Advanced Programming (GPGPU)

Mike Houston

The world changed over the last year...

- **Multiple GPGPU initiatives**
  - Vendors without GPGPU talking about it
- **A few big apps:**
  - Game physics
  - Folding@Home
  - Video processing
  - Finance modeling
  - Biomedical
  - Real-time image processing
- **Courses**
  - UIUC – ECE 498
  - Supercomputing 2006
  - SIGGRAPH 2006/2007
- **Lots of academic research**
- **Actual GPGPU companies**
  - PeakStream
  - RapidMind
  - Accelware
  - ...

What can you do on GPUs other than graphics?

- Large matrix/vector operations (BLAS)
- Protein Folding (Molecular Dynamics)
- FFT (SETI, signal processing)
- Ray Tracing
- Physics Simulation [cloth, fluid, collision]
- Sequence Matching (Hidden Markov Models)
- Speech Recognition (Hidden Markov Models, Neural nets)
- Databases
- Sort/Search
- Medical Imaging (image segmentation, processing)
- And many, many more…

GPGPU

http://www.gpgpu.org

Task vs. Data parallelism

- **Task parallel**
  - Independent processes with little communication
  - Easy to use
    - “Free” on modern operating systems with SMP
- **Data parallel**
  - Lots of data on which the same computation is being executed
  - No dependencies between data elements in each step in the computation
  - Can saturate many ALUs
  - But often requires redesign of traditional algorithms

CPU vs. GPU

- **CPU**
  - Really fast caches (great for data reuse)
  - Fine branching granularity
  - Lots of different processes/threads
  - High performance on a single thread of execution
- **GPU**
  - Lots of math units
  - Fast access to onboard memory
  - Run a program on each fragment/vertex
  - High throughput on parallel tasks

- **CPUs are great for *task* parallelism**
- **GPUs are great for *data* parallelism**

The Importance of Data Parallelism for GPUs

- **GPUs are designed for highly parallel tasks like rendering**
- **GPUs process *independent* vertices and fragments**
  - Temporary registers are zeroed
  - No shared or static data
  - No read-modify-write buffers
  - In short, no communication between vertices or fragments
- **Data-parallel processing**
  - GPU architectures are ALU-heavy
    - Multiple vertex & pixel pipelines
    - Lots of compute power
  - GPU memory systems are designed to *stream* data
    - Linear access patterns can be prefetched
    - Hide memory latency

GPGPU Terminology

Arithmetic Intensity

- **Arithmetic intensity**
  - Math operations per word transferred
  - Computation / bandwidth
- **Ideal apps to target GPGPU have:**
  - Large data sets
  - High parallelism
  - Minimal dependencies between data elements
  - High arithmetic intensity
  - Lots of work to do without CPU intervention

Data Streams & Kernels

- **Streams**
  - Collection of records requiring similar computation
    - Vertex positions, Voxels, FEM cells, etc.
  - Provide data parallelism
- **Kernels**
  - Functions applied to each element in stream
    - transforms, PDE, …
  - No dependencies between stream elements
    - Encourage high Arithmetic Intensity

Scatter vs. Gather

- **Gather**
  - Indirect read from memory (x = a[i])
  - Naturally maps to a texture fetch
  - Used to access data structures and data streams
- **Scatter**
  - Indirect write to memory (a[i] = x)
  - Difficult to emulate:
    - Render to vertex array
    - Sorting buffer
  - Needed for building many data structures
  - Usually done on the CPU

Mapping algorithms to the GPU

Mapping CPU algorithms to the GPU

- **Basics**
  - Stream/Arrays -> Textures
  - Parallel loops -> Quads
  - Loop body -> vertex + fragment program
  - Output arrays -> render targets
  - Memory read -> texture fetch
  - Memory write -> framebuffer write
- **Controlling the parallel loop**
  - Rasterization = Kernel Invocation
  - Texture Coordinates = Computational Domain
  - Vertex Coordinates = Computational Range

Computational Resources

- **Programmable parallel processors**
  - Vertex & Fragment pipelines
- **Rasterizer**
  - Mostly useful for interpolating values (texture coordinates) and per-vertex constants
- **Texture unit**
  - Read-only memory interface
- **Render to texture**
  - Write-only memory interface

Vertex Processors

- **Fully programmable (SIMD / MIMD)**
- **Processes 4-vectors (RGBA / XYZW)**
- **Capable of scatter but not gather**
  - Can change the location of current vertex
  - Cannot read info from other vertices
  - Can only read a small constant memory
- **Vertex Texture Fetch**
  - Random access memory for vertices
  - Limited gather capabilities
    - Can fetch from texture
    - Cannot fetch from current vertex stream

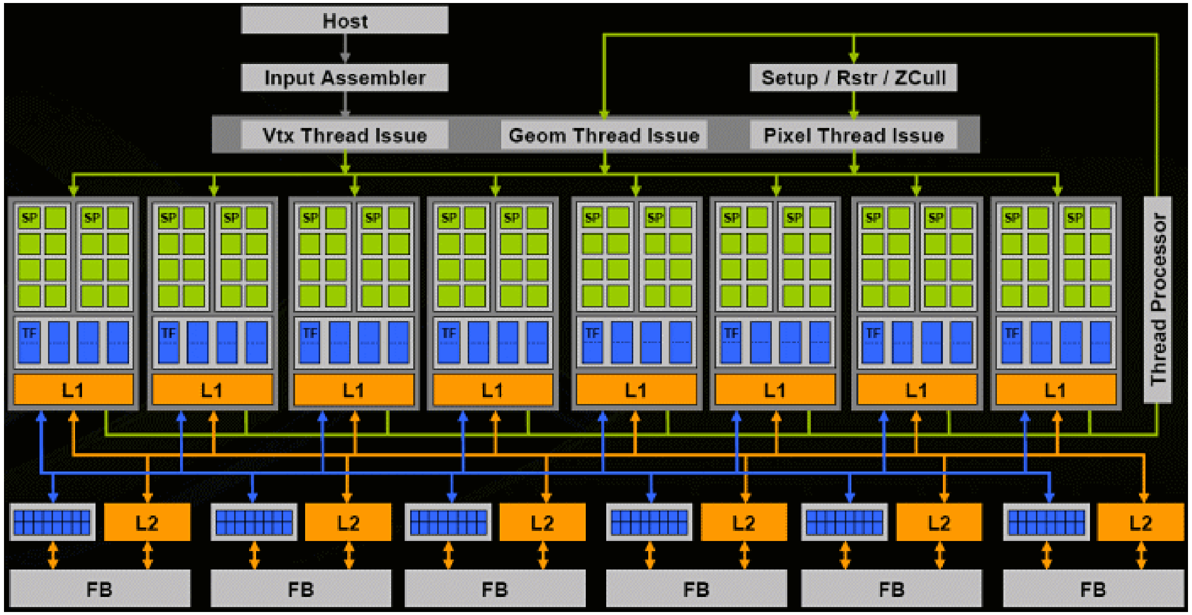
Fragment Processors

- **Fully programmable (SIMD)**
- **Processes 4-component vectors (RGBA / XYZW)**
- **Random access memory read (textures)**
- **Generally capable of gather but not scatter**
  - Indirect memory read (texture fetch), but no indirect memory write
  - Output address fixed to a specific pixel
- **Typically more useful than vertex processor**
  - More fragment pipelines than vertex pipelines
  - Direct output (fragment processor is at end of pipeline)
  - Better memory read performance

- **For GPGPU, we mainly concentrate on using the fragment processors**
  - Most of the flops
  - Highest memory bandwidth

And then they were unified…

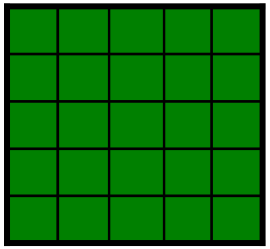
- **Current trend is to unify shading resources**
  - DX10 – vertex/geometry/fragment shading have similar capabilities
  - Just a "pool of processors"
    - Scheduled by the hardware dynamically
    - You can get "all" the board resources through each



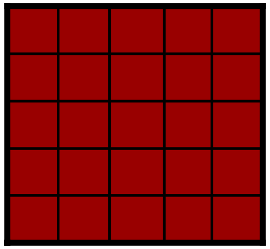
NVIDIA 8800GTX

GPGPU example – Adding Vectors

- **Place arrays into 2D textures**
- **Convert loop body into a shader**
- **Loop body = Render a quad**
  - Needs to cover all the pixels in the output
  - 1:1 mapping between pixels and texels
- **Readback framebuffer into result array**



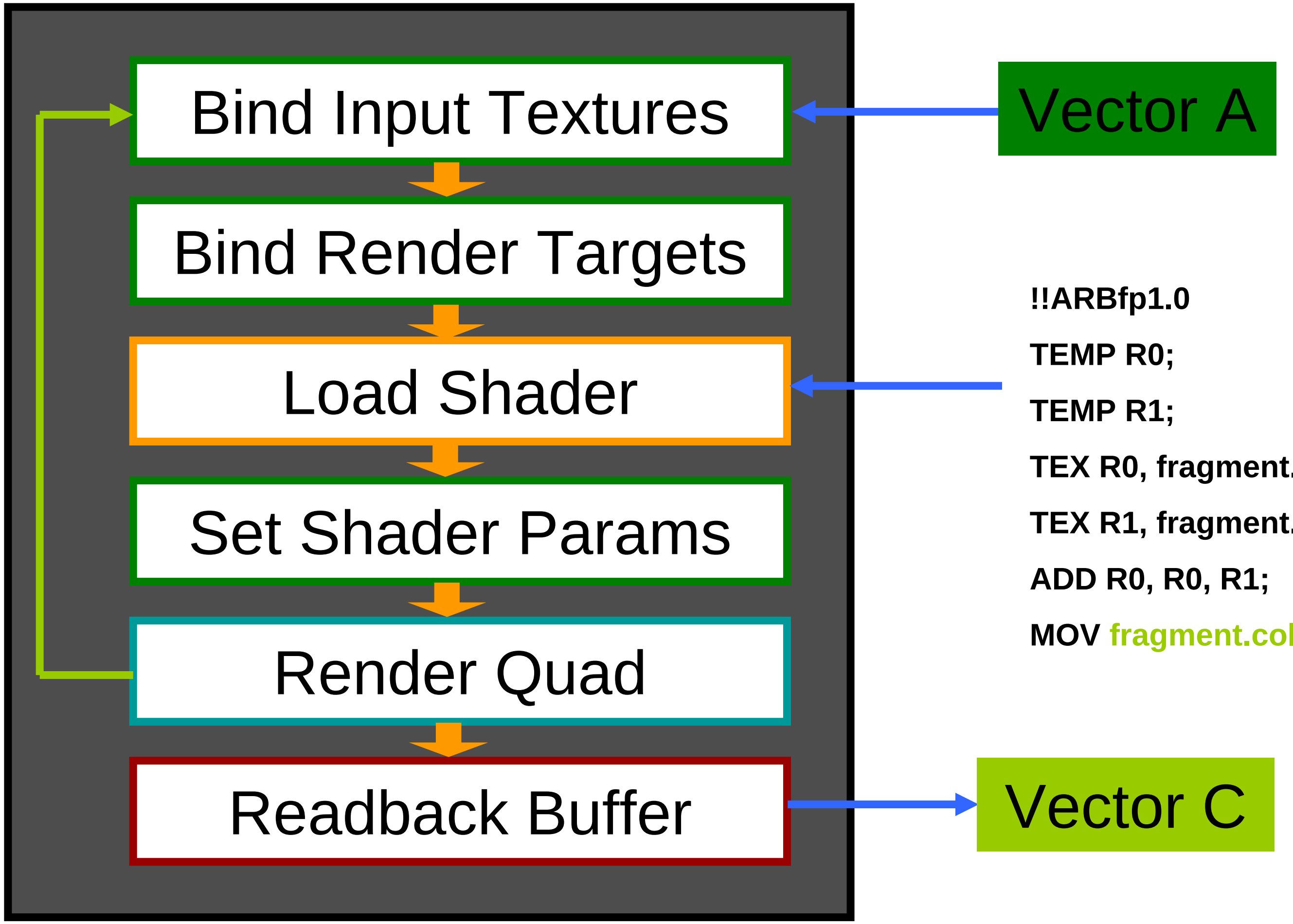
```
float a[5*5];
float b[5*5];
float c[5*5];
//initialize vector a
//initialize vector b
for(int i=0; i<5*5; i++)
{
    c[i] = a[i] + b[i];
}
```



0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

```
!!ARBfp1.0
TEMP R0;
TEMP R1;
TEX R0, fragment.position, texture[0], 2D;
TEX R1, fragment.position, texture[1], 2D;
ADD R0, R0, R1;
MOV fragment.color, R0;
```

How this basically works - Adding vectors




```
!!ARBfp1.0
TEMP R0;
TEMP R1;
TEX R0, fragment.position, texture[0], 2D;
TEX R1, fragment.position, texture[1], 2D;
ADD R0, R0, R1;
MOV fragment.color, R0;
```

Rolling your own GPGPU apps

- **Lots of information on GPGPU.org**
- **For those with a strong graphics background:**
  - Do all the graphics setup yourself
  - Write your kernels:
    - Use high level languages
      - Cg, HLSL, ASHLI
    - Or, direct assembly
      - ARB_fragment_program, ps20, ps2a, ps2b, ps30
- **High level languages and systems to make GPGPU easier**
  - BrookGPU (http://graphics.stanford.edu/projects/brookgpu/)
  - RapidMind (http://www.rapidmind.net)
  - PeakStream (http://www.peakstreaminc.com)
  - CUDA – NVIDIA (http://developer.nvidia.com/cuda)
  - CTM – AMD/ATI (ati.amd.com/companyinfo/researcher/documents.html)

Basic operations

- **Map**
- **Reduce**
- **Scan**
- **Gather/Scatter**
  - Covered earlier

Map operation

- **Given:**
  - Array or stream of data elements A
  - Function $f(x)$
- **map(A, f) = applies f(x) to all $a_i \in A$**
- **GPU implementation is straightforward**
  - A is a texture, a_i are texels
  - Pixel shader implements $f(x)$, reads a_i as x
  - Draw a quad with as many pixels as texels in A with $f(x)$ pixel shader active
  - Output(s) stored in another texture

Courtesy John Owens

Parallel Reductions

- **Given:**
  - Binary associative operator $\oplus$ *with identity I*
  - Ordered set s = [a_0, a_1, …, a_{n-1}] of n elements
- **Reduce($\oplus$, s) returns $\mathbf{a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}}$**
- **Example:**
  - Reduce(+, [3 1 7 0 4 1 6 3]) = 25

- **Reductions common in parallel algorithms**
  - Common reduction operators are +, x, min, max
  - Note floating point is only pseudo-associative

Courtesy John Owens

Parallel Scan (aka prefix sum)

- **Given:**
  - Binary associative operator $\oplus$ with identity I
  - Ordered set s = [a0, a1, …, an-1] of n elements
- **scan($\oplus$, s) returns**

  $$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

- **Example:**

  scan(+, [3 1 7 0 4 1 6 3]) = [3 4 11 11 15 16 22 25]

(From Blelloch, 1990, "Prefix Sums and Their Applications")

Courtesy John Owens

Applications of Scan

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction
- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms

Courtesy John Owens

Brook: General Purpose Streaming Language

- **Stream programming model**
  - GPU = streaming coprocessor
- **C with stream extensions**
- **Cross platform**
  - ATI & NVIDIA
  - OpenGL, DirectX, CTM
  - Windows & Linux

Streams

- **Collection of records requiring similar computation**
  - particle positions, voxels, FEM cell, ...

```
Ray r<200>;
float3 velocityfield<100,100,100>;
```

- **Similar to arrays, but...**
  - index operations disallowed: **`position[i]`**
  - read/write stream operators

```
streamRead (r, r_ptr);
streamWrite (velocityfield, v_ptr);
```

Kernels

- **Functions applied to streams**
  - similar to for_all construct
  - no dependencies between stream elements

```
kernel void foo (float a<>, float b<>,
                 out float result<>) {
  result = a + b;
}

float a<100>;
float b<100>;
float c<100>;

foo(a,b,c);
```

```
for (i=0; i<100; i++)
      c[i] = a[i]+b[i];
```

Kernels

- **Kernel arguments**
  - input/output streams

```
kernel void foo (float a<>,
                 float b<>,
                 out float result<>) {
   result = a + b;
}
```

Kernels

- **Kernel arguments**
  - input/output streams
  - gather streams

```
kernel void foo (..., float array[] ) {
      a = array[i];
}
```

Kernels

- **Kernel arguments**
  - input/output streams
  - gather streams
  - iterator streams

```
kernel void foo (..., iter float n<> ) {
      a = n + b;
}
```

Kernels

- **Kernel arguments**
  - input/output streams
  - gather streams
  - iterator streams
  - constant parameters

```
kernel void foo (..., float c ) {
      a = c + b;
}
```

Reductions

- **Compute single value from a stream**
  - associative operations only

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}
```

```
float a<100>;
float r;

sum(a,r);
```

```
r = a[0];
for (int i=1; i<100; i++)
  r += a[i];
```

Reductions

- **Multi-dimension reductions**
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
                 reduce float r<>)
  r += a;
}
```

```
float a<20>;
float r<5>;

sum(a,r);
```

```
for (int i=0; i<5; i++)
  r[i] = a[i*4];
  for (int j=1; j<4; j++)
    r[i] += a[i*4 + j];
```

Stream Repeat & Stride

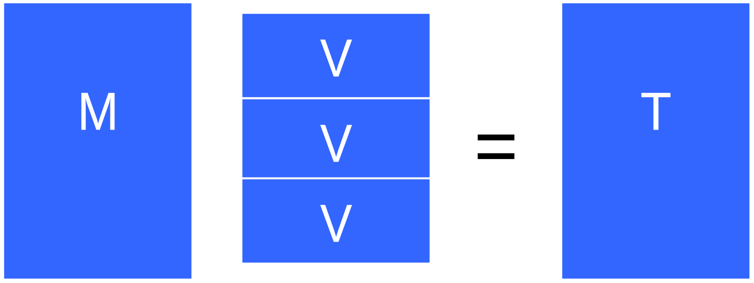
- **Kernel arguments of different shape**
  - resolved by repeat and stride

```
kernel void foo (float a<>, float b<>,
                 out float result<>);

float a<20>;
float b<5>;
float c<10>;

foo(a,b,c);
```

```
foo(a[0],  b[0], c[0])
foo(a[2],  b[0], c[1])
foo(a[4],  b[1], c[2])
foo(a[6],  b[1], c[3])
foo(a[8],  b[2], c[4])
foo(a[10], b[2], c[5])
foo(a[12], b[3], c[6])
foo(a[14], b[3], c[7])
foo(a[16], b[4], c[8])
foo(a[18], b[4], c[9])
```

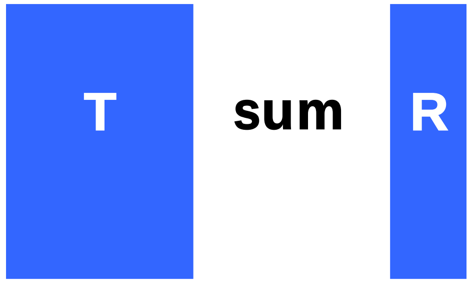
Matrix Vector Multiply

```
kernel void mul (float a<>, float b<>,
                  out float result<>) {
  result = a*b;
}

reduce void sum (float a<>,
                  reduce float result<>) {
  result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```



Matrix Vector Multiply

```
kernel void mul (float a<>, float b<>,
                  out float result<>) {
   result = a*b;
}

reduce void sum (float a<>,
                  reduce float result<>) {
   result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```

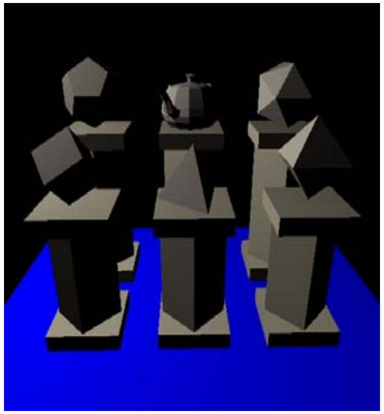


Runtime

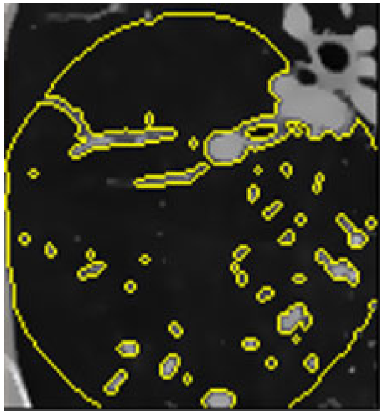
- **Accessing stream data for graphics aps**
  - Brook runtime api available in C++ code
  - autogenerated .hpp files for brook code

```
brook::initialize( "dx9", (void*)device );

// Create streams
fluidStream0 = stream::create<float4>( kFluidSize, kFluidSize );
normalStream = stream::create<float3>( kFluidSize, kFluidSize );

// Get a handle to the texture being used by
// the normal stream as a backing store
normalTexture = (IDirect3DTexture9*)
                 normalStream->getIndexedFieldRenderData(0);

// Call the simulation kernel
simulationKernel( fluidStream0, fluidStream0, controlConstant,
                  fluidStream1 );
```

Applications

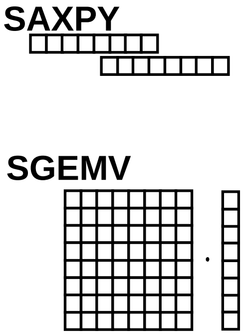


ray-tracer



segmentation

fft edge detect



linear algebra

Folding@home distributed computing

Brook for GPUs

- **Release v0.4 available on Sourceforge**
  - CVS tree *much* more up to date and includes CTM support
- **Project Page**
  - http://graphics.stanford.edu/projects/brook
- **Source**
  - http://www.sourceforge.net/projects/brook
- **Paper:**

  Brook for GPUs: Stream Computing on Graphics Hardware

  Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan

Understanding GPUs Through Benchmarking

Introduction

- **Key areas for GPGPU**
  - Memory latency behavior
  - Memory bandwidths
  - Upload/Download
  - Instruction rates
  - Branching performance
- **Chips analyzed**
  - ATI X1900XTX (R580)
  - NVIDIA 7900GTX (G71)
  - NVIDIA 8800GTX (G80)

GPUBench

- **An open-source suite of micro-benchmarks**
  - GL (we'll be using this for the talk)
  - DX9 (alpha version)
- **Developed at Stanford to aid our understanding of GPUs**
  - Vendors wouldn't directly tell us arch details
  - Behavior under GPGPU apps different than games and other benchmarks
- **Library of results**

http://graphics.stanford.edu/projects/gpubench/

Memory latency

- **Questions**
  - Can latency be hidden?
  - Does access pattern affect latency?

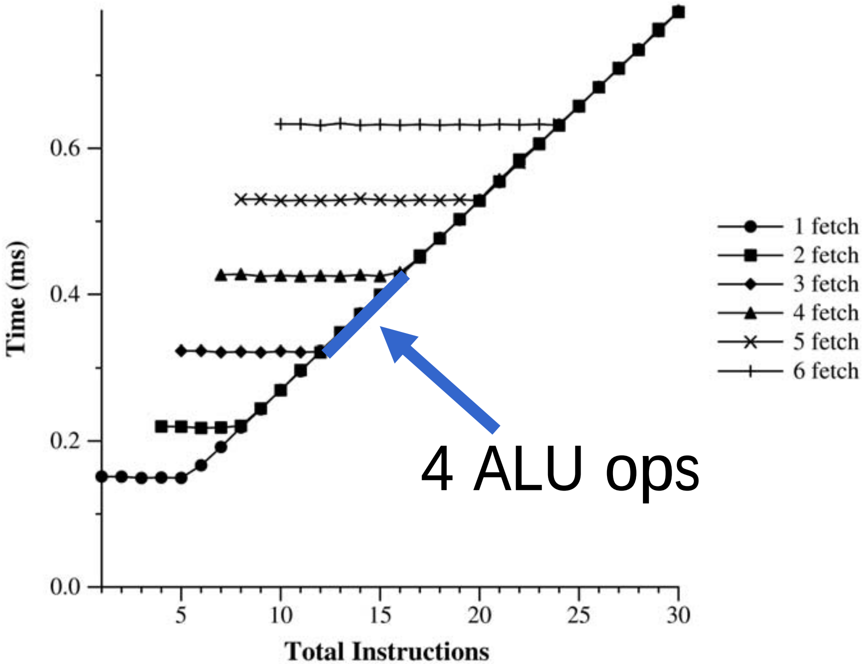
Methodology

- **Try different numbers of texture fetches**
  - Different access patterns:
    - Cache hit – every fetch to the same texel
    - Sequential – every fetch increments address by 1
    - Random – dependent lookup with random texture
- **Increase the ALU ops of the shader**
- **ALU ops *must* be dependent to avoid optimization**

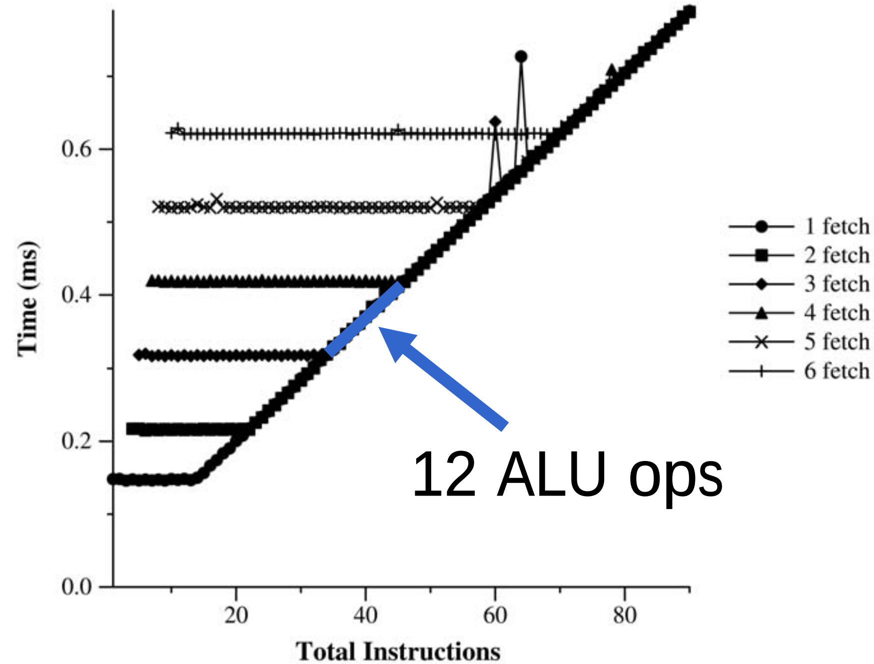
- **GPUBench test: fetchcost**

Fetch cost – ATI – cache hit

X1900XTX has 3X the ALUs per pipe



ATI X1800XT

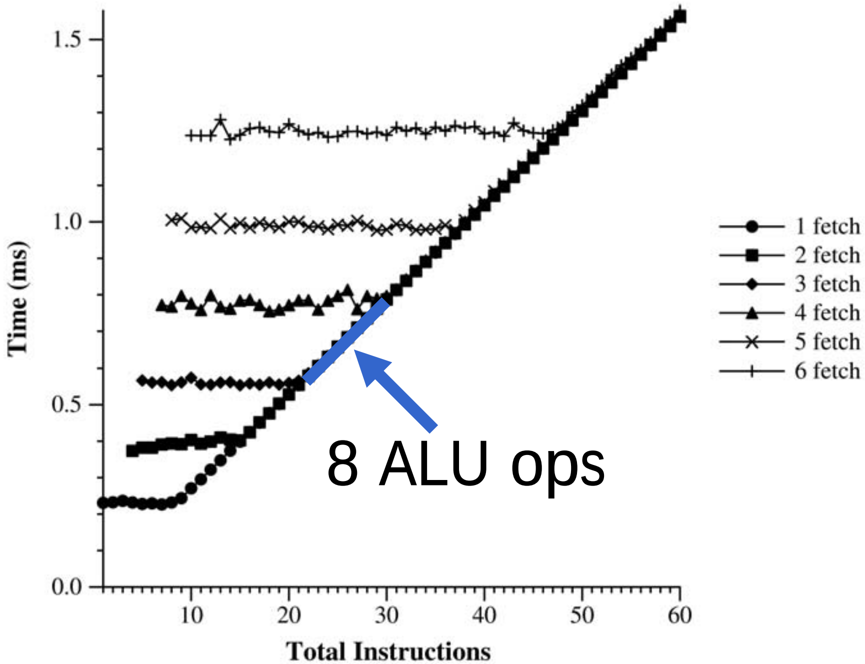


ATI X1900XTX

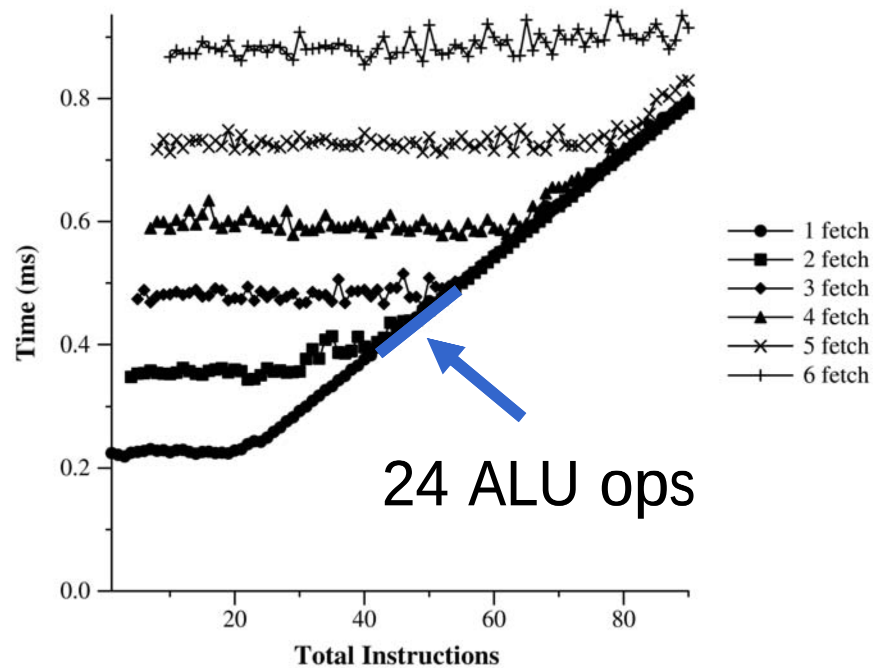
Cost = max(ALU, TEX)

Fetch cost – ATI – sequential

X1900XTX has 3X the ALUs per pipe



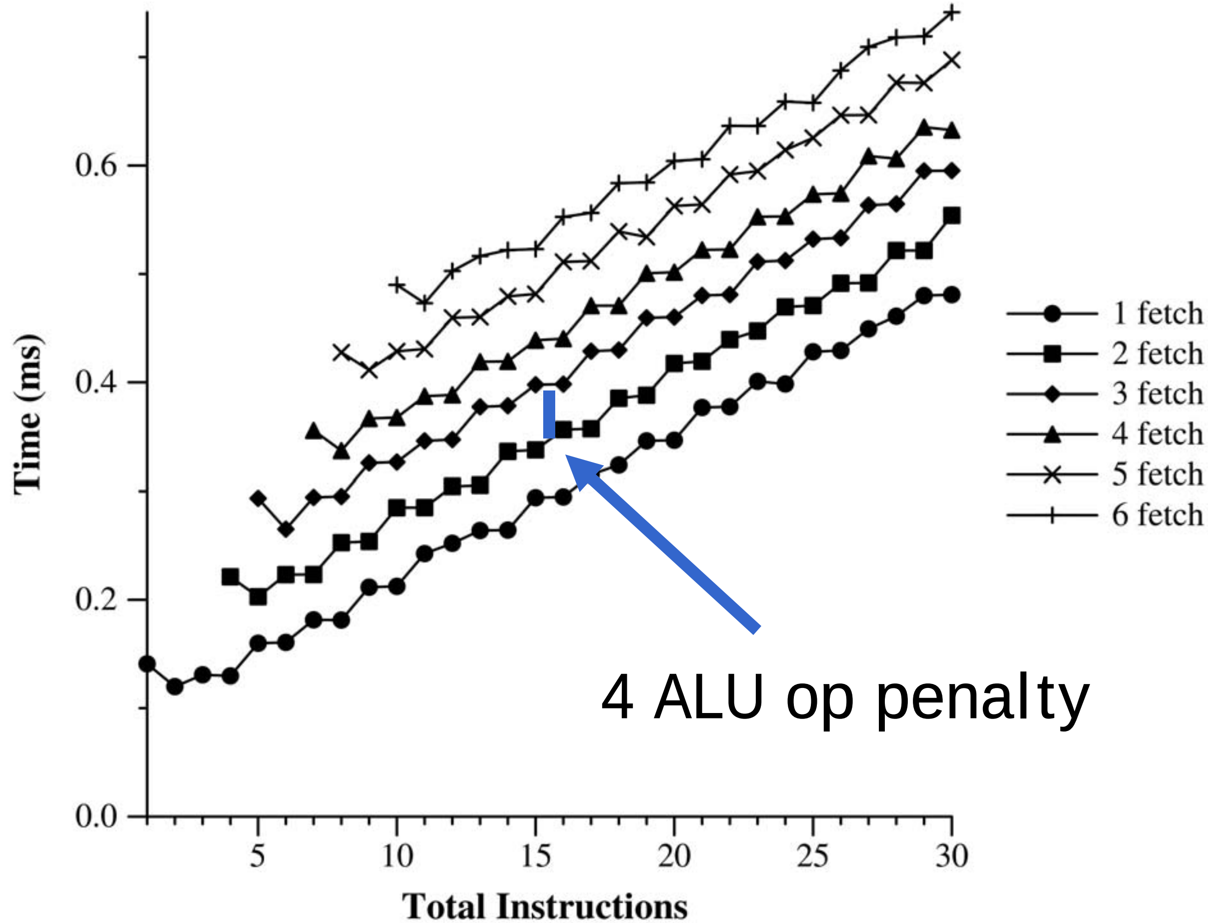
ATI X1800XT



ATI X1900XTX

Cost = max(ALU, TEX)

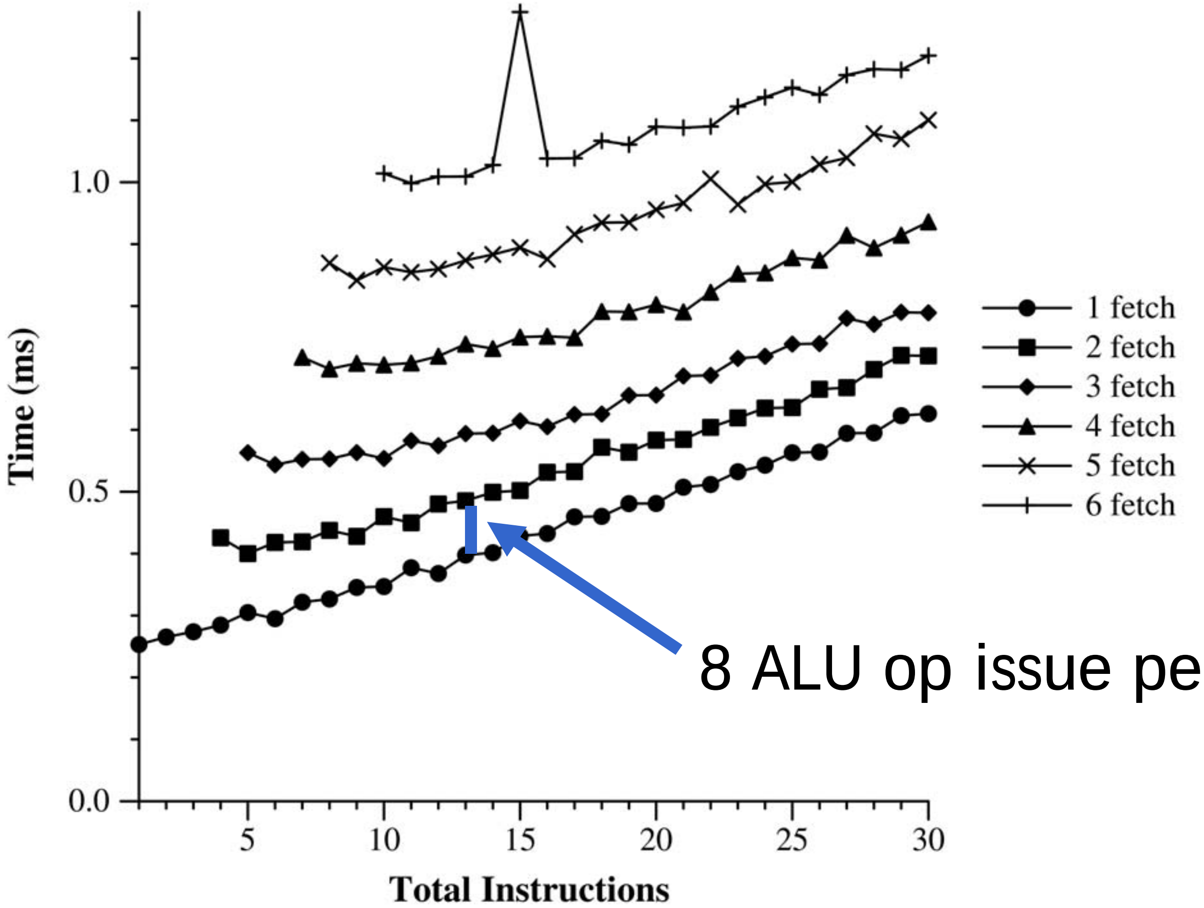
Fetch cost – NVIDIA – cache hit



Cost = sum(ALU, TEX)

NVIDIA 7900 GTX

Fetch cost – NVIDIA – sequential



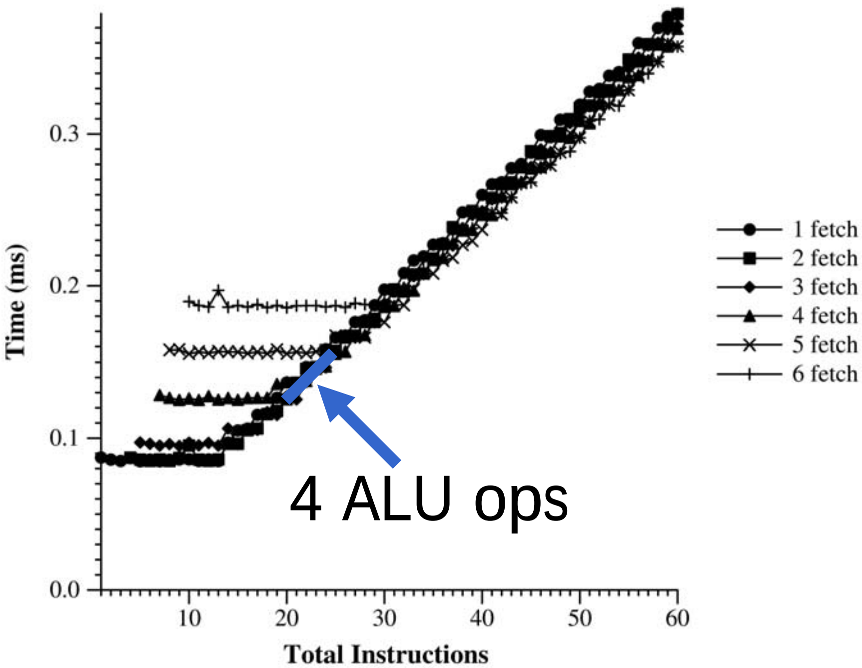
8 ALU op issue penalty

Cost = sum(ALU, TEX)

NVIDIA 7900 GTX

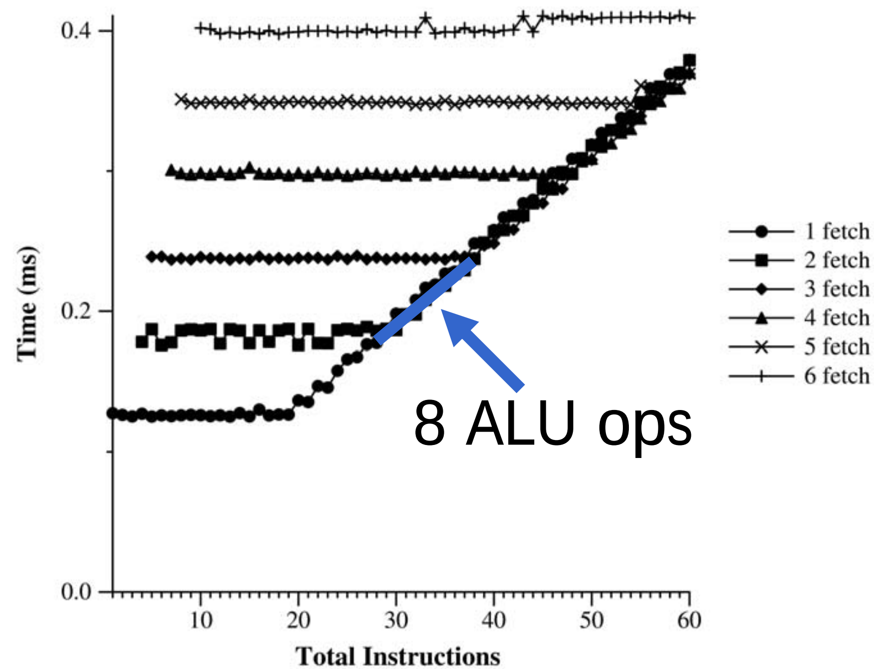
Fetch cost – NVIDIA 8800 GTX

Cache



NVIDIA 8800 GTX

sequential



NVIDIA 8800 GTX

Cost = max(ALU, TEX)

Bandwidth to ALUs

- **Questions**
  - Cache performance?
  - Sequential performance?
  - Random-read performance?

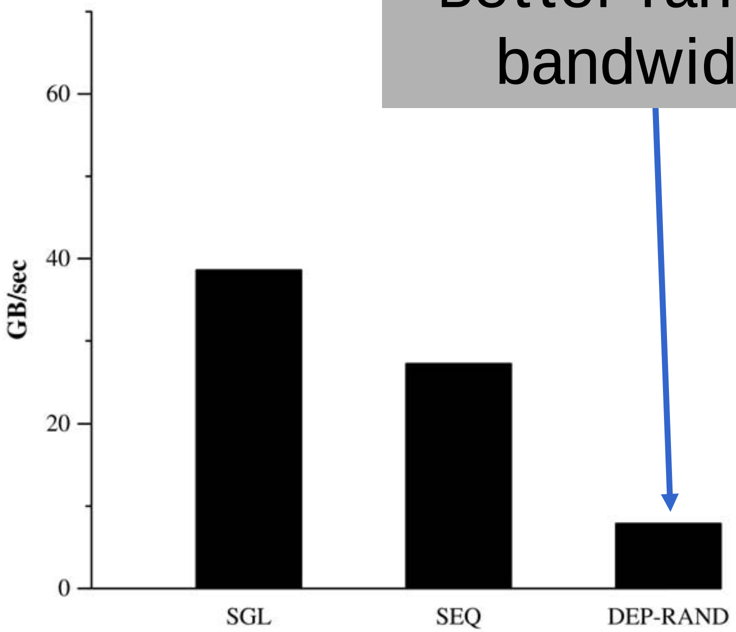
Methodology

- **Cache hit**
  - Use a constant as index to texture(s)
- **Sequential**
  - Use fragment position to index texture(s)
- **Random**
  - Index a seeded texture with fragment position to look up into input texture(s)

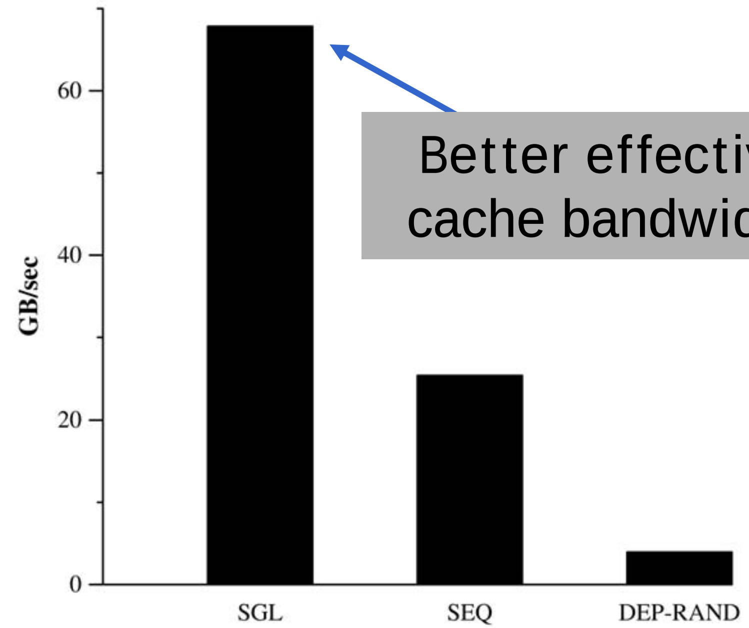
- **GPUBench test: inputfloatbandwidth**

Results

Better random bandwidth



ATI X1900XTX

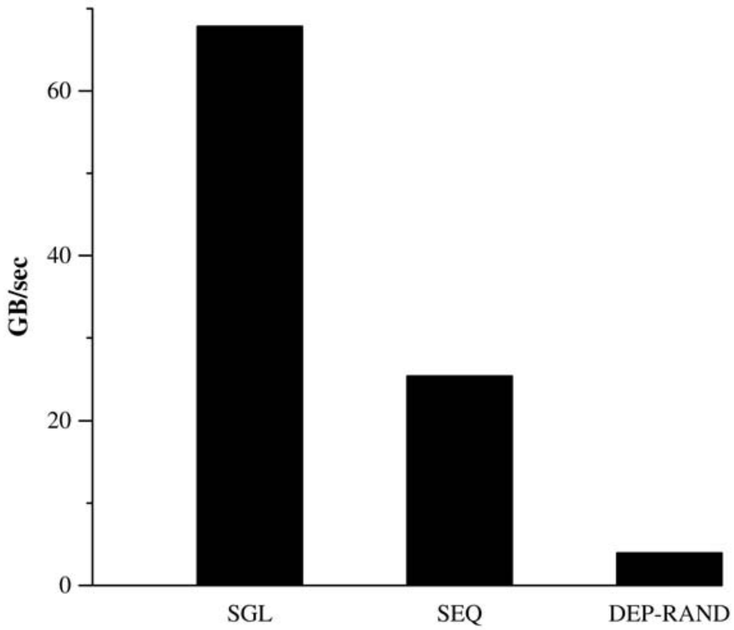


Better effective cache bandwidth

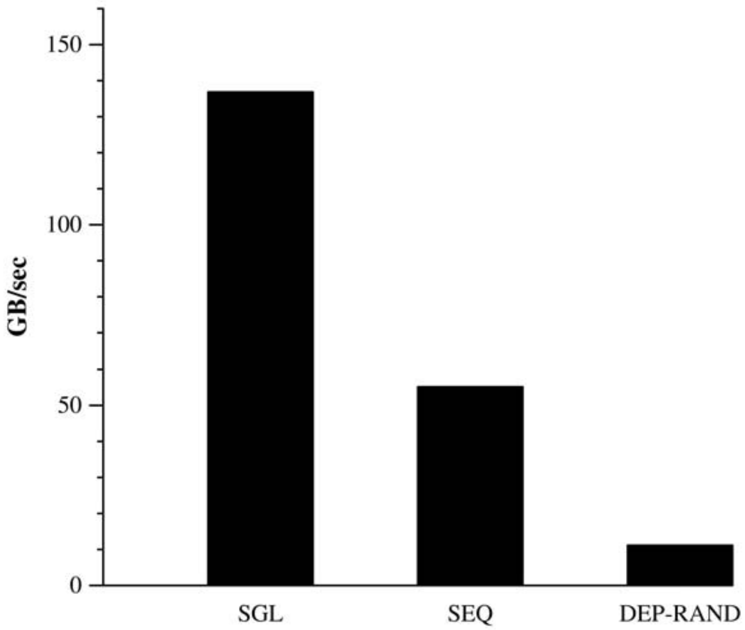
NVIDIA 7900GTX

Sequential bandwidth (SEQ) about the same

Results



NVIDIA 7900GTX



NVIDIA 8800GTX

2X bandwidth of 7900GTX

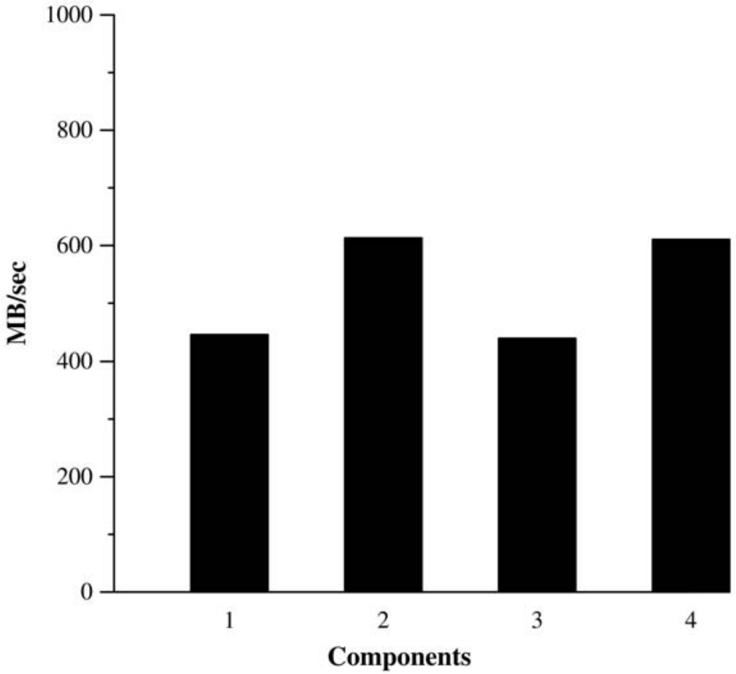
Off-board bandwidth

- **Questions**
  - How fast can we get data on the board (download)?
  - How fast can we get data off the board (readback)?

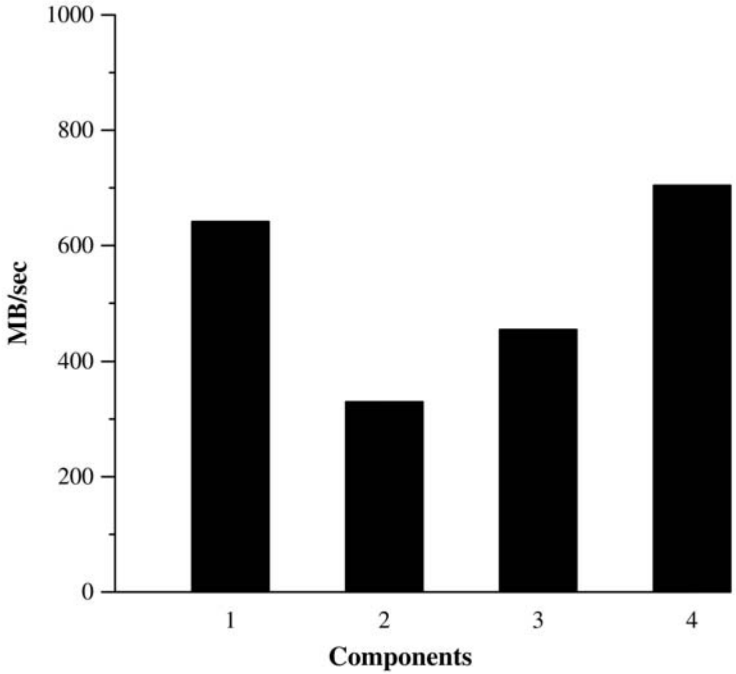
- **GPUBench tests:**
  - download
  - readback

Download

Host to GPU is slow



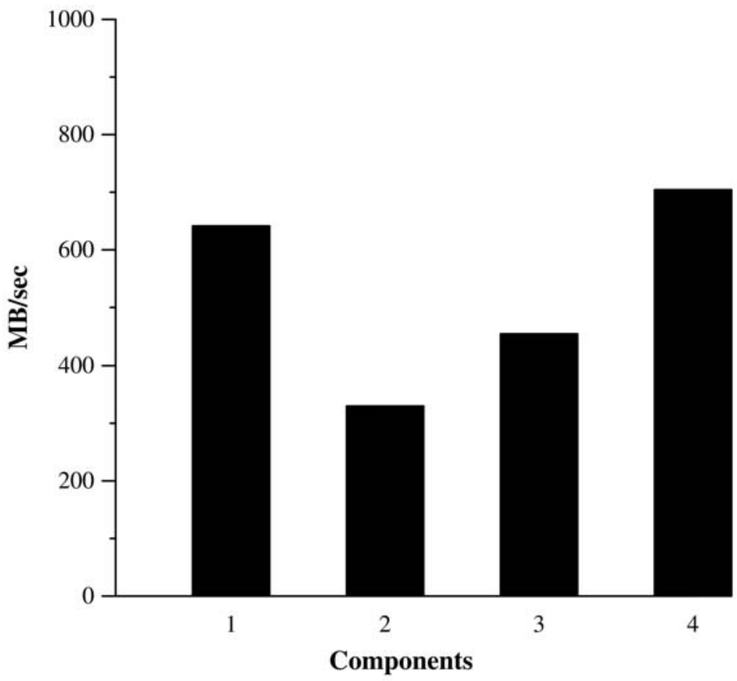
ATI X1900XTX



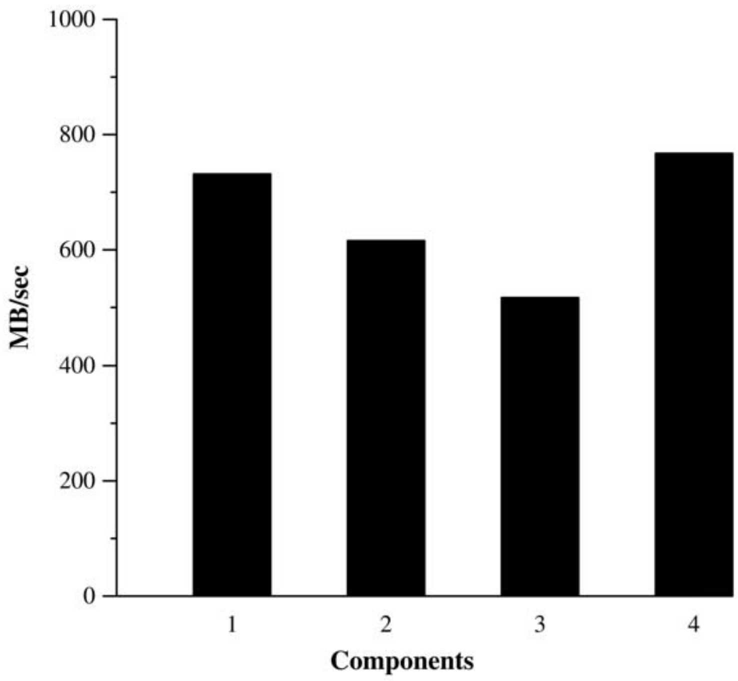
NVIDIA 7900GTX

Download

Next generation not much better...



NVIDIA 7900GTX

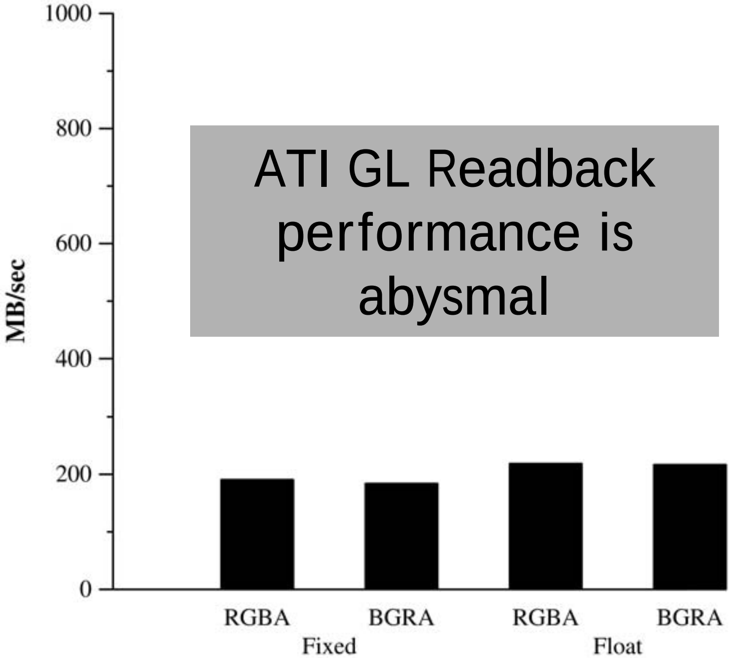


NVIDIA 8800GTX

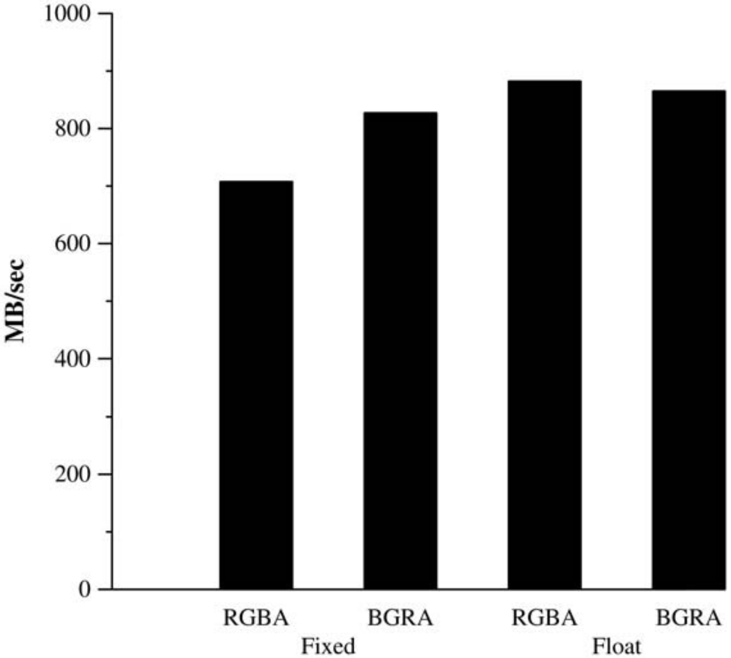
Readback

GPU to host is slow

ATI GL Readback performance is abysmal



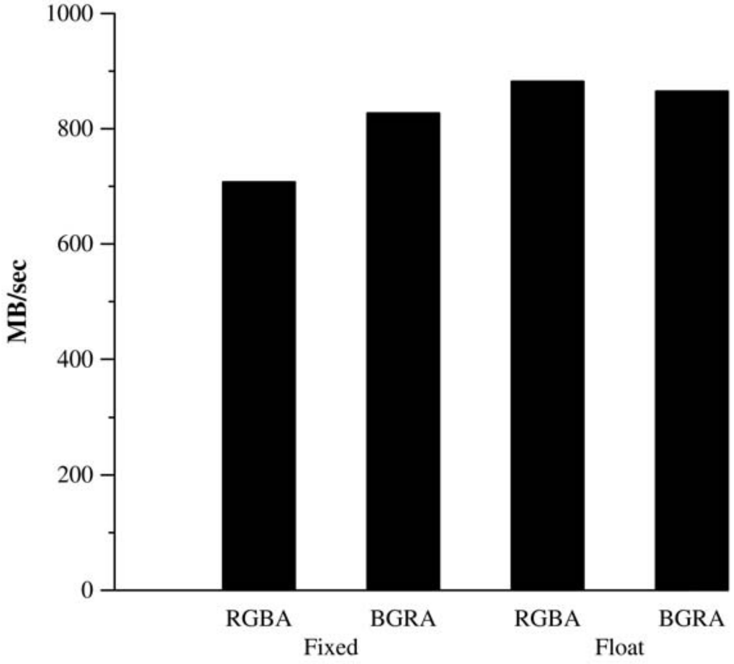
ATI X1900XTX



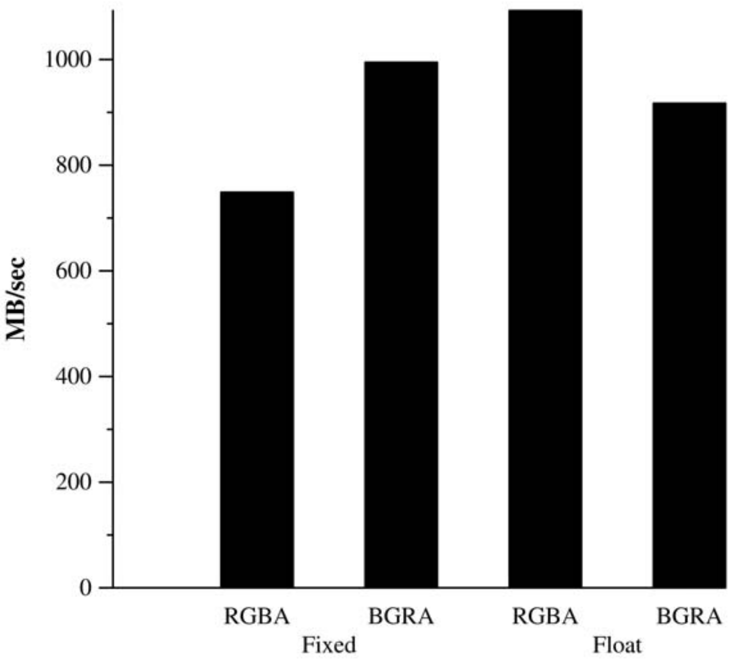
NVIDIA 7900GTX

Readback

Next generation not much better…



NVIDIA 7900GTX



NVIDIA 8800GTX

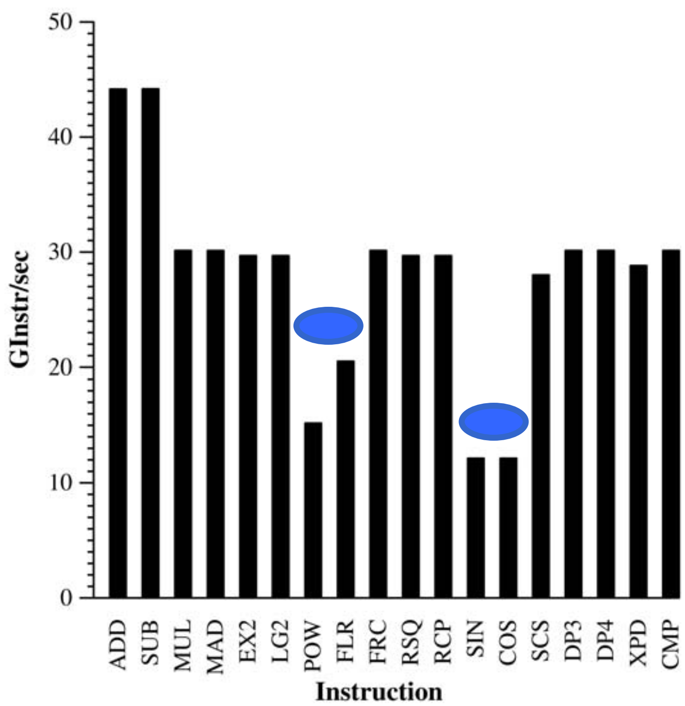
Instruction Issue Rate

- **Questions**
  - What is the raw performance achievable?
  - Do different instructions have different costs?
  - Vector vs. scalar issue differences?

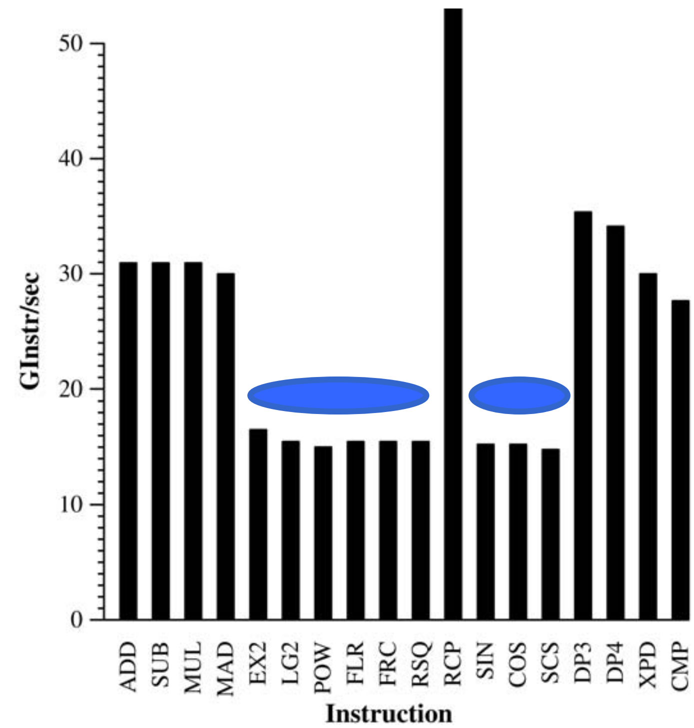
Methodology

- **Write *long* shaders with dependent instructions**
  - >100 instructions
  - All instructions dependent
    - But try to structure to allow for multi-issue
- **Test float1 vs. float4 performance**
- **GPUBench tests:**
  - instrissue

Results – Vector issue



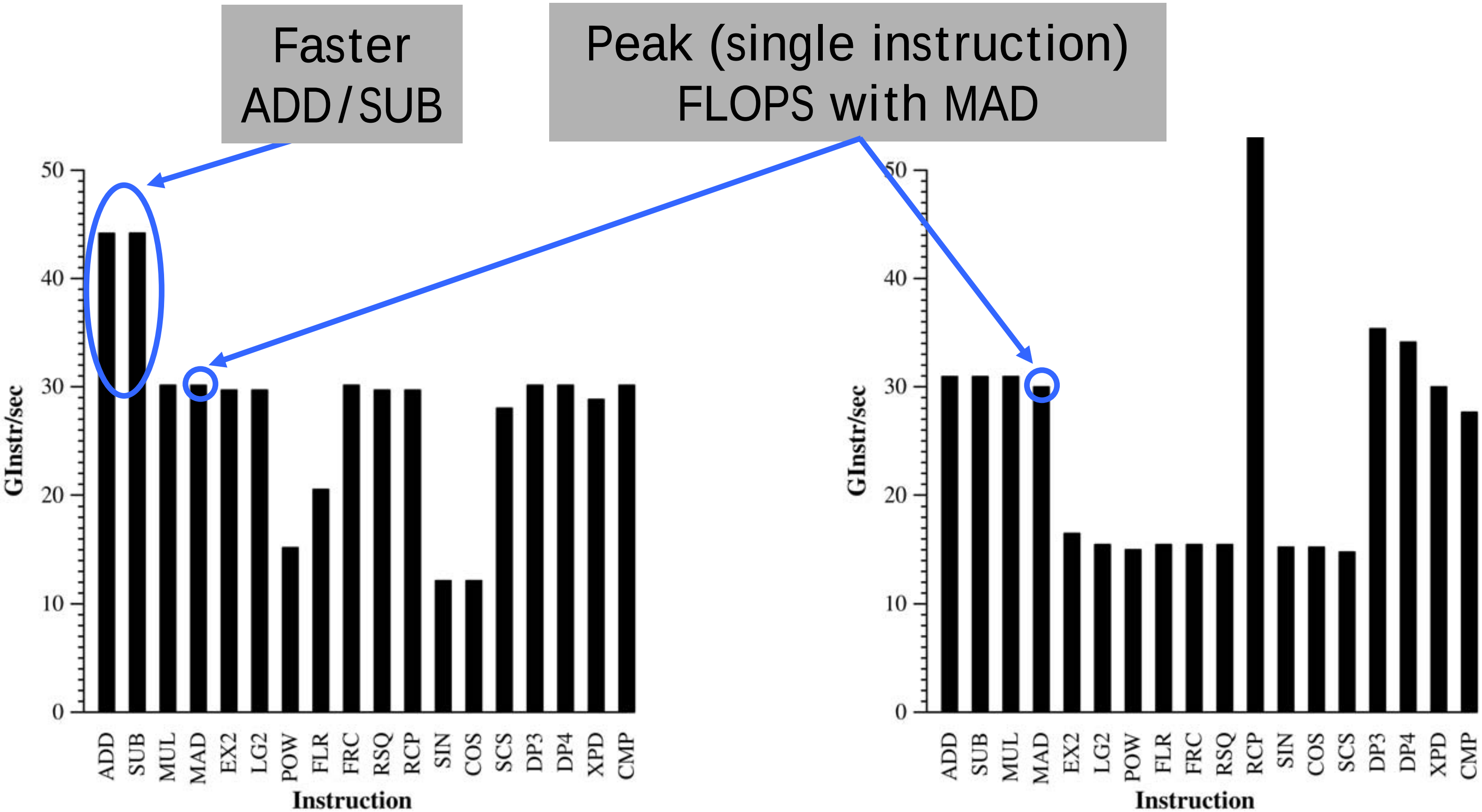
ATI X1900XTX



NVIDIA 7900GTX

= More costly than others

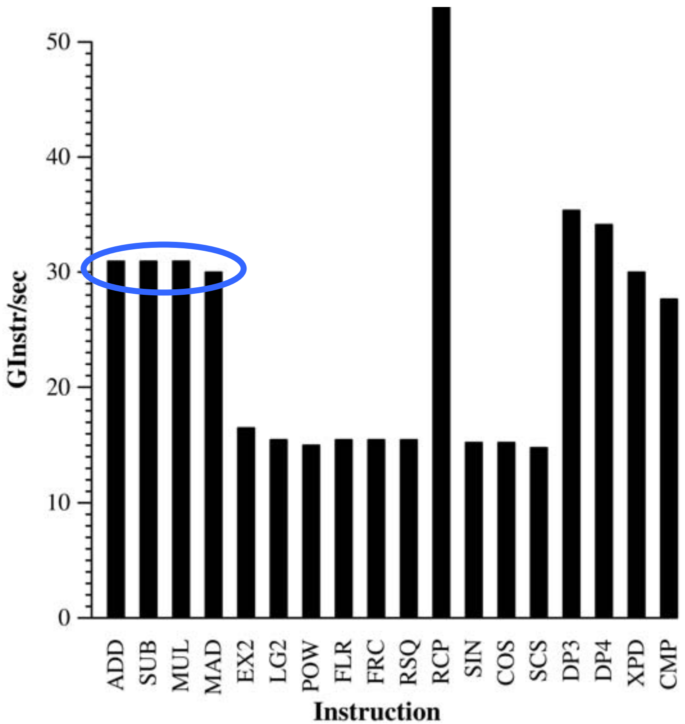
Results – Vector issue



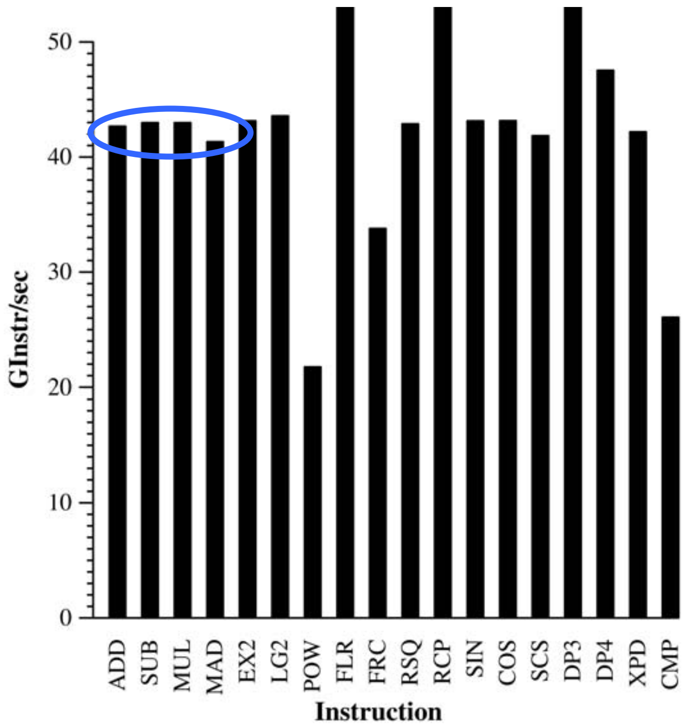
ATI X1900XTX

NVIDIA 7900GTX

Results – Vector issue



NVIDIA 7900GTX

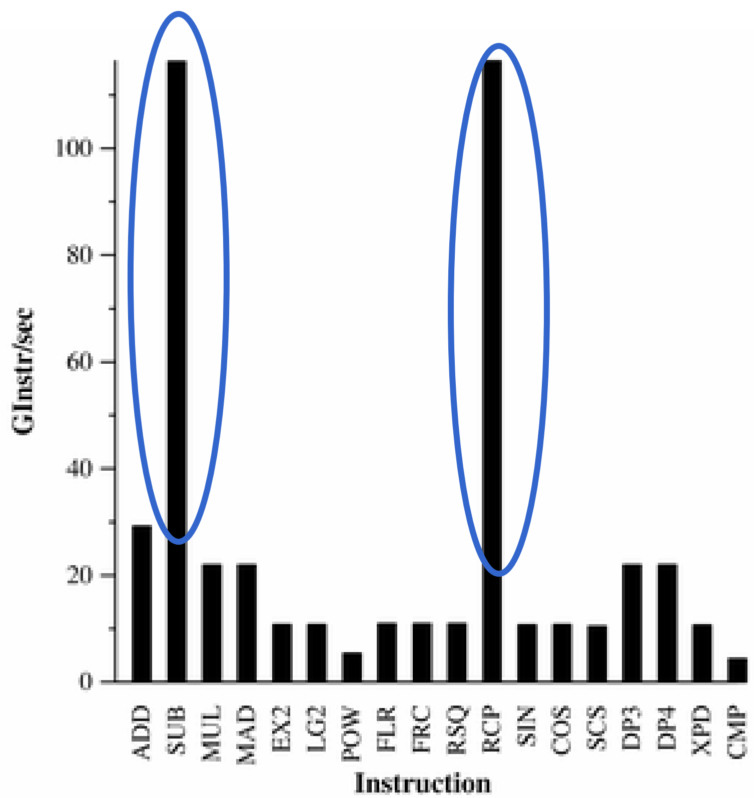


NVIDIA 8800GTX

8800GTX is 37%
faster (peak)

When benchmarks go wrong…

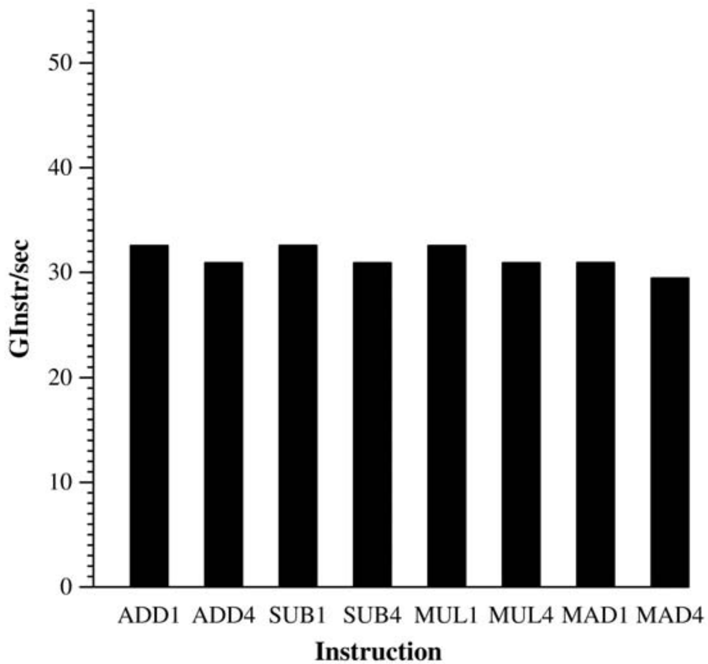
- **Smart compilers subverting testing and optimizing away shaders. Bug found in previous subtract test. No clever way to write RCP test found yet…** *Always sanity check results against theoretical peak!!!*



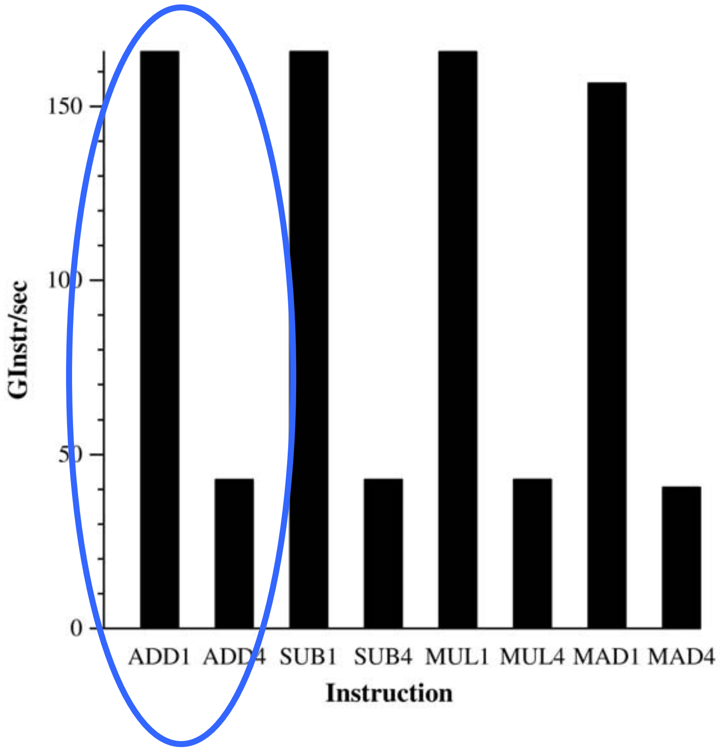
NVIDIA 7800GTX

GPUBench 1.2

Results – Scalar issue



NVIDIA 7900GTX



NVIDIA 8800GTX

8800GTX is a scalar issue processor

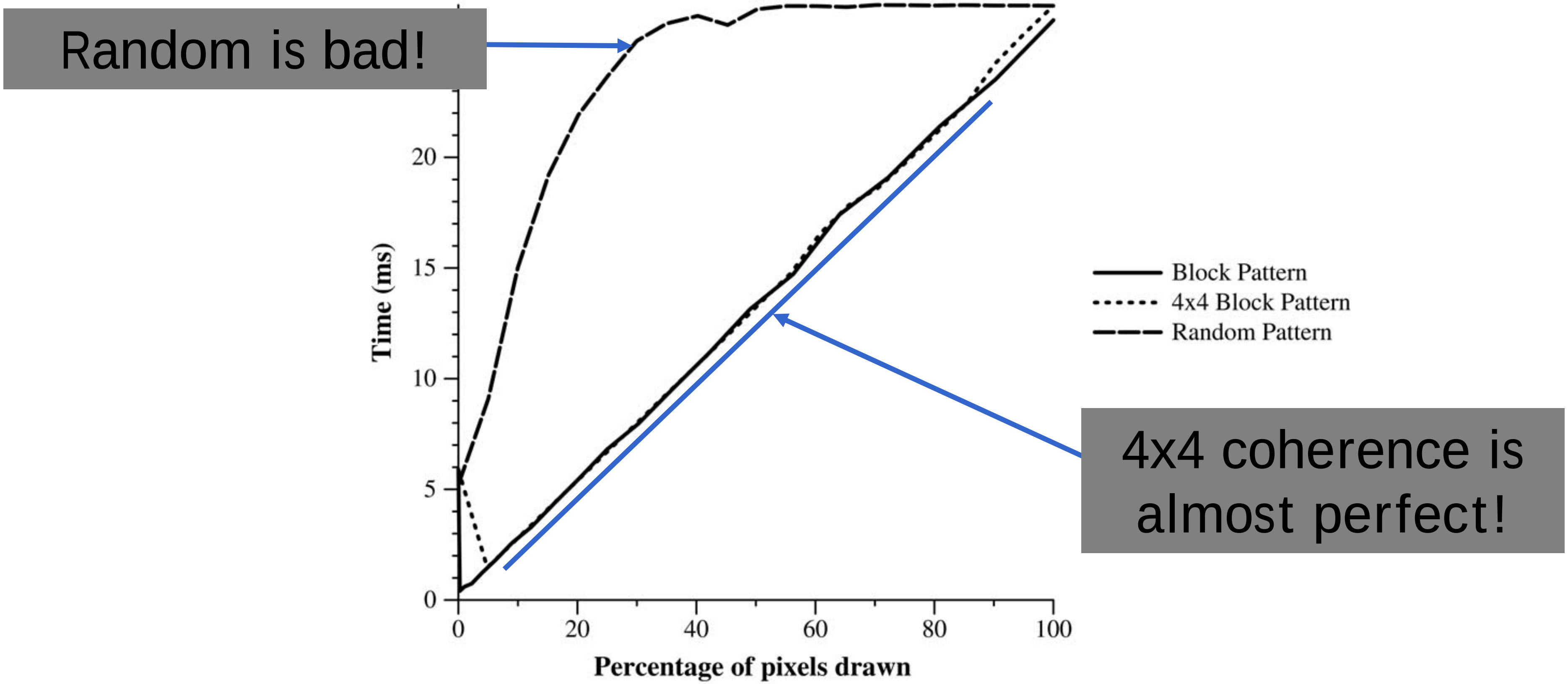
Branching Performance

- **Questions**
  - Is predication better than branching?
  - Is using “Early-Z” culling a better option?
  - What is the cost of branching?
  - What branching granularity is required?
  - How much can I really save branching around heavy computation?

Methodology

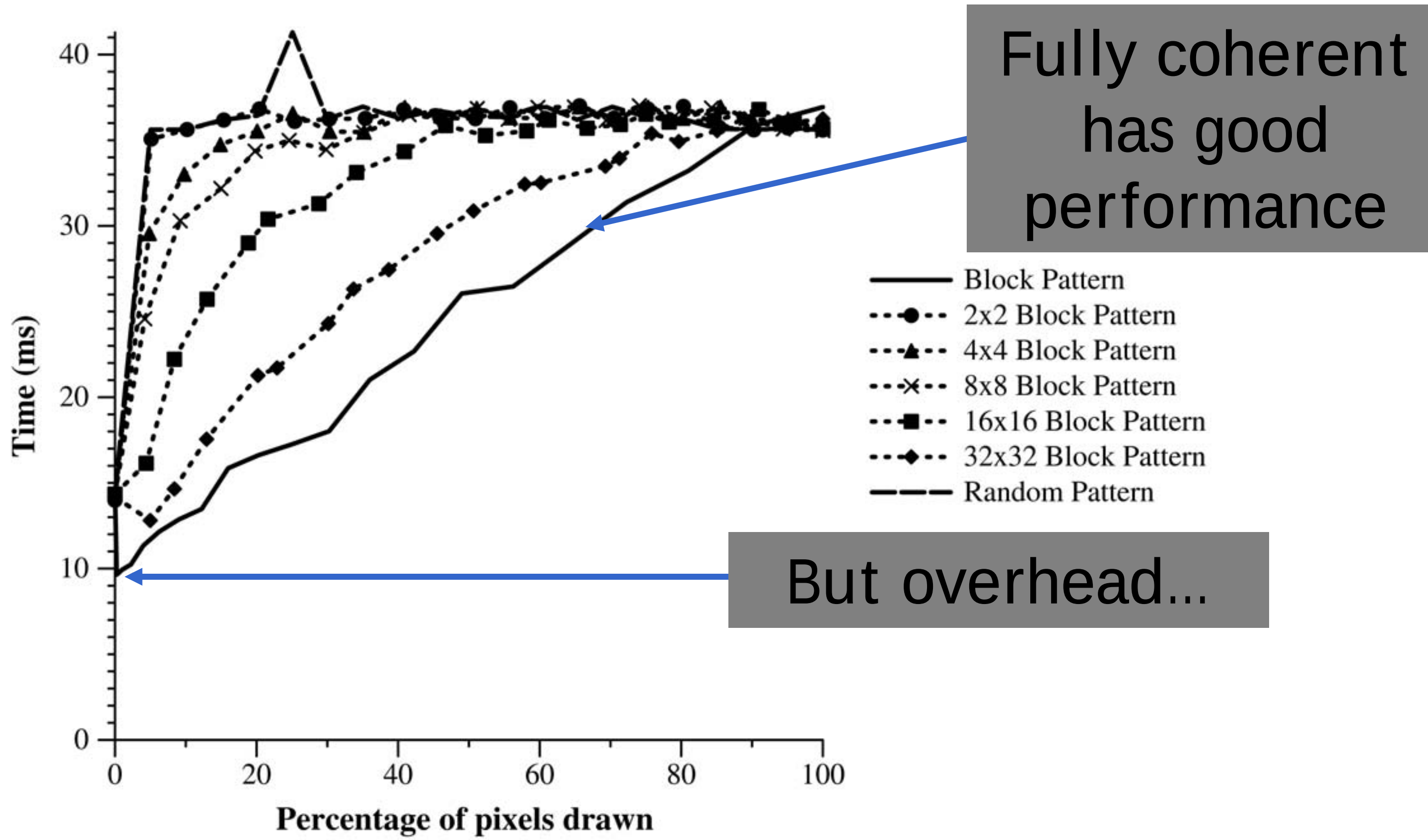
- **Early-Z**
  - Set a Z-buffer and compare function to mask out compute
  - Change coherence of blocks
  - Change sizes of blocks
  - Set differing amounts of pixels to be drawn
- **Shader Branching**
  - If{ do a little }; else { LOTS of math}
  - Change coherence of blocks
  - Change sizes of blocks
  - Have differing amounts of pixels execute heavy math branch
- **GPUBench tests:**
  - branching

Results – Early-Z - NVIDIA



NVIDIA 7900GTX

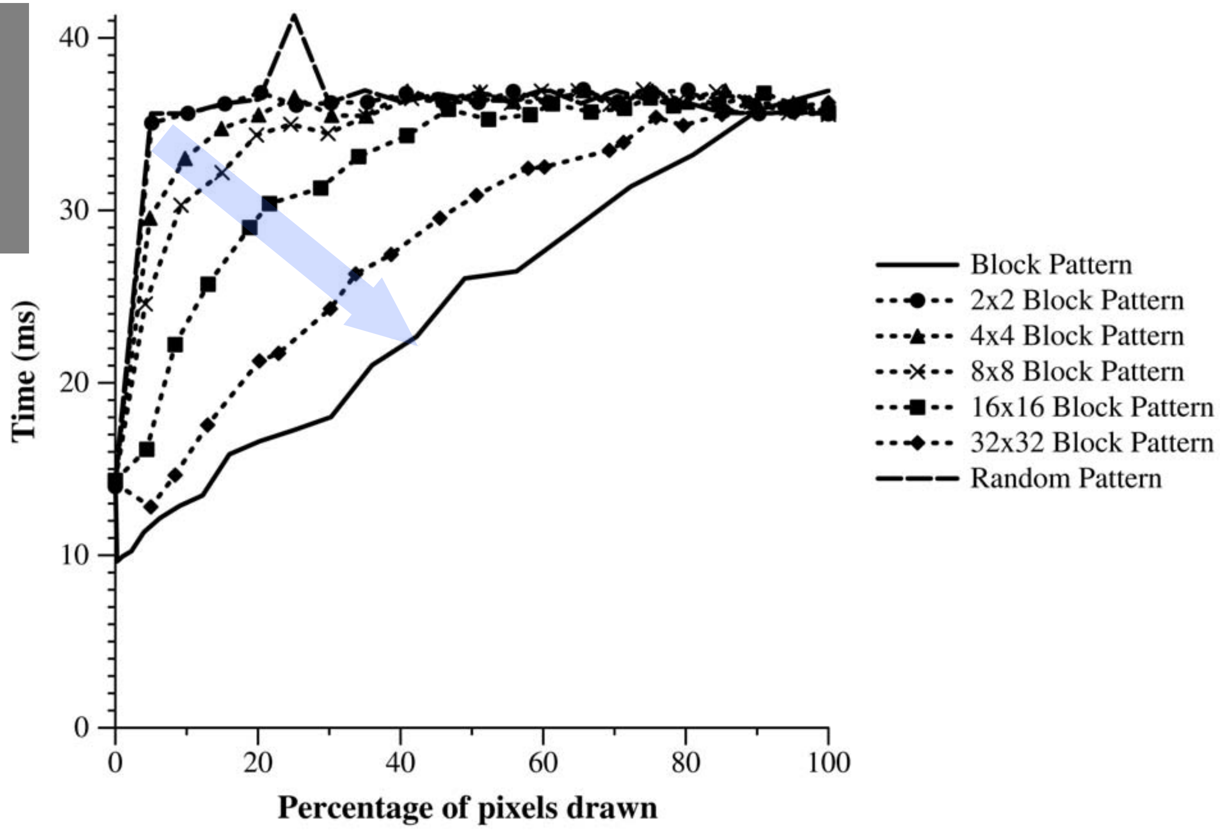
Results – Branching - NVIDIA



NVIDIA 7900GTX

Results – Branching - NVIDIA

Performance increases with branch coherence

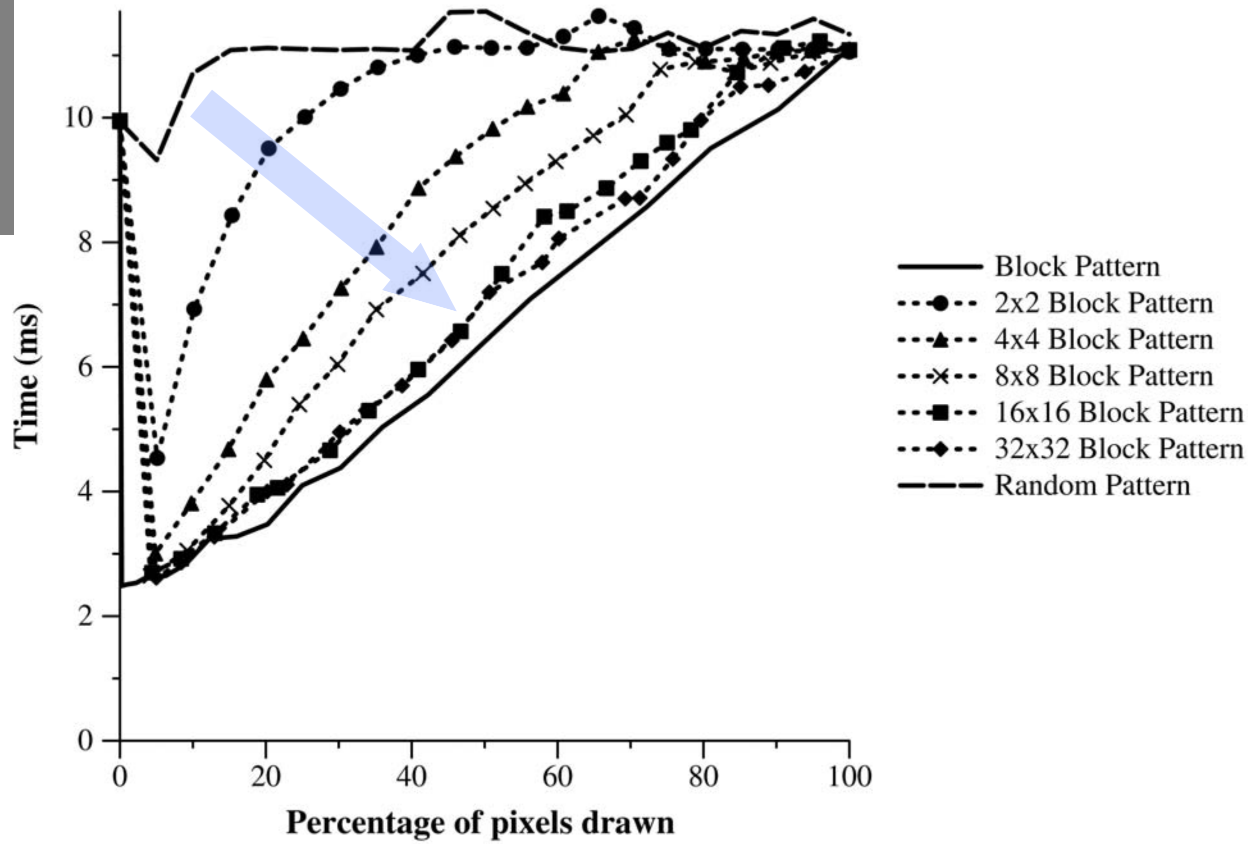


NVIDIA 7900GTX

Need > 32x32 branch coherence

Results – Branching - NVIDIA

Performance increases with branch coherence



NVIDIA 8800GTX

Need > 16x16 branch coherence
(Turns out 16x4 is as good as 16x16)

Summary

- **Benchmarks can help discern app behavior and architecture characteristics**
- **We use these benchmarks as predictive models when designing algorithms**
  - Folding@Home
  - ClawHMMer
  - CFD
- **Be wary of driver optimizations**
  - Driver revisions change behavior
    - Raster order, scheduler, compiler