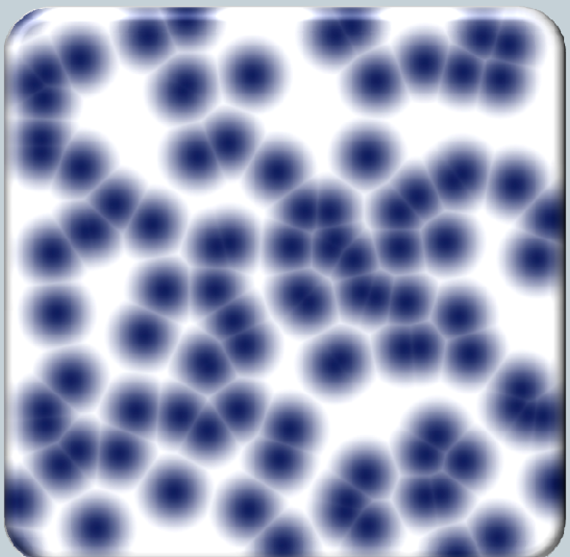
A -Very- Brief Introduction to (GP) GPU

with Applications to Voronoi and Related Problems

M. ADİL YALÇIN



Summary

Note: No summary can be given in 50 minutes

- Introductory References for GP-GPU
  - SIGGRAPH GPGPU Course 2004
  - SIGGRAPH GPGPU Course 2005
  - IEEE Visualization 2005 Tutorial
  - … and many more
    - (these sessions/workshops lasted many hours)
- >1000 of slides
- >100 of academic papers / whitepapers

GPU References

- For graphics rendering related algorithms, see:
  - GPU Gems Series (all available online)
  - NVIDIA / ATI / DirectX SDK's & Developer Sites
  - ShaderX Series
- If you want to learn OpenGL:
  - OpenGL Bible 4th Edition
  - OpenGL Official references (GL / GLSL / Extension Specs)

GPU References

- Real-Time Rendering (Selected as course book in other uni's)
- The CG Tutorial (Selected as course book in other uni's)
- Non-Photorealistic Computer Graphics - Modeling, Rendering and Animation
- Texturing & Modeling - A Procedural Approach

(These are old, but can be fundemental)

- Graphics Gems Series
- Game Programming Gems Series

And More…

Web Resources

- Game Developer Magazine
- Gamasutra
- GameDev.net

- An extensive list in:
  - http://www.realtimerendering.com

GP-GPU comes in handy when you do…

- Computational Geometry
  - Voronoi diagrams
  - Mesh refinement / LoD
- Novel Graphics Algorithms
  - Supporting Realtime Algorithms
    - Example: HDR and Tone Mapping
  - Global Illumination
    - Ray Tracining / Photon Mapping
- Computer Vision
- Image Processing
- Physics Simulation / Physically-Based Simulations
  - Contact / intersection / collusions – Constraint System solvers
  - Fluid dynamics, boilingm etc
- Signal / Audio Processing
- … The limit may be the sky itself (You can animate 3d clouds-smoke on GPU by GP-GPU techniques :))

The API's for GP-GPU

First Approach:

You can use what you already have:

- OpenGL
- DirectX

- Even the all-fixed-function pipeline is open to some interesting algorithms.
  - Ex: Voronoi diagrams construction using regular meshes!
    - Mentioned back in 1997: OpenGL Programming Guide 2nd Edition
    - An academic study followed…

Voronoi Diagrams – GPGPU First Steps…

- Note : When you use GP-GPU, you –generally- get discrete results. (Details will be explained soon)
- Map each Voronoi seed to a mesh
  - Points -> Cone
  - Line -> Tent
  - Curved surfaces, …
- Z-buffer will manage the closest-seed information.
- You render each Voronoi seed once. => O(N)
- Errors: Distance approximation / combinatorial
  - Error sources: hardware limitations / mesh generation

The API's for GP-GPU

Second Approach:

Use new API's that do not focus on "rendering“, yet are designed for underlying GPU hardware.

- NVIDIA's CUDA / ATI's STREAM
  - Each provides its own tools for performance measuring.
  - These API's are vendor dependant.
- KHRONOS' OpenCL

  - **Note :**
  - Intel also develops a hybrid CPU-GPU architecture: **Larrabee**
  - Watch Tom Forsyth's blog for more information.

An Overview of OpenCL

- It is a : **Heterogenous Parallel Programming Standart.**
- It is a : Hardware abstraction layer in a C-based Programming Interface
- Proposed by Apple, specification relased in <1 year, conformence tests will be released very soon.
- Aim: Program for all computational units using a unified API.
- What it does:
  - Executes compute kernels
  - Manage scheduling, compute, and memory resources
- Architectural Models for:
  - Platform, Memory, Execution, Programming
- This is only a summary of summary.

DirectX vs. OpenGL : Does it matter?

- Answer : Mostly, No.

- OpenGL is "not" less capable than DirectX.
- OpenGL is open to extensions (and extensions can be developed if required –Not saying it is “easy”).
- Doom3 / Prey has OpenGL renderers (and can look facinating).
- If you want new features in DirectX, wait for the next release. If you want new features in OpenGL, specify extension, and update the driver (yet, not easy).

How OpenGL progresses?

- (Vendors ->) Ext -> ARB -> To OpenGL Core
- Until now, aimed backward compability:
  - What runs in OpenGL 1.1 still run in todays hardware/drivers.
- Backward compability was expected to be broken with OpenGL 3.0, but only a deprecation was introduced.
- Forget glBegin() / glEnd blocks
  - CPU<->GPU communication is costly.
- Forget automated standart shading methods.
  - Gouraud/phong shading => You can implement these and many other lighting models / BRDF in GPU yourself.
  - Actually, all fixed-function will be deprecated in a few years.

Some Observations

- The new standarts API’s are mostly similar to existing standarts.
- Mostly, the same "fundemental" restrictions or optimizations apply between API's.
  - Ex: Use batches of as large chunks as possible.
- Mostly, the underlying hardware is the same.
  - Differences can be minor, but also can determine the winner platform.

Graphics Related Academic Research

- Stanford
- MIT
- North Carolina
- Columbia, Cornell, Virginia, Princeton. California, Utah …
  - Many fascinating personal & academic projects.
- Okan Arıkan (Texas) : Ambient Occlusion (on GPU)
  - Mathematically heavy/interesting, yet practical lighting model
  - CryEngine made it popular.
- Subsurface Scattering of Light
  - Realistic Skin Rendering
- Fur / Cloth
  - Physical vs Rendering model generally separated
  - Now, we can merge these into a single computational platform.
  - Achieve more realistic / faster results.
- Fluid

Graphics Related Industry Research

- Mostly lead by NVIDIA & ATI
  - Note: NVIDIA offers financial support programs to graduate students. (no more support in 2008, see economic crisis.)
- Also, Intel, PowerVR, Texas Instruments, etc

- Production - Animation Studios
  - PIXAR => RenderMan – Reyes Rendering Architecture
- Game Development Studios
  - John Carmack (Quake/Doom/etc) is a contributor to some OpenGL Extensions.
  - Many leading studios work in close relation with hardware vendors (performance do matter a lot to achieve AAA titles, so does drivers).

Why prefer GPU computations?

- GPUs are fast…
  - 3 GHz Pentium4 theoretical: 6 GFLOPS, 5.96 GB/sec peak
  - GeForceFX 5900 observed: 20 GFLOPs, 25.3 GB/sec peak
    - The latest hardware (G92) approaches 500 GFLOPS
- GPUs are getting faster
  - CPUs: annual growth ; 1.5 × à decade growth ; 60×
  - GPUs: annual growth > 2.0 × à decade growth > 1000
- Modern GPUs are programmable.
- Modern GPUs support high precision data/arithmetics (IEEE 32-bit floating points)

(Adapted From SIGGRAPH 2004 course notes)

Is GP-GPU computations easy?

- Yes. &. No.

- Hardware is driven by "graphics" community (can it change with GP-GPU ???)
- Cannot easily "port" code from CPU to GPU.
- Underlying GPU architectures are:
  - Inherently parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret

(Adapted From SIGGRAPH 2004 course notes)

Problems : from CPU to GPU

- Only some types of problems can benefit GPU architecture
- Data Parallelism : Emberassingly Data-parallel problems
  - Data "should" also be homogenous.
    - Vector Processing : Small-scale parallelism
    - Data Layout : Large-scale parallelism
- ALU-heavy computations per data.
  - Memory latency hidden as computations get more complex.
  - Number of instructions per data was very limited, but current progress in HW mostly removed this problem (10000's of ins.)

(Adapted From SIGGRAPH 2004 course notes)

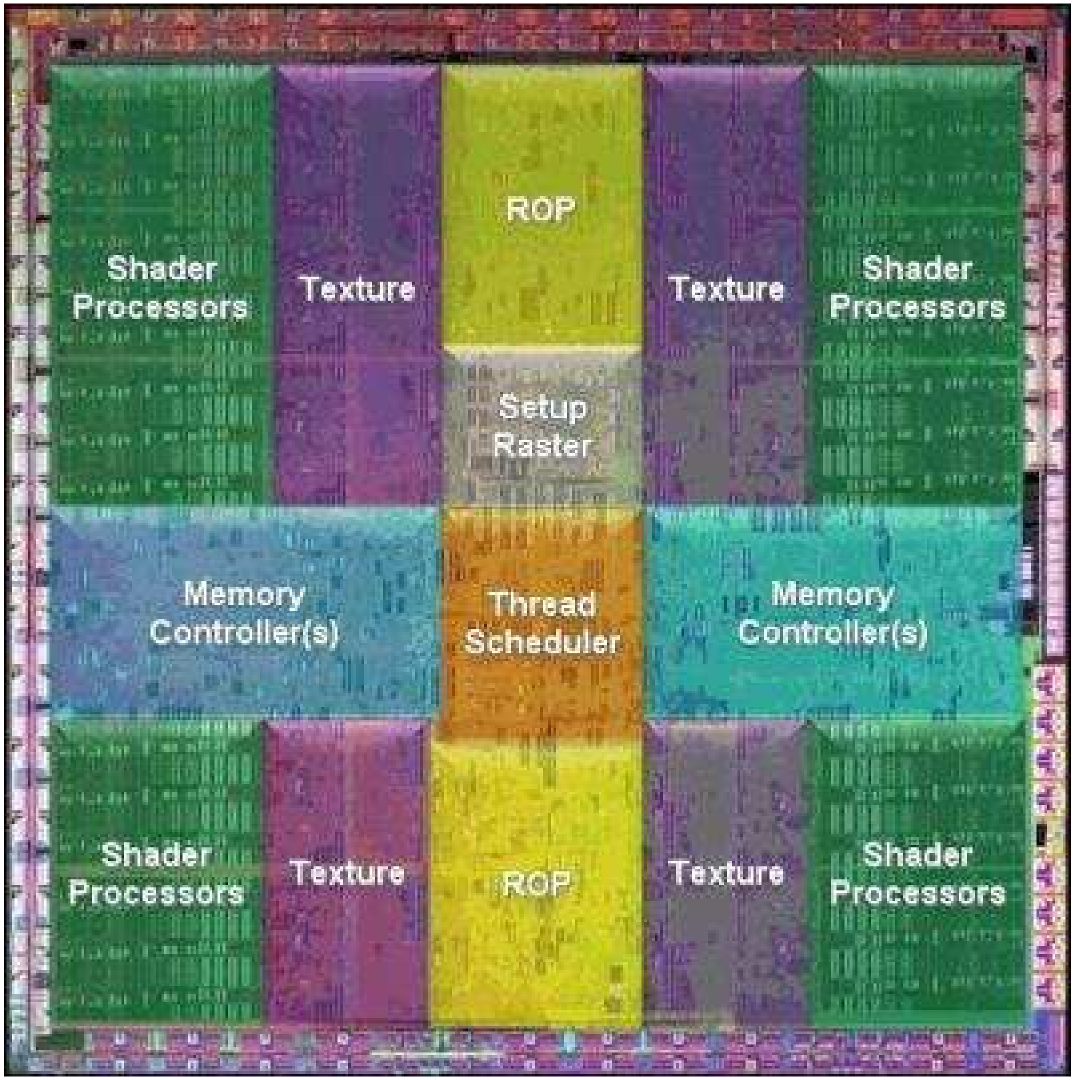
Problems : from CPU to GPU

- Streams
  - Collection of records requiring similar computations.
    - These should be not be dependant on each other.
- Kernels
  - Functions applied to each record in stream.
    - Shaders in OpenGL/DirectX. Kernel C-functions in CUDA
- Simulations
  - Mostly made up of several steps, possibly updating entire grid.
  - The steps are your kernel functions.

(Adapted From SIGGRAPH 2004 course notes)

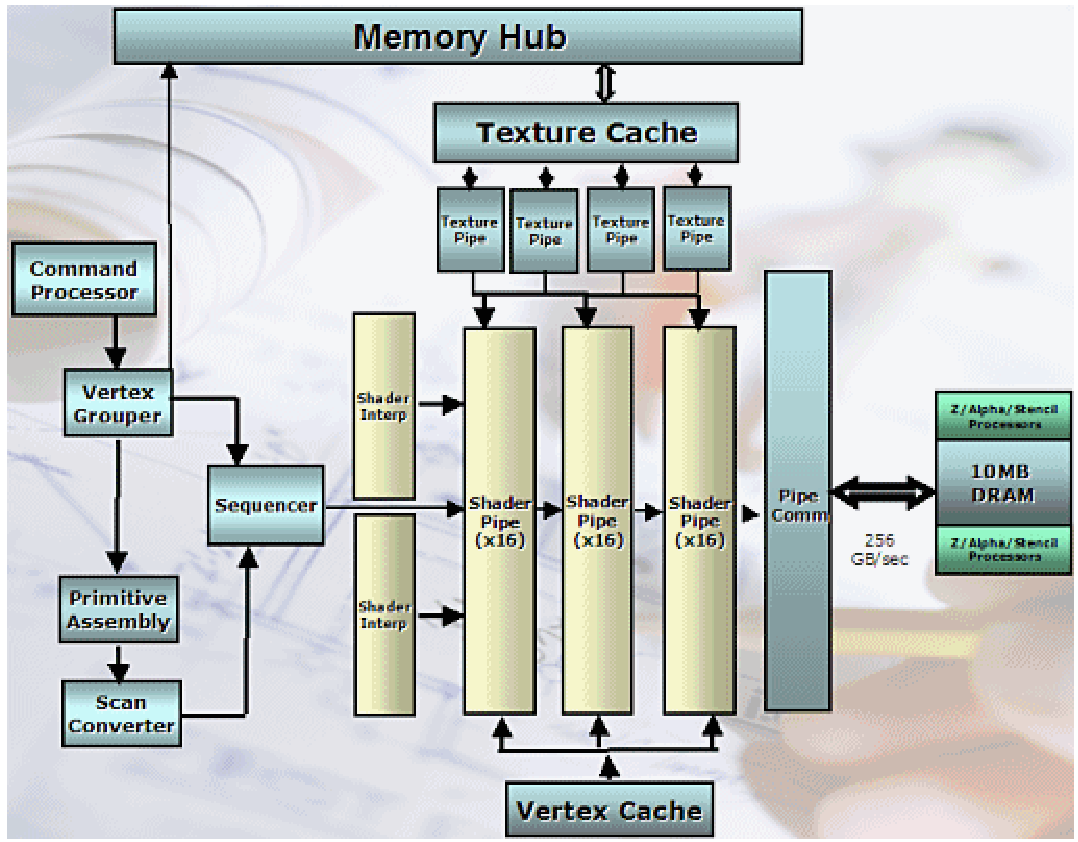
SAMPLE: NVIDIA GT200

The basic components are marked.



SAMPLE: XBOX 360 GPU

Basic Pipeline Diagram



GPU Resources

- Programmable Parallel Processors
  - (480 cores in NVIDIA 8800, and you can link multiple processors to get multiples of that)
- Rasterizer
  - Interpolation happens between vertex->fragment shaders
- Texture Unit
  - Read-only memory
- Render Target
  - Write-only memory

(Adapted From SIGGRAPH 2004 course notes)

Concepts: from CPU to GPU

- Stream/Data Array => Texture
- Shared Memory Read => Texture Samplers
- Loop body / kernel => Fragment Programs
- Writing to array => (Multiple) Render Targets

- If your system is based on fragment shaders:
  - Just draw a full-screen quad, the fragment will be automatically generated by the rasterized and provided to fragment shader.

(Adapted From SIGGRAPH 2004 course notes)

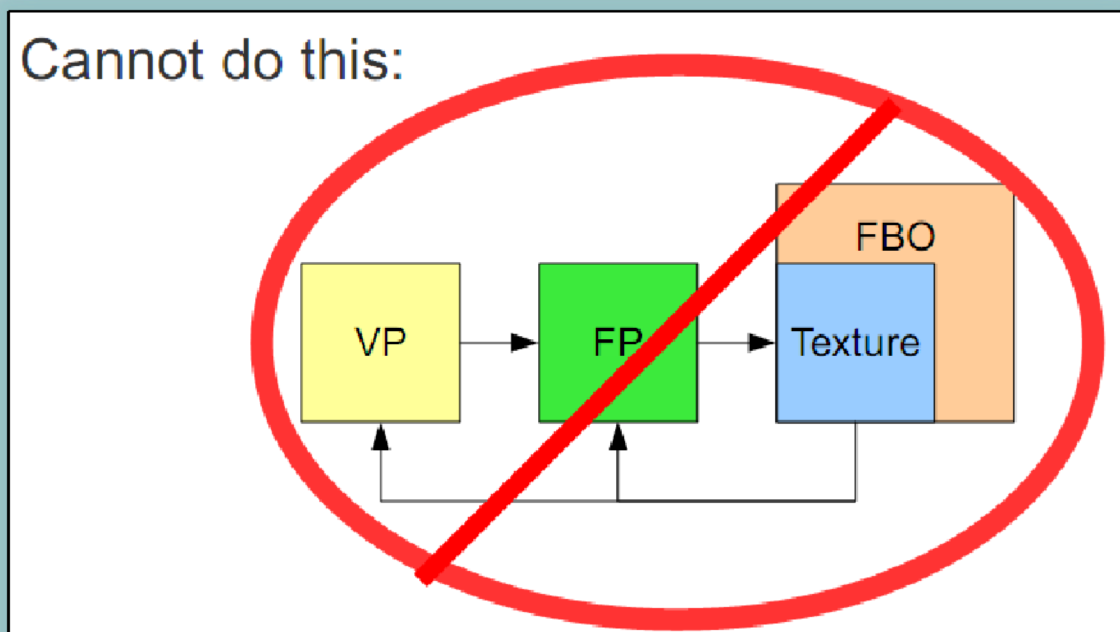
Instruction vs Program

- Generally, GPU's can handle Single Instruction Multiple Data (SIMD).
  - Every data should follow the same path!
  - Branching is expensive!
    - All GPU based API's
      - Restrict number of branching conditions and loops.
      - Prohibit using recursion!
  - Avoid branching when possible.
    - You may need to adjust your data into separate streams.
    - You can map fixed-fucntion functionalities to your problem (if you use OpenGL or DirectX)

(Adapted From SIGGRAPH 2004 course notes)

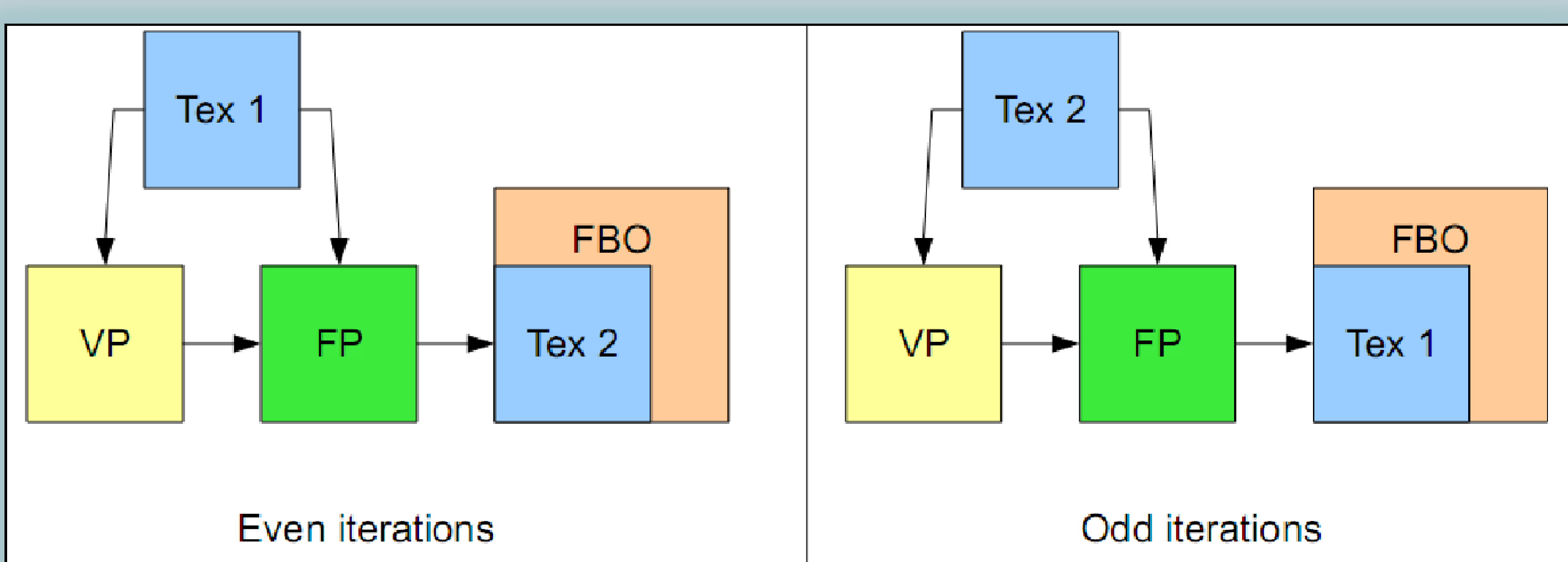
Ping-Pong Buffers

- Problem
  - No simultaneous read-write access to texture memory:



Ping-Pong Buffers

- Solution:
  - Use two textures, one is the source, one is the target.
  - Swap source and target so that the previous result is used as source.



Ping-Pong Buffers

- Drawbacks
  - Double data size required
  - Time overhead for swap
- Three different ways to achieve ping-ponging:
  - Two FBOs, two textures
    - Switch using glBindFramebuffer();
  - One FBO, two textures
    - Switch using glFrameBufferTexture();
  - One FBO, two textures
    - Switch using glDrawBuffer(); // Most optimal for hardware
    - Switch using glDrawBuffers(); // If you need MRT

GPU capabilities – What's done.

- Solving algebraic equations.
  - "Sparse matrix conjugate gradient solvers" and "regular-grid multigrid solvers" already defined in literature.
  - These numerical solutions are already applied to 2D wave equations and incopmressable 2D Navier-Strokes equations.
- Sorting & Searching
  - Your algorithm should be data-parallel
    - (ex: bitonic merge sort[Batcher68], parallel, but not O(nLogN))
  - Kernel : Compare and swap data.
- Database Operations
  - Depth tests : allows comparisons
  - Stencil tests : Data validation / stroing comparison results.
  - Hardware occlusion queries : COUNT() operation. (details left out.)

(Adapted From SIGGRAPH 2004 course notes)

Talking with GPU - Languages

- NVIDIA Sg
- HLSL
- GLSLang
- Sh
  - Univeristy of Waterloo => project has been commercialized
- Brook
  - Stanford University
- Cuda
- Also, (though not designed for GPU's): RenderMan language

NOTE: All are mostly based on C syntax / grammar.

Talking in Assembly Level

- There was assembly level programming in GPU’s first.
  - From Okan Arikan’s Pet-Projects page: “All the texturemapping stuff is implemented in low level assembly. ... I think writing low level assembly for fast texturemapping and math stuff is quite fun... (or at least was fun. It's been 5-6 years since I wrote this one. Since then graphics capabilities on PC's increased so much that everybody takes high quality bi-tri-linear texturemapping for granted).”
- ABS, ADD, CALL, DCL, M3x2, MAX, REP, ...
  - Ex: Optimization using MADR
    - Multipply and add is a single instruction in NVIDIA/ATI GPU’s.

Little stuff do matter!

- In my Voronoi implementation, I observed that I had used an -not-really-necessary branching instruction.
- FPS: Before : ~20
- FPS: After : ~30 (The original work's result in same configuration)
- This was a top-level conditional conditional expression.
  - I removed a low-level conditional expression (selecting metric type using application configuration), improvement was only ~3 FPS.
- Architecture knowledge and some experience is likely to be the key in optimizing shaders (kernels, in GP-GPU jargon).

Little stuff do matter!

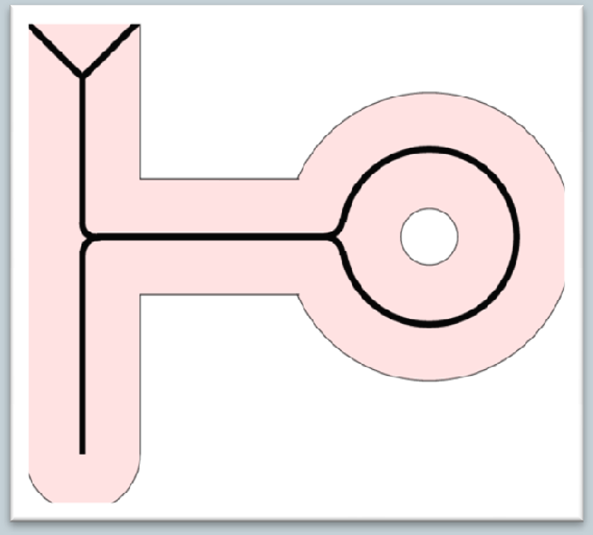
1. Remember that different hardware have different characteristics and performance.
   - Generally, fragment shaders do most of the work in GP-GPU stuff.
   - Yet, some of the work can be shifted to vertex shader and rasterization interpolators.
   - Also, GPU's are similar to CPU's in many ways (ALU logic, registers, caches, etc), so your experience in one of them can be worth in both architectures.
2. Remember that hardware is evolving fast
   - In 2004 SIGGRAPH Course, disadvantages and advantages of DirectX and OpenGL was noted. Today, there is no difference in those respects.

Beginning Voronoi's using Generalizations

- There can be many types of generalizations of Voronoi Diagrams.
  - Higher dimentions (>2D)
  - Different seed types (line, curve, closed surface, or meshes in 3D)
  - Distance metric (Usual function: L1-metric/Euclidian. Others: Manhattan distance, Affine/curved diagrams (Power, Möbius, Apollonius, Anisotropic, etc))
    - Information-theoric diagrams: Ex: Bergman and Csiszar.
      - Are not generally symmetric and not in Euclidian-space.
  - Second (and higher order) diagrams: Each point in space keeps closest 2 (or n) points.

Problems closely related to Voronoi

- Distance Transform
  - “The shortest distance from a point in space to some set of closest point (or, any type of seed) in that space.”
- Voronoi Skeletons
  - Can be derived from distance transform information.
- And Delanuay Tringulation
  - The dual

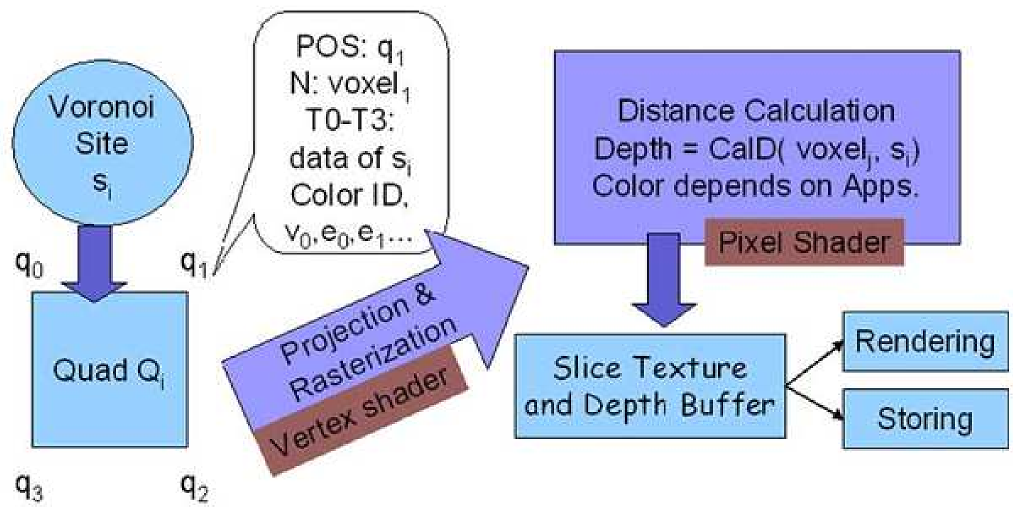


A More Elegant Shader Based Solution

" A simple GPU-based approach for 3D Voronoi diagram construction and visualization "

This work aims 3D Voronoi diagrams, but works in 2D slices.

- Distance computation: Render a full-quad geom holding related seed data.
- As distance data is generated, the diagram is updated.
- Separates visualization problem (same idea used in my project.)



Jump Flooding

- See video for method description…
- My implementation is based on this work and its variants.
- A VoronoiSolver class, vertex/fragment solver shaders and visualization shaders implemented.
- Multiple solvers can exist and be computed by GPU in a single application.

Jump Flooding

- Implementation: Data is distributed over many texture units.
  - Two textures store the closest Voronoi seed information. (Ping-pong buffers)
  - One texture stores the seed positions (indexed through seedID's)
  - One texture stores the seed colors positions (indexed through seedID's)
- Only "closest seed" data is stored in Voronoi solution, the distance to seed is re-calculated by the shaders when required using

Jump Flooding

- Constructor shader:
  - For each fragment(texel) : find the closest Voronoi seed which is in one stepSize length away neighbourhood.
  - 8-neighborhood is used for this purpose.
- Visualization shader:
  - Using seed information, can generate distance transformations and mark Voronoi edges / Soronoi seeds.
    - Only this shader has seed color input. (Colors were for visualization purposes.)

Delanuay: GPU competes CPU

- "Computing two-dimensional Delaunay triangulation using graphics hardware“
  - One of the works of the derivate of implemented work
    - Published in I3D 2008 – GPGPU Session
- Can produce exact results in continous domain.
  - Recent implementation available on web does nearly all computation phases in GPU. (But requires at least NVIDIA G80 cores)
- Result: Up to %53 improvements over best CPU implementation (Triangle) when number of sites ~1 million.
  - Remember “large batches of data” / data parallelism .

The Future

- Rendering is not a solved problem!
  - Still, many people working in technology-leading teams and developing the methods that will be the "new" standarts.
- GP-GPU is a fruitful area both for academic and industrial research.
  - Even standart graphics API's can do more than you imagine at times, not to mention new API's.
- **Users** : GPU's are in every desktop computers (end even mobile phones). They want to be amazed.
  - Develop on-the-edge high-performance applications using GPU power

The Future

- "Even though modeling and rendering in CG have been improved tremendously in past 35 years, we are still not at the point where we can model a tiger swimming in the river in all its glorius details. By automatically, I mean in a way that does not need careful meanual tweaking by an artist/expert.

  The bad news is that we have a still long way to go.

  The good news is that we have a still long way to go.“

Alain Fournier, Cornell Workshop, 1998

The Future

- Real-time rendering approaches pre-rendered CG.
  - Current rendering pipeline continues to develop fast.
- Can global-illumination models become real-time?
  - Everyone in industry has his/her own idea. Who can see the future?

<u>A trivia</u>

- One of the first studies in Computer Graphics:
  - How to decide visibility based on object depth?
- The practical answer that has been used for decades:
  - Z-buffer!
    - And many concepts followed: such as early z-culling.
  - Yet, Z-Buffer was mentioned to be "impractical" in that first study :)

Questions for Us

- Where does Bilkent stand in this massive architecture?
  - No course introduces recent Realtime/Interactive Rendering concepts.
  - No course introduces the GPU hardware, parallelism and the related computational models.
  - Proposed course project topics fail to provide the vision required.
    - Fact: ~7 graduate courses in Computer Graphics related fields.
- In Turkey: No SIGGRPAH paper or industry driven heavy graphics projects.

Thanks…

- Questions are welcome !