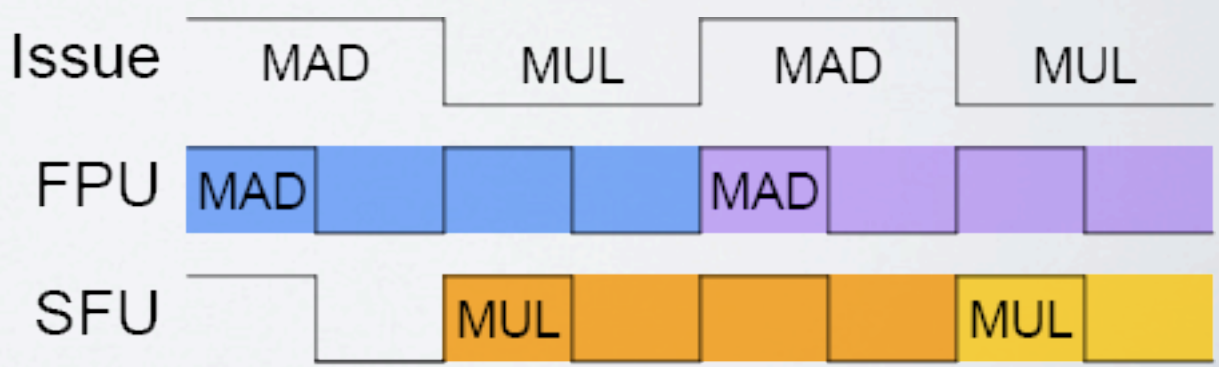
- 16K register file partitioned across SPs
  - Each SP has 2K entry to be used by 128 threads, organized in 16 or 24 *banks*
  - 4-128 entries/thread, statically allocated at compile time
- 16K shared memory
  - 4096 entries organized in 16 banks with 32-bank width
  - Support atomic instructions across threads of a block (ex. CAS)

SIMT

- SIMT: Single Instruction Multiple Thread
- No speculation; wait till address resolution and continue
- Width is not visible architecturally, unlike SIMD which demands packing data into vectors
  - N-way divergent "gracefully" executes serially
- Threads are independent; no register sharing, only shared memory sharing (though warp voting is allowed)

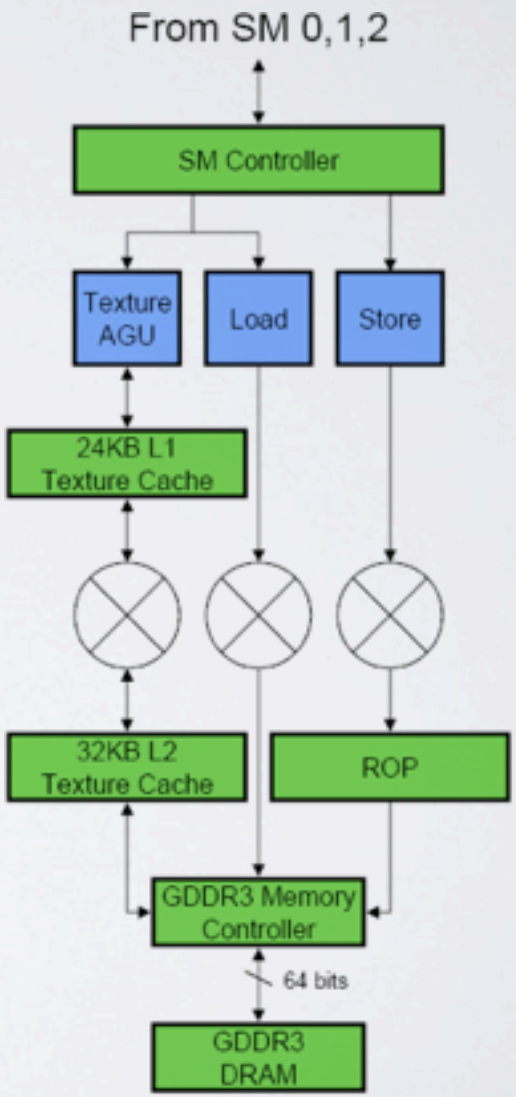
SM EXECUTION UNITS

- Executes at shader-clock speed ; slower core-clock for control logic and storage arrays
- Fused-MAD, single FPU for double precision, Special Function Unit (SFU)
- CPI=4 on ALU
- Dual issue illusion:



MEMORY PIPELINE

- 4B alignment for coalescing
- LD/SD issued in SMs, executed in special units
- Memory accesses issued in warp, executed in half-warp
- ~128-port register-file/shared memory to sustain service rate at low core clock speed



CPU VS GPU

- Many transistors for cache
- Control flow optimizations: Out of order cores, branch predictors
- IEEE compliant, double FP precision
- High clock/memory speeds

- Most transistors for PUs
- Little control flow, shared among chunks of PUs
- Fixed function units,
- Lately IEEE-compliant, limited double FP support
- Medium clock/memory speeds

CPU VS GPU

- 2-3 Cache levels, at least I/O coherent
  - CPU SMPs
- DRAM Dimms, ECC
- Low latency
- Can handle stack-based, pointer-chasing patterns

- Big register files, fast on chip memory, programmer/ compiler managed
  - multi-GPU only for gfx (SLI)
- PCB-mounted high performance RAM, no ECC
- High throughput, memory bandwidth

GPUS FOR GPGPU

The software stack enabling GPGPU:
A case study with NVIDIA's CUDA

ALAS: GPGPU TIME!

- *Intention*: use the programmable part of the GPU for general purpose computing
- *Method*: write programs in a high-level data-parallel language to be compiled by the driver JIT compiler and run on the GPU
- *Result*: applied properly and on proper applications, it can deliver many orders of magnitude of speedup
- *Culprits*: you have to remember all those architecture notes so far

THE GPU DRIVER

- Key component: a Just-In-Time (JIT) compiler for Shading language to GPU ISA
- Focus on extreme register pressure, SIMD optimizations, loop unrolling, scheduling
- Aid the hardware by informing of buffer needs statically
- Interface with the OS, deal with all other stuff the OS wants
- Probably the hardest and most complicated driver in a PC

GPGPU LANGUAGES?

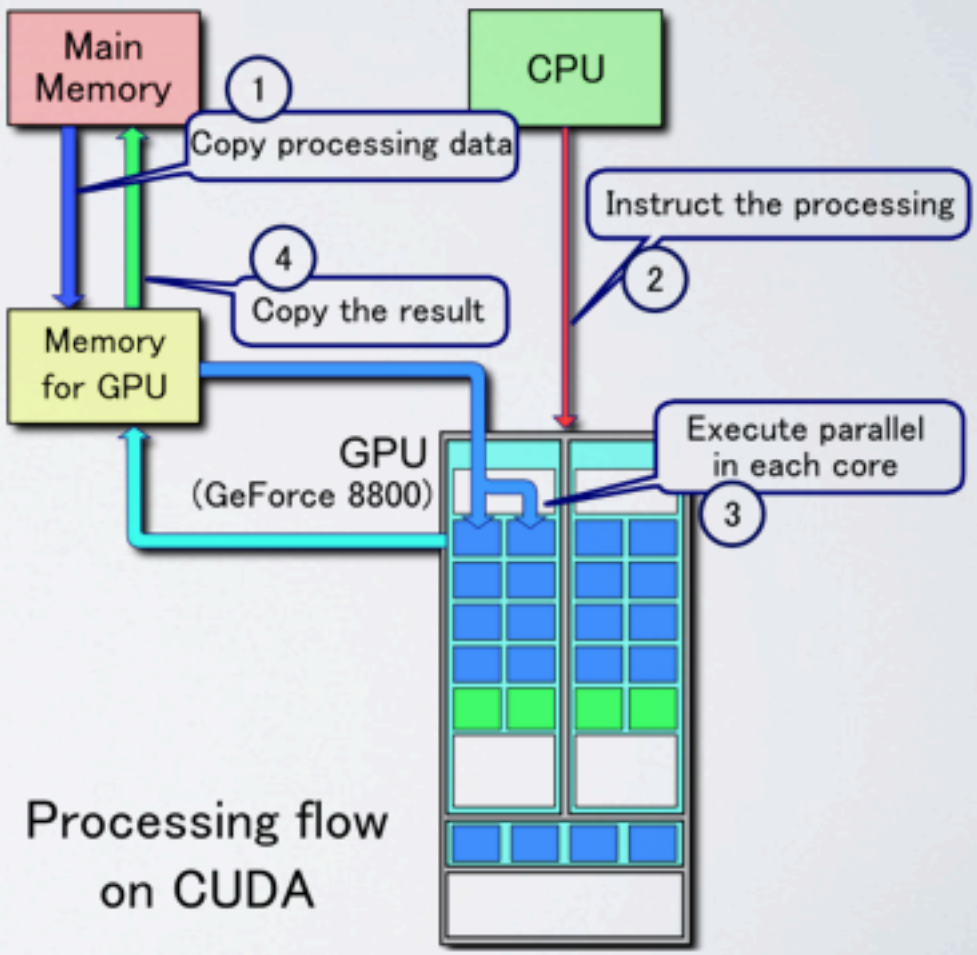
- In the beginning it was all OpenGL-hacks
- No real languages - mostly extensions over C-like languages as compiler directives (pragmas)
  - AMD CTM: very low-level extensions allowing one to build a GPGPU framework
  - Sh, Brook for GPUs: closer to proper languages but unsuccessful

GPGPU "LANGUAGES"

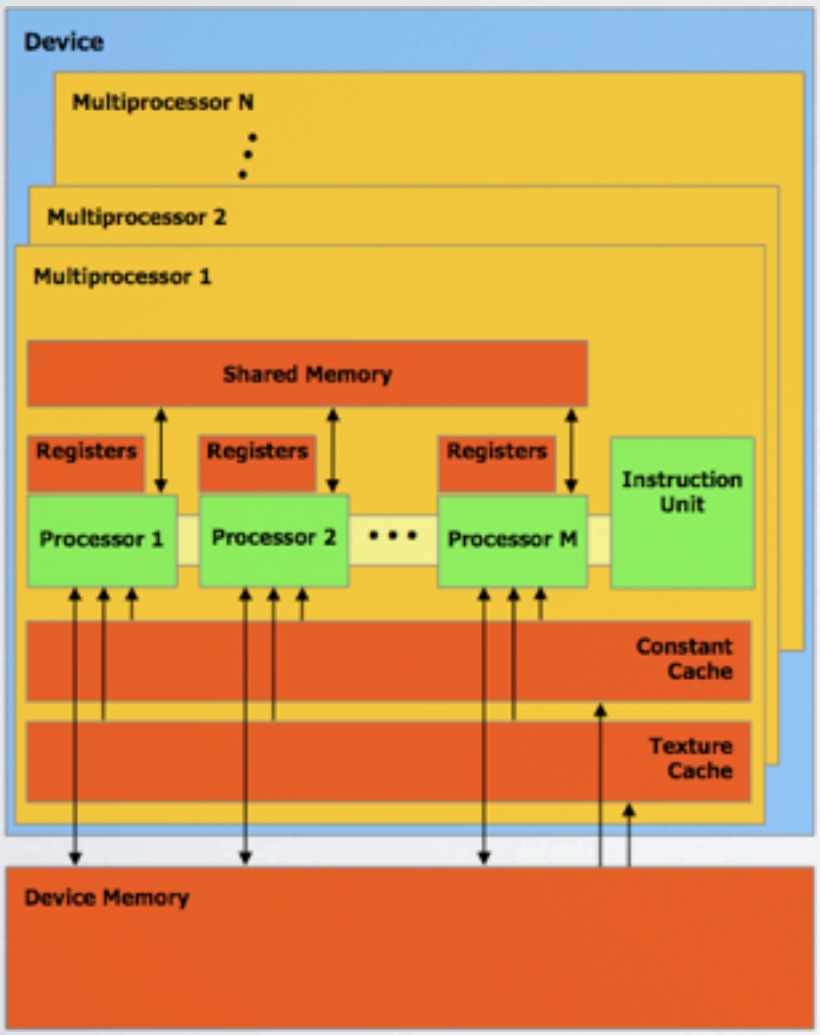
- CUDA (Compute Unified Device Architecture) by NVIDIA
  - Version 2.2 out last Wednesday (1.0 more than two years ago)
  - Applications such as PhysX, Media transcoders, CAD, Research...
  - Not just language extensions: an SDK, including runtime components, the driver, etc
- OpenCL (Open Computing Language) by the Khronos Group
  - Spec 1.0, no implementation yet

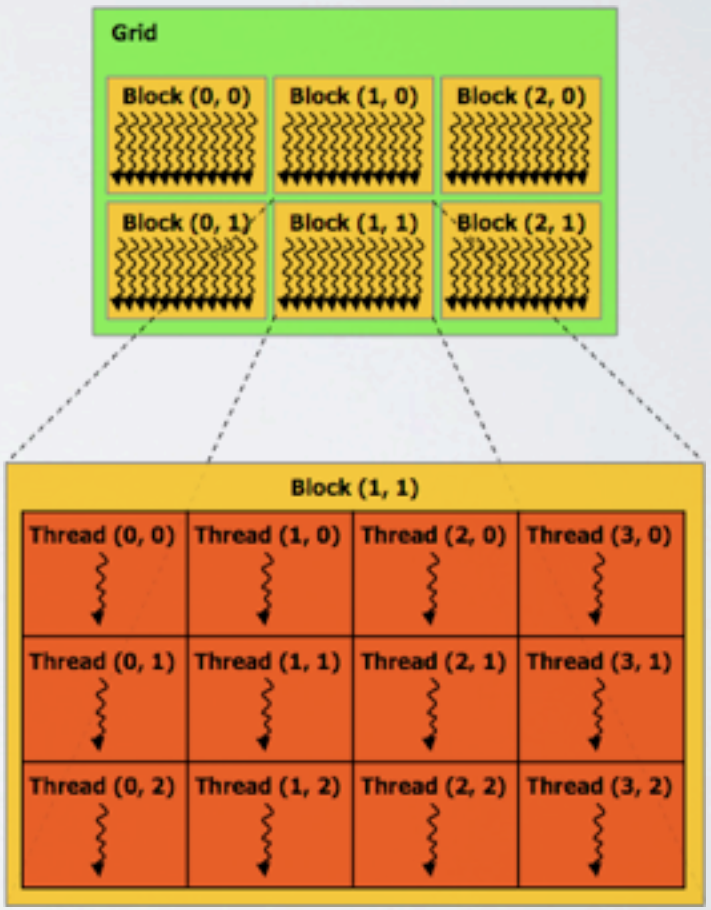
CUDA

- need to express functions to be executed on device
- need data-management routines (explicit memory hierarchy management)
- express parallelism by distributing jobs through thread ids



CUDA MEMORY AND THREAD HIERARCHY





ABOUT THE THREAD HIERARCHY

- Every block must be completely independent from computations in any other block ➡ No hard limit - scalability/portability (Hint: scheduled on different SMs)
- The number of threads per block is limited by device capacity (Hint: executed on same SM)
- Threads of the same block can perform atomic ops and be synchronized over shared memory (*__syncthreads()*)
- The grid structure can be 2D - the block structure can be 3D

ABOUT THE MEMORY HIERARCHY

- Registers are the fastest (~4 cycles/access) ; but are limited (Hint: Register pressure)
- Shared memory can be as fast as registers *if accessed properly (*Hint: memory banks). Unpredictable order and result of synchronous accesses, unless atomic.
- Constant and texture caches are not under software control but are as fast as shared memory (on cache hit)
- Global memory access is slow (~400-600 cycles) - Memory access coalescing enables higher bandwidth

MEMORY ACCESS COALESCING

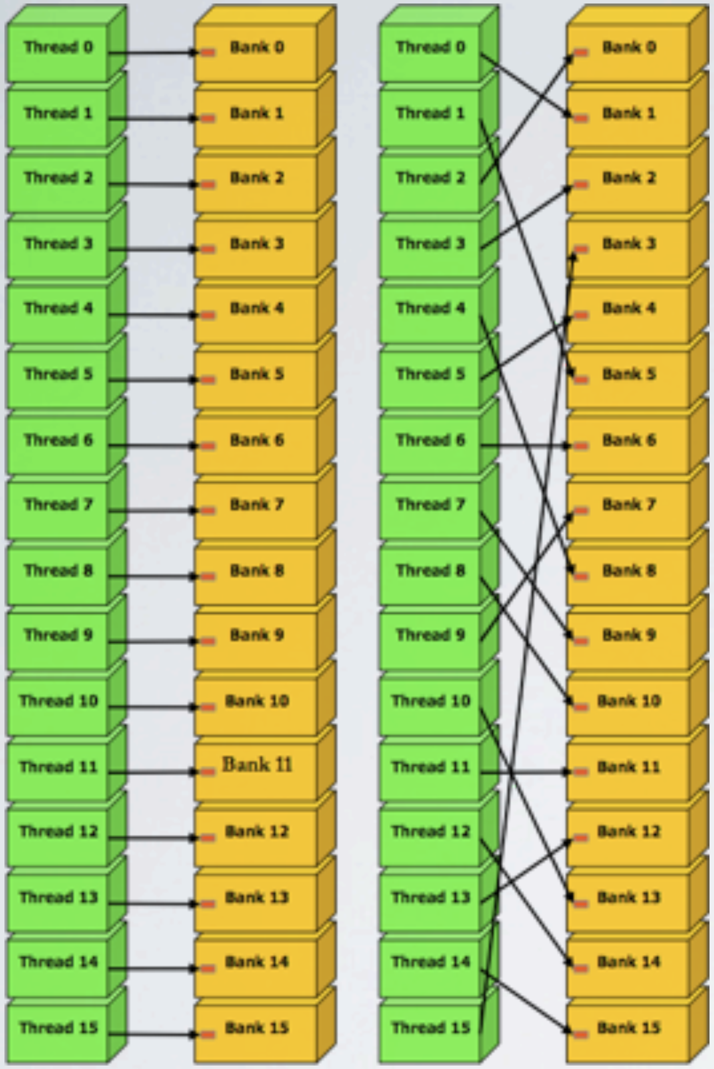
- Remember: at the very bottom, GPUs are wide-SIMD architectures
- Coalescing memory accesses will increase bandwidth from slow global memory: grab as much from the common parts as possible
- Coalescing shared memory access will allow for register-like, high-throughput performance from it: place so bank conflicts, hence serialization, is avoided

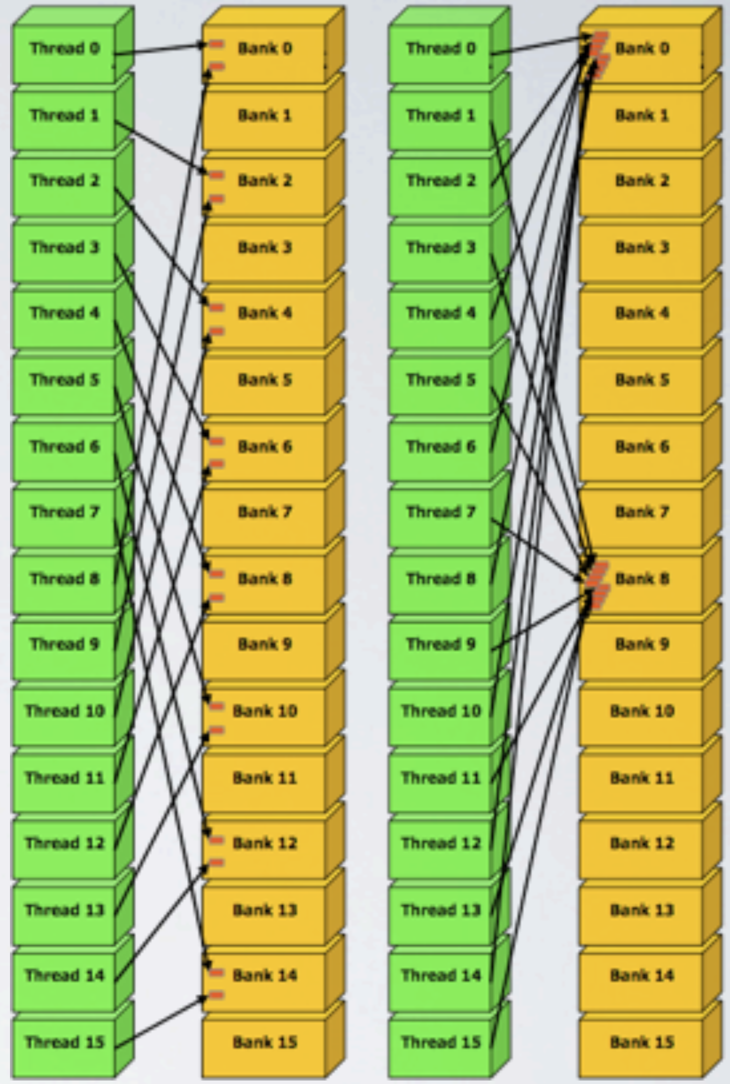
GLOBAL MEMORY ACCESS COALESCING

Left: random float memory access within a 64B segment, resulting in one memory transaction.

Center: misaligned float memory access, resulting in one transaction.

Right: misaligned float memory access, resulting in two transactions.





SHARED MEMORY ACCESS COALESCING

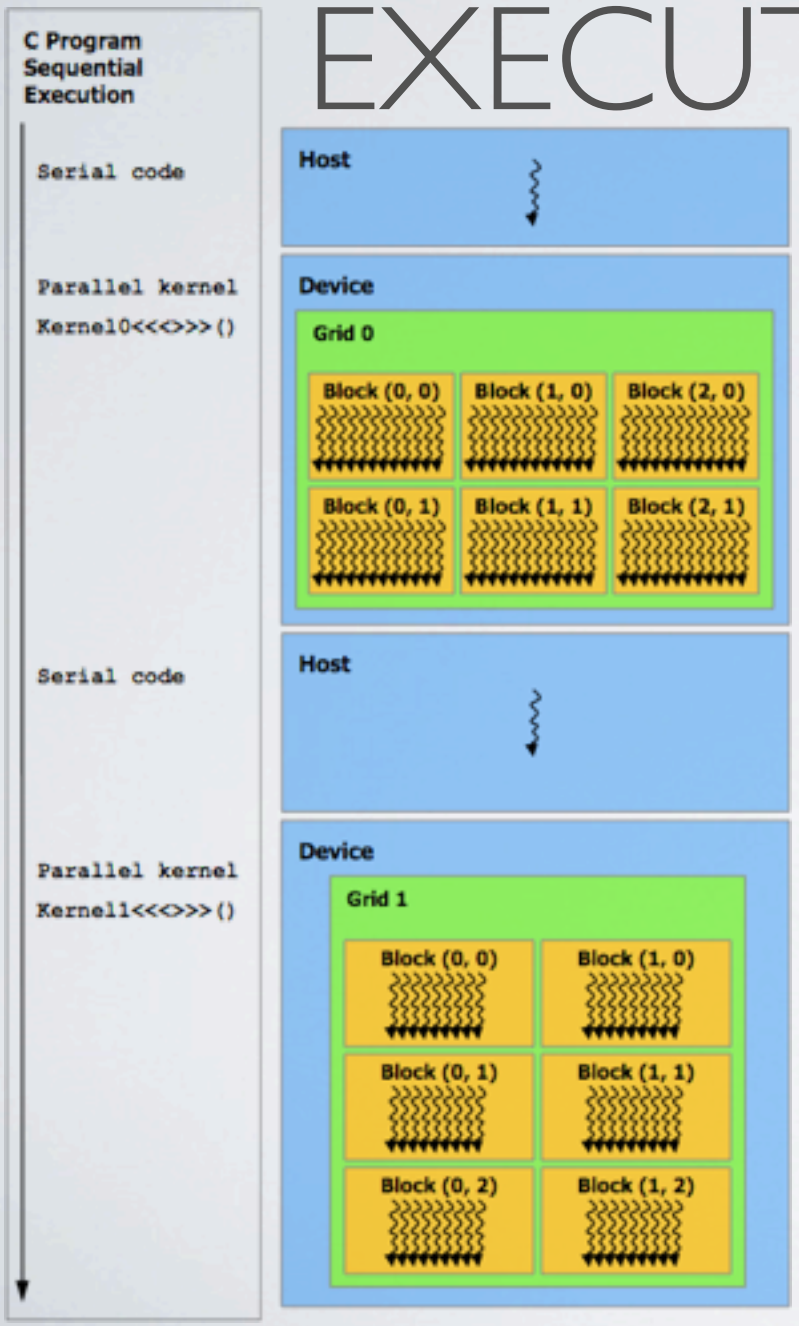
Left: No bank conflicts

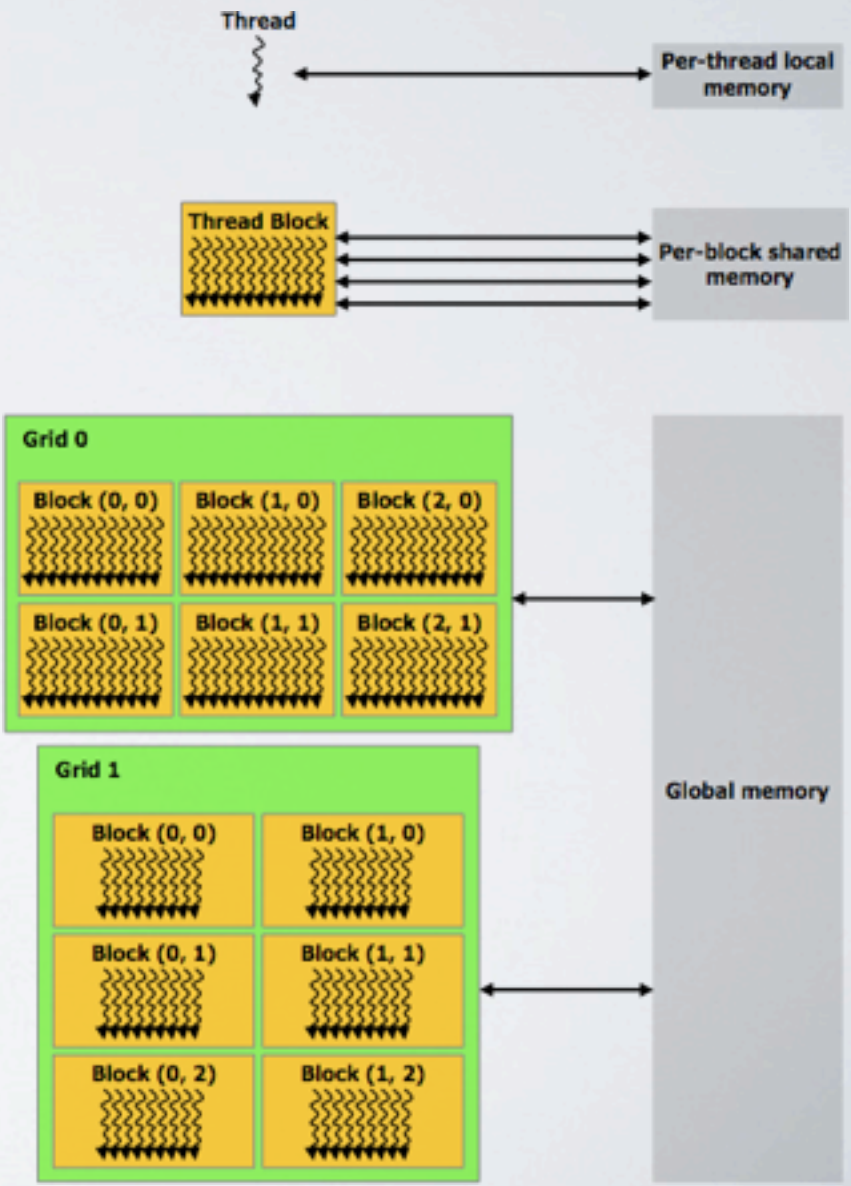
Right: Strides of 2 and 8 words ➡2 and 8 -way bank conflicts

COMMUNICATION AND SYNCHRONIZATION

- No memory barriers: Implement one by adding a new kernel function
- No global synchronization constructs: If you need it, your work is probably not data-parallel enough
- No guarantees in how blocks are scheduled (Hint: Thread scheduler)
- No guarantees in warps-order

CUDA MEMORY AND EXECUTION MODELS





CUDA TOOLSET

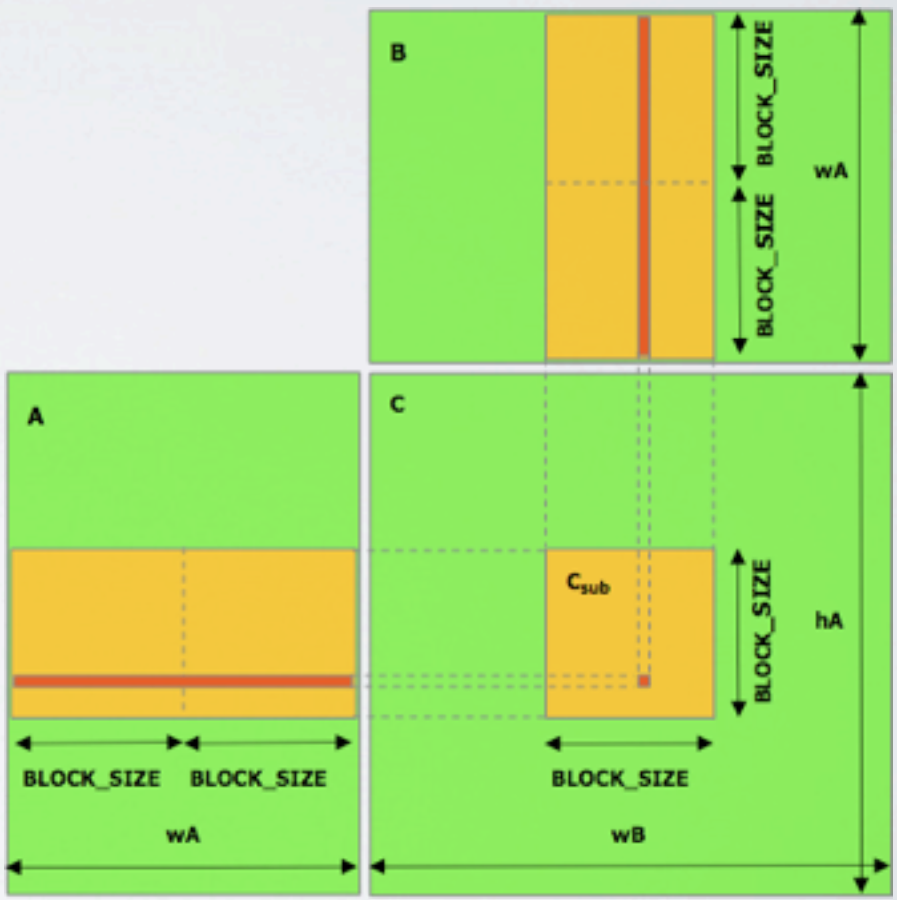
- Driver: CUDA is built as Open64-optimizing compiler extensions
- SDK: nvcc compiler, handy wrapper routines, sample projects
  - Develop in *emulation-mode*: run CUDA code on your CPU
- Profiler: *very* important tool for performance analysis
- Debugger: in emulation mode (Linux only: debug on graphics card execution with GDB)

GREAT EXPECTATIONS

- You can write code probably right away, but sub-optimal
- Descent speedup even for suboptimal algorithm on data parallel application (order of magnitute)
- Two orders of magnitude speedup on optimized algorithm, workload split across blocks/threads and especially memory accesses
- Start at http://www.nvidia.com/object/cuda_home.html#

EXAMPLE: MATRIX MULTIPLY

- A[hA x wA] * B[hB x wB] = C[hA x wC]
- Can be naturally split in blocks
- Copy sub-matrices to shared memory
- Bring in memory once, for more than one blocks



Host mul

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device
// multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A ,
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

Device mul

```
// Compute C = A * B
//   wA is the width of A
//   wB is the width of B
__global__ void Muld(float* A, float* B, int wA,
int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    // ...
```

```
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // 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 the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

OUTRO

- Expect more general purpose implementations
  - Pioneer: IBM Cell
  - Big bet: Intel Larrabee
  - Ray-tracing capable many-many-core GPUs (?)
- Attended Dr. C. Baten's talk?
  - Heterogeneity is a big question
- Expect more language innovation for the shake of GPGPU

WHY GPUS ARE EXCITING

- GPUs push innovation in architecture, and enable new compiler optimizations
- Lack of expressive programming languages opens interesting questions
- OpenCL3, DirectX11 compute
- A teraflop PU for X00$
- Innovation in algorithm design: exciting to rethink well known winners for GPGPU
- 90% of students who enter CS wanted to become game developers
- You can always take a break and play a game

DISCLAIMER

All trademarks, copyrighted material and works of art belong to the respective, referenced owners

REFERENCES

- GPU Computing, J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, Proceedings of the IEEE, Vol. 96, Issue: 5, May 2008.
- A Closer Look at GPUs, K. Fatahalian and M. Houston, Communications of the ACM, Vol. 51, Issue: 10, October 2008.
- Memory access coalescing: a technique for eliminating redundant memory accesses, Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, Pages: 186 - 195
- NVIDIA's 1.4 Billion Transistor GPU: GT200 Arrives as the GeForce GTX 280 & 260 at http://www.anandtech.com/video/showdoc.aspx?i=3334&p=2
- NVIDIA's GT200: Inside a Parallel Processor @ http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

MEDIA

- http://pc.ign.com/objects/901/901016.html
- http://vr-zone.com/articles/nvidia-geforce-gtx-280-preview/5872.html?doc=5872
- http://www.tomshardware.com/news/GPU-videocard-Nvidia-AMD,6533.html
- http://arstechnica.com/gaming/news/2008/06/gaming-expected-to-be-a-68-billion-business-by-2012.ars
- http://en.wikipedia.org/wiki/CUDA
- http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf