Intro to GPGPU
General Purpose GPU programming

Massimo Coppola
04/05/2018

GPU Computing

- The need for efficient specialized processing of 3D meshes promoted the adoption of the SIMD programming model

- How the model was born, evolved over time

- What are GPUs good at?
  - Large data sets
  - Arithmetic intensity = High compute/IO ratio
  - Minimal control flow or recursion
  - High locality

The birth of Graphic Processing Units

- The graphics pipeline
  - General methodology to produce graphic output on raster devices like computer displays
  - Start from elementary data (vertexes) and transform them into pixels
  - State of the art evolved over the years, to possibly very complex structures
    - Cfr. OpenGL 1.1 state machine
  - We only survey the basic principles
- Graphics pipeline, or its stages, can have both SW and HW implementation
- Tradeoff between flexibility and performance

The objects

- Vertex : a point in a coordinate system
- Primitive : graphic object comprising one or more vertexes, possibly other parameters
- Pixel : image element in a raster display

- Coordinate systems for Vertexes, Primitive, Pixel usually do not coincide
- They have typically different dimensionality
  - E.g. render 3D space on a 2D display

- Widespread use of homogeneous coordinates
  - Represent points in 2D spaces with 3 coordinates, and points in 3D spaces with 4-dimension coordinates
  - Allow representing linear affine transformations and projections as linear operators → implemented as matrix multiplication
  - Common, very efficient execution of graphic transformations

Elementary Graphics Pipeline

1. Vertex generation
2. Vertex processing
3. Primitive generation
4. Primitive processing
5. Pixel generation (Rasterization)
6. Pixel Processing
7. Pixel writing

- Some steps are more deeply customizable
- Some steps are efficiently realized in HW

Example

1. Vertex generation
   - retrieve/generate coordinates, apply geometric transformation
2. Vertex processing
   - Apply/attach visualization parameters to vertexes, apply per-object transformations
3. Primitive generation
   - Group connected vertexes and turn them into squares, spheres, surfaces, lines …
4. Primitive processing
   - Apply shading models, colors, textures custom transformation to primitives
5. Pixel generation (Rasterization)
   - Slice primitives according to the output device resolution and features
   - Compute/interpolate texture pixels from texture memory matching with primitive coordinates, to define each pixel characteristics in the slices
6. Pixel Processing
   - Process pixels accordind to lighting models, (anti) aliasing and other postprocessing techniques
7. Pixel writing
   - Framebuffer operation, appropriate memory format (e.g. alpha channel)

Evolution and tranformation of GPUs

- From 1985 (e.g. Commodore Amiga) to 1990 (S3 chips and followers) and beyond, 2D and then 3D accelerated units spread in the personal computer market
  - Early experiences at Xerox PARC in 1975
  - Driven mainly by the game market
  - Less by Windowing systems, professional graphic use
- More and more specific stages in the pipeline implemented in HW on a chip of the graphic device
- In the end, all stages of a 3D pipeline implemented in HW
- **Load balancing among the stages and flexibility become issues for all-HW implementation**

Load balance in the pipeline

- More pixel than raster elements (slices of primitives)
- More raster elements than vertexes
- Expected primitive distribution, surface hiding and other masking effects can affect this balance

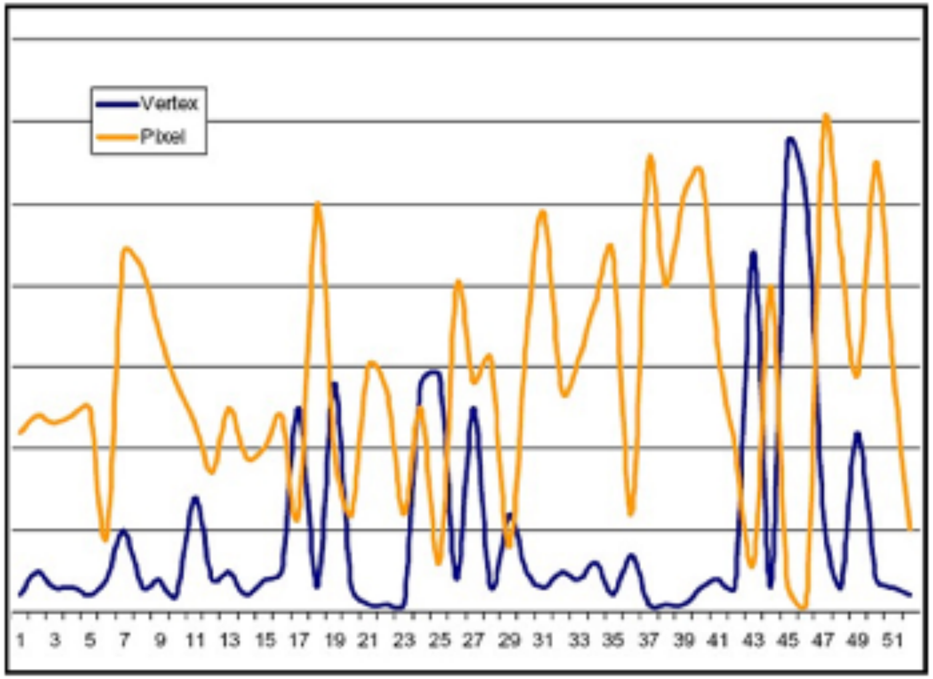


Figure 14. Characteristic pixel and vertex shader workload variation over time

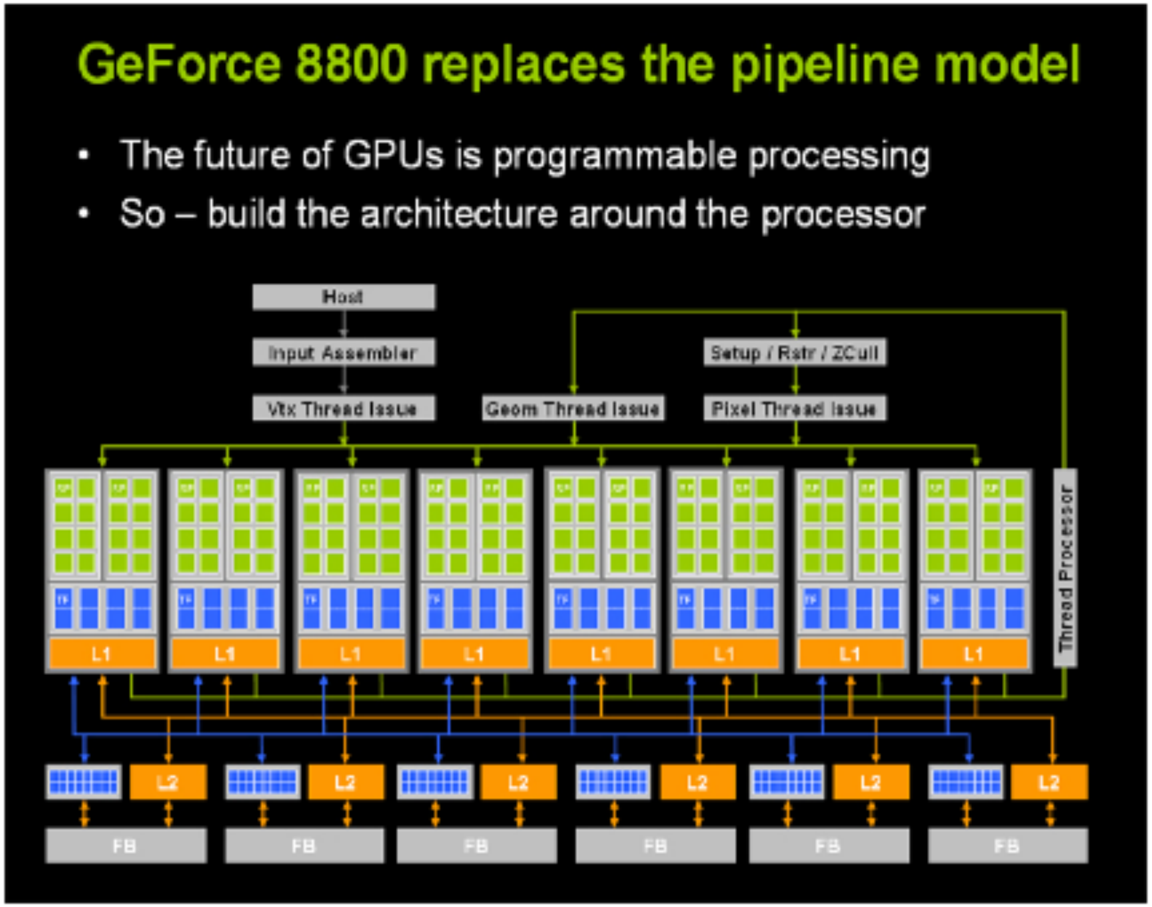
Image from NVIDIA GeForce 8800 architecture documentation, 2006

Push toward unification

- A fixed number of vertex units and pixel units leads to **poor resource use under varying workloads**
- Fixed, HW-cabled functionalities are easily reproduced in SW
  - no generic CPU functionalities needed
- **Special units gradually replaced by unified units** alike to stream processors, with limited programming capabilities
- Allocation of code to stream units initially done by specialized SW = **graphic drivers**

First “programmable” GPUs

- Replace the graphic pipeline in the HW
- Maintain some special purpose units in HW
  - e.g. texture caching and sampling
- Architecture optimized for streaming
  - Custom RAM bus
  - No read/write conflicts
  - Small caches
  - High on-chip ALU/memory ratio
  - Single precision, non IEEE floating point



GeForce 8800 replaces the pipeline model

- The future of GPUs is programmable processing
- So – build the architecture around the processor

Example from GeForce 8800 docs

GPGPU

- General Purpose Graphic Unit Programming
- More and more graphic cores, and increasing core computing power
- **People started tapping into the graphic unit** via OpenGL primitives
  - Exploit the computational semantics of specific graphic operations to achieve access to the HW
  - Tasks fit for stream processing : physics, image manipulation, large data with few dependencies
- GPGPU research area was born
  - Physical simulation coupled with rendering
  - Textures and vertexes (read-only) are input streams
  - Need to write results !
    - Copy framebuffer (write-only) to texture after computation
    - Skip last pipeline stages and save results to texture memory (*stream output* in DirectX10)

New, programmable GPUs

- GPU producers understood the market value
  - GPU became more programmable
  - General programming issues accounted for
    - Double precision IEEE f.p. arithmetic
    - More efficient branches in GPU code
- **Architecture is still optimized for streaming**
  - The model exposed is very much SIMD like
  - No support for reading/writing the same memory area
  - No or limited support for communication among code instances
    - to avoid synchronization and pipeline stall detection logic

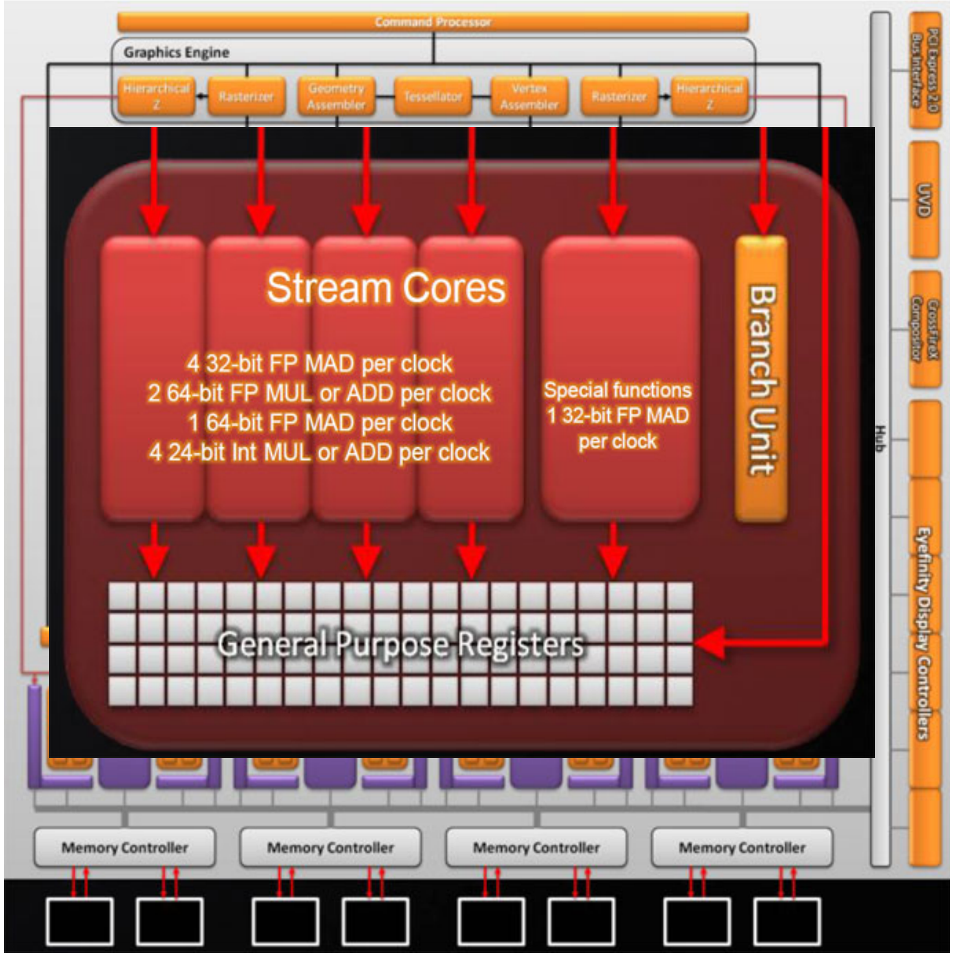
- GPUs are optimized for long computation run with reduced dependencies
- CPUs for general access patterns and concurrency

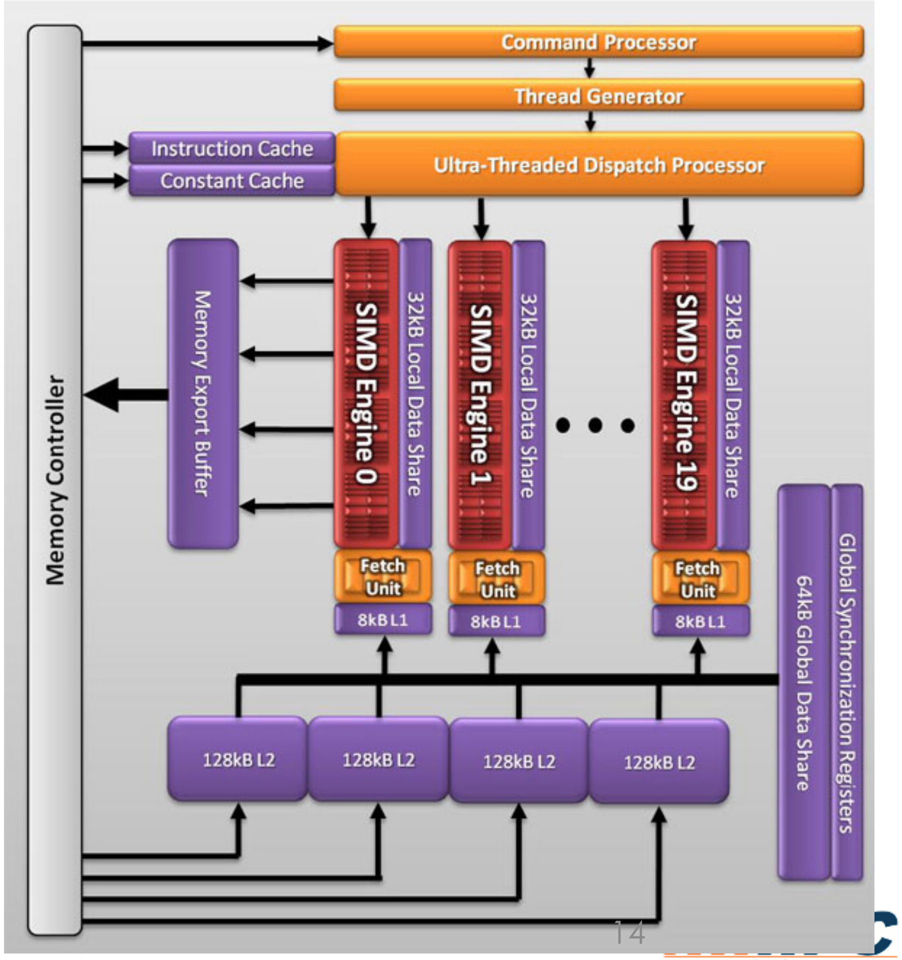
GPU HW optimizations

- **Very large RAM bus**
  - Multiple data transfers per cycle
  - On rising and falling clock edge
- High Bandwidth translates to **low latency for sequential** (or easily predictable) **access patterns**
  - Parallel units in a GPU exploits different data items from a set of common input data streams
- **High ALU density**
  - high number of ALU/FPU units per chip, working in parallel
- **Cores are grouped** as thread processors
  - cores in a same thread block share the same program code
  - and groups of ALUs/FPUs cores sharing the control unit
    - cores either process or skip instructions → branches are inefficient
  - thus thread processors' cores share code **and** program flow
  - sometimes available: shared set of registers and caches
- **Different threads processors are truly independent**
  - also a constraint: you can't synchronize them

Hardware Model

ATI “Cypress” RV870





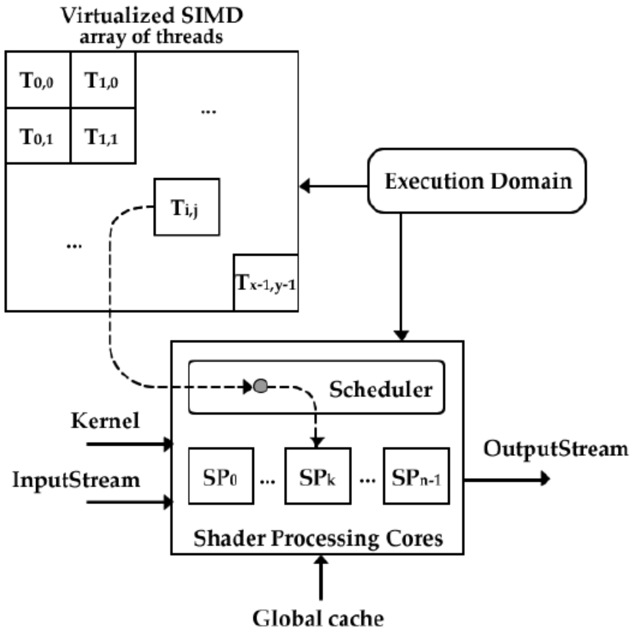
Computational Model

- Stream Computing
  - SIMD-like programming model
  - Multiple processing units

- Non-determinism
  - how data in streams gets processed by the cores is left to the board firmware

- The computation of each core is driven by a program, ***kernel***

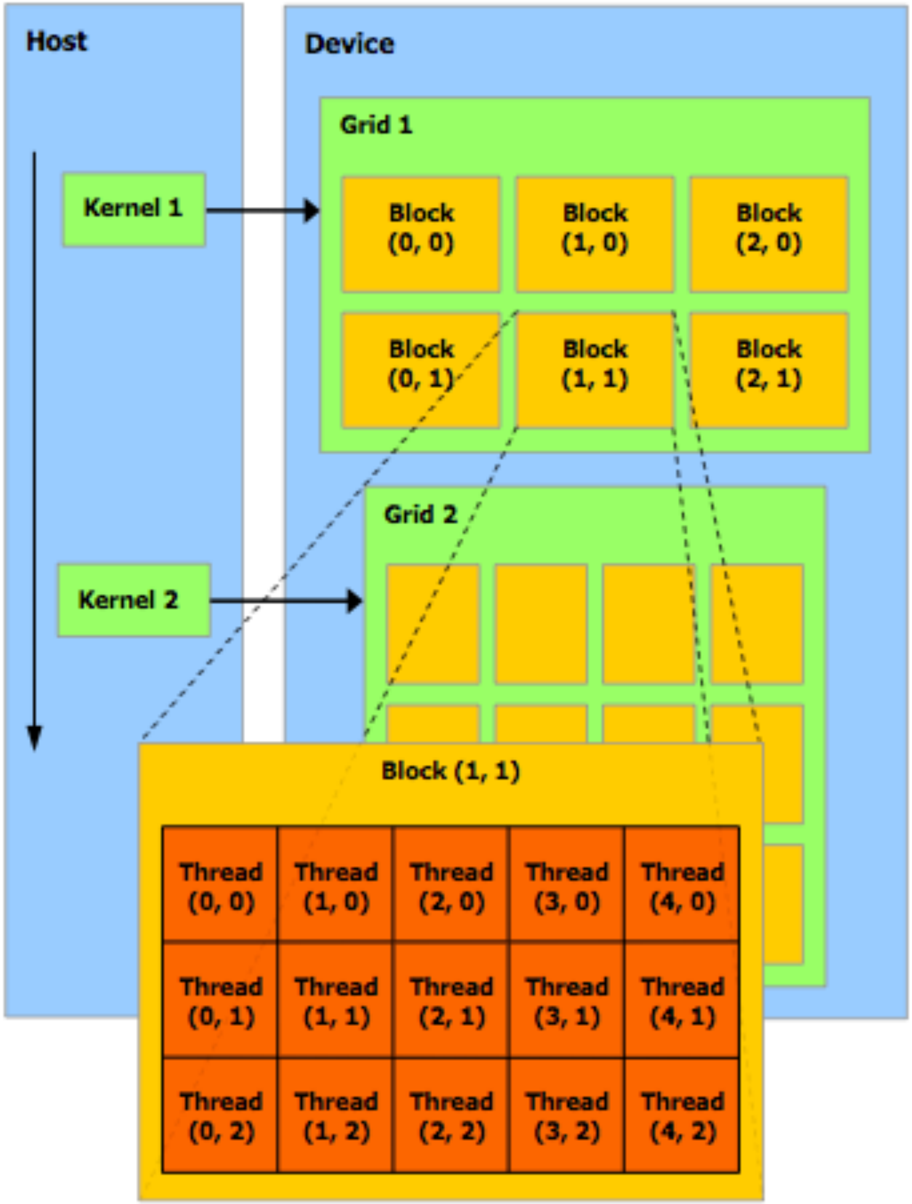
- The GPU infrastructure is responsible for assigning cores to kernels
  - each running instance of a kernel is called ***thread***
  - each thread has an associated set of output locations in the GPU memory referred as the ***domain of execution.***



(Early) Proprietary Programming Models

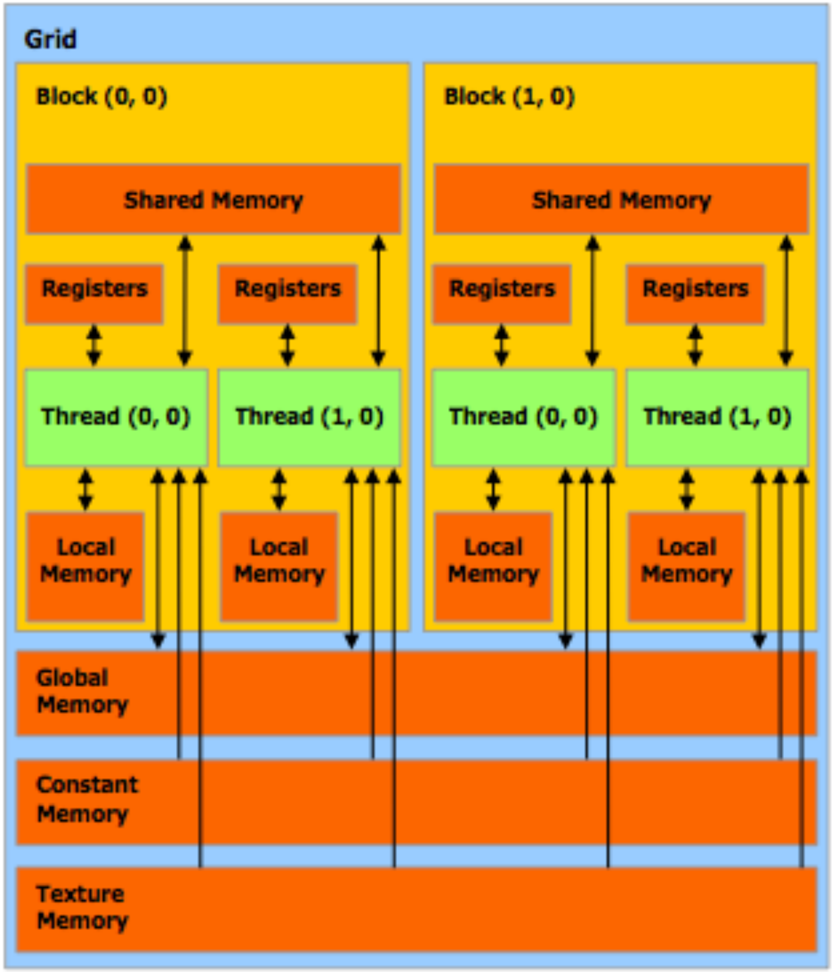
- Brook+ and CUDA
  - Provide sound language abstractions to define computational kernels
  - In a subset of standard sequential languages
    - each one assigned to one or more thread processors
  - Main issue is to define in which memory space each data/variable is actually kept

- Each kernel is mapped onto one or more thread blocks
- Each Block can execute several sub-computations
- Kernel instances (threads) in a thread block can be interleaved or parallel



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

- The kernel instance in each core can access several spaces
- Language qualifiers on functions and variables
- Concurrency issues → avoid dependencies



A thread has access to the device's DRAM and on-chip memory through a set of memory spaces of various scopes.

More general Programming Models

- **RapidMind** (now discontinued)
  - Language-based approach focused on portability
  - Common set of SIMD-like primitives compiled to
    - GPUs
    - Cell Multicore
    - X86 multicore CPUs
  - Interesting idea → acquired (by Intel) in 2009
    - The team was merged to the CT Intel project, producing the Array Building Blocks (ABB) in 2010
    - CT project was discontinued in 2012 and joined within the Intel TBB and Cilk projects (Cilk is based on ABB)
    - Cilk support recently discontinued in major open-source compilers (GCC, LLVM)

More general Programming Models

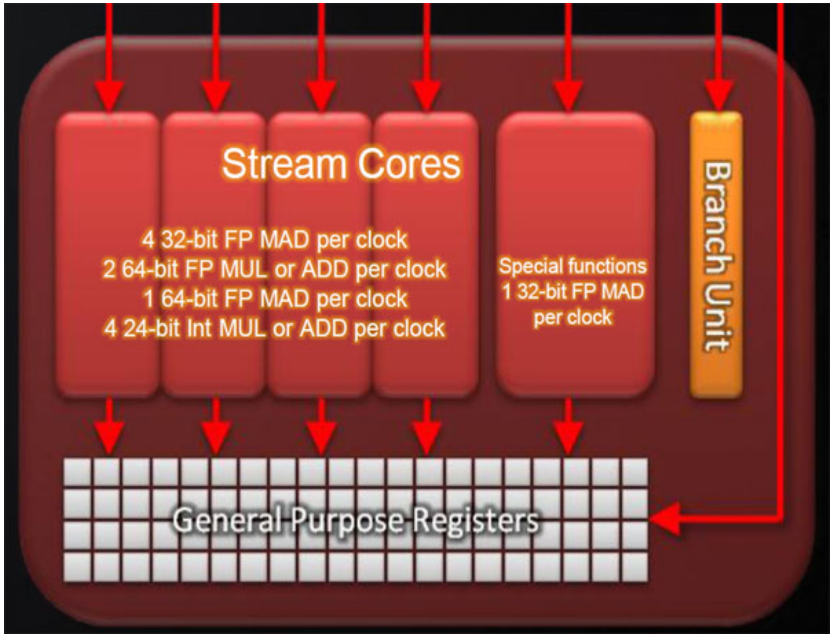
- **OpenCL**
  - focused on computational exploitation of GPU,
  - evolving API and language, follows up on CUDA and Brooks+
  - Evolves toward single-source CPU/ GPU language (SYCL++)
  - Aims at much broader device compatibility
    - FPGA, DSP, Embedded systems

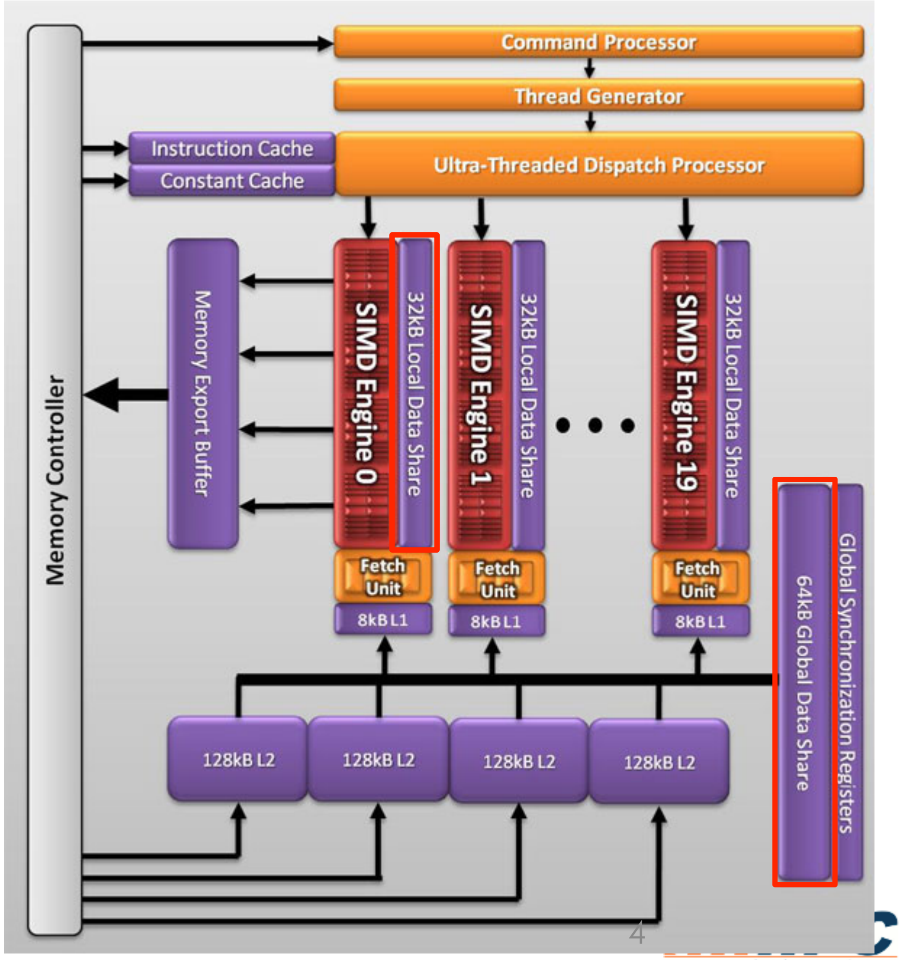
(discrete) GPU and CPU interaction

- The main limit of conventional GPU approach
- Interaction with the CPU bus is a bottleneck
  - PCI bus (PCI-X …) is fast, but slower than the memory interface of the GPU
  - CPU/GPU data exchange rate and overhead is influenced by
    - driver/OS management
    - hardware capability (is DMA controlled by both sides?)
- To scale up you needed
  - an ALU-intensive, regular problem
  - infrequent interaction with the CPU
  - Salability improves together with the efficiency of asynchronous interaction with the CPU
- Playground changes when memory is shared between CPU and GPU (virtual or phisical sharing)

Hardware Model

ATI "Cypress" RV870


Stream Cores
4 32-bit FP MAD per clock
2 64-bit FP MUL or ADD per clock
1 64-bit FP MAD per clock
4 24-bit Int MUL or ADD per clock
Special functions
1 32-bit FP MAD per clock
Branch Unit
General Purpose Registers


Command Processor
Thread Generator
Instruction Cache
Constant Cache
Ultra-Threaded Dispatch Processor
Memory Controller
Memory Export Buffer
SIMD Engine 0
32kB Local Data Share
SIMD Engine 1
32kB Local Data Share
• • •
SIMD Engine 19
32kB Local Data Share
Fetch Unit
8kB L1
Fetch Unit
8kB L1
Fetch Unit
8kB L1
64kB Global Data Share
Global Synchronization Registers
128kB L2
128kB L2
128kB L2
128kB L2