

Vector Computing on GPUs

Egle Pilipaviciute
& Mark Robinson

Vector Computing



Not for everyone (not small)

What if vector computing was
for everyone?

GPUs: powerful vector processors



(smaller than Cray)

What is GPU?

- Graphical Processing Unit usually on a video card or on the motherboard
- Designed for high performance 2D and 3D graphical output (from logical coordinates to pixel data)

Outline

- GPU Technology
- Possible Applications
- Programming for a GPGPU
- Experiments
- Conclusion

Why GPUs?

- GPUs are cheap
 - Commodity hardware
- GPUs are fast
 - Speeding up computation-intensive algorithms
 - Evolving faster than CPU
 - Doubling every six months

Why GPUs?

- GPUs are programmable
- A lot of interest in research community:
 - SIGGRAPH 2004
 - VIS 2004

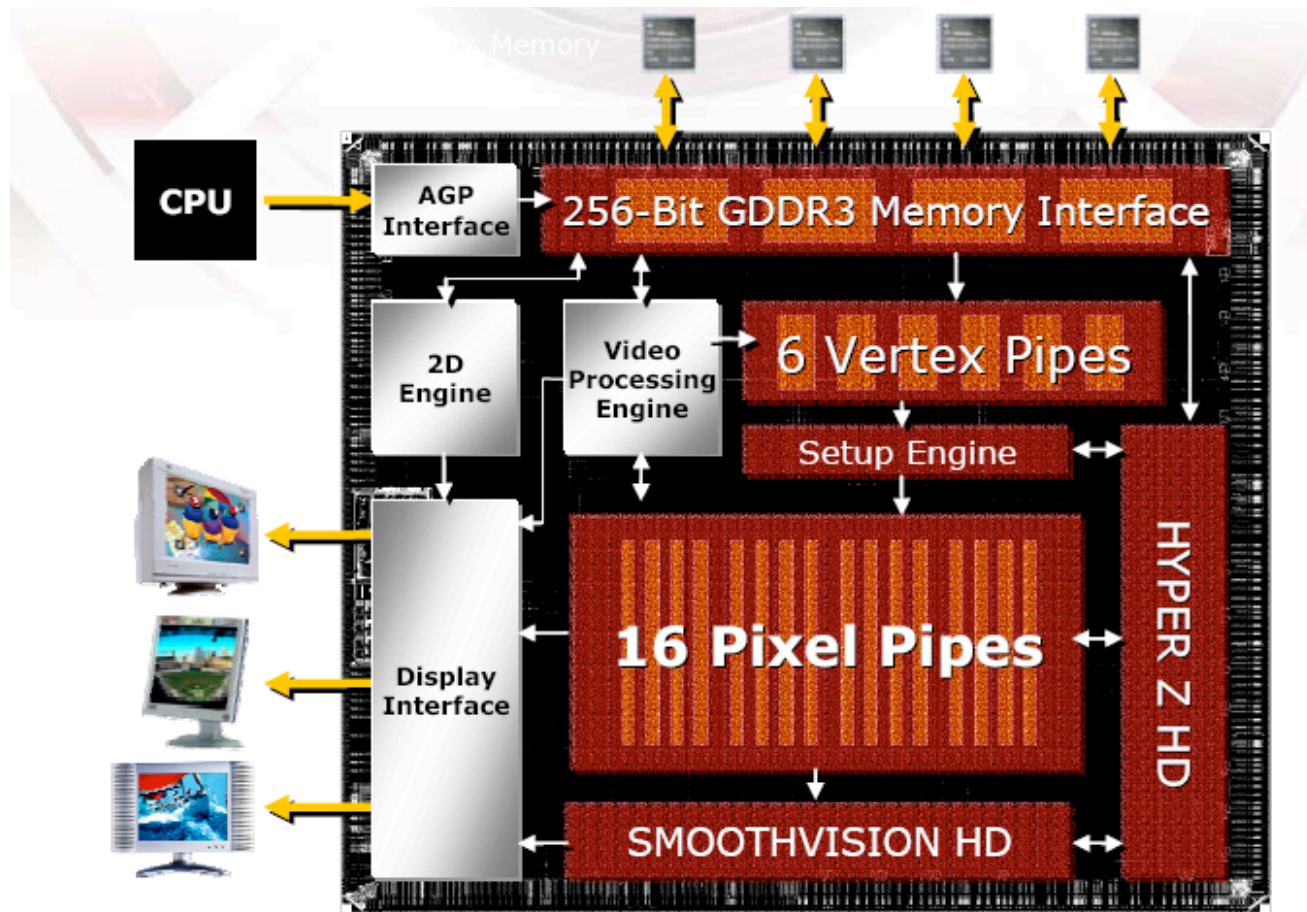
Motivation

- Personal interest
- Potential impact on visualization research
- Fall 2004 Computer Architecture project

Why Are GPUs Fast?

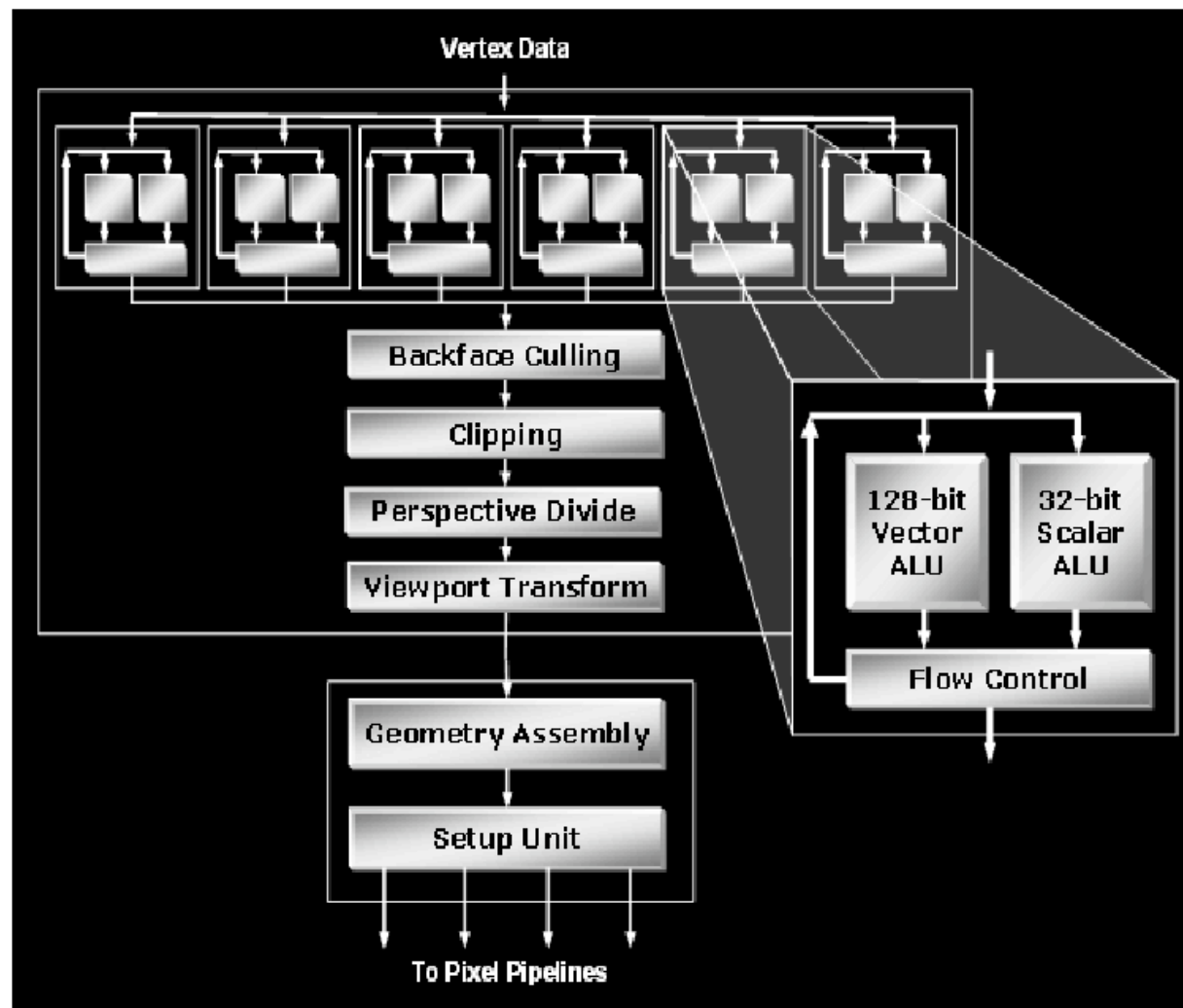
- Transistors:
 - Latest NVidia graphics card: 222 million transistors
 - Latest AMD Athlon: 105.9 million transistors
 - Latest Intel Pentium 64 bit: 169 million transistors
- Memory Bandwidth:
 - Current commodity GPUs around 25-30 GB/s
 - Top of the line motherboards around 8GB/s
- GPUs don't have to multi-task
- Computations done in parallel

Current top-of-the-line GPU



Radeon X800 Simplified Block Diagram

Vertex Processing Unit



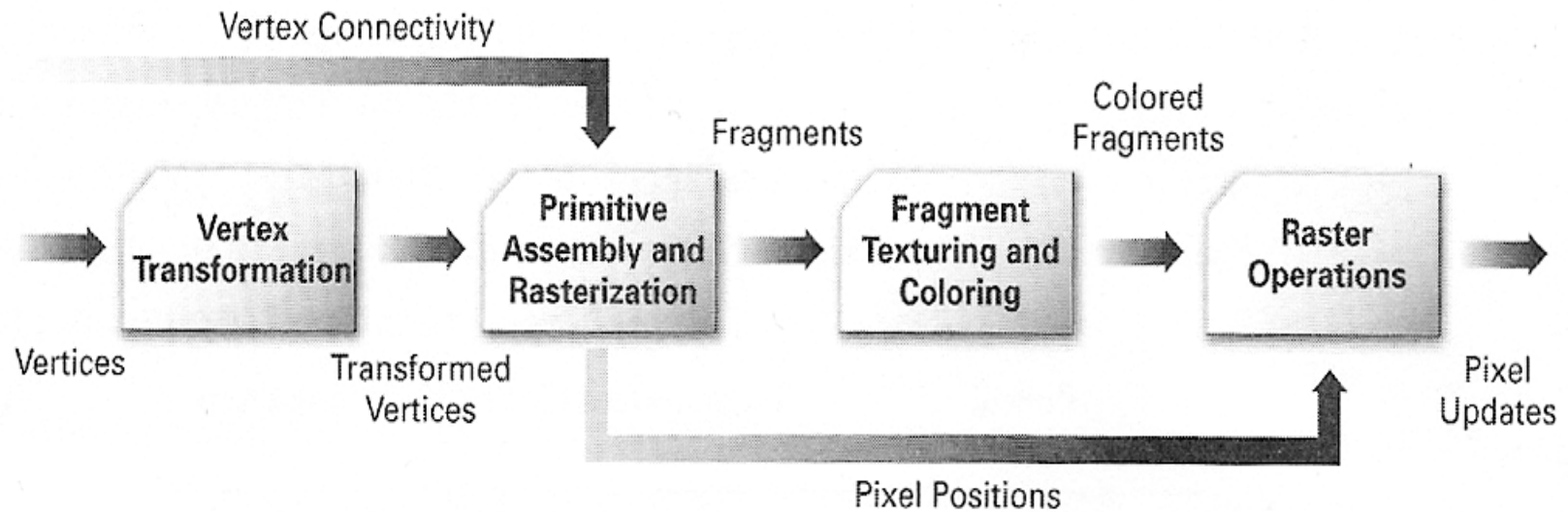
Vertex Processing Unit

- Performs many computations
- Each includes two ALUs
 - 128-bit floating point vector ALU
 - 32-bit floating point scalar ALU
- Flow Control (loops, subroutines)

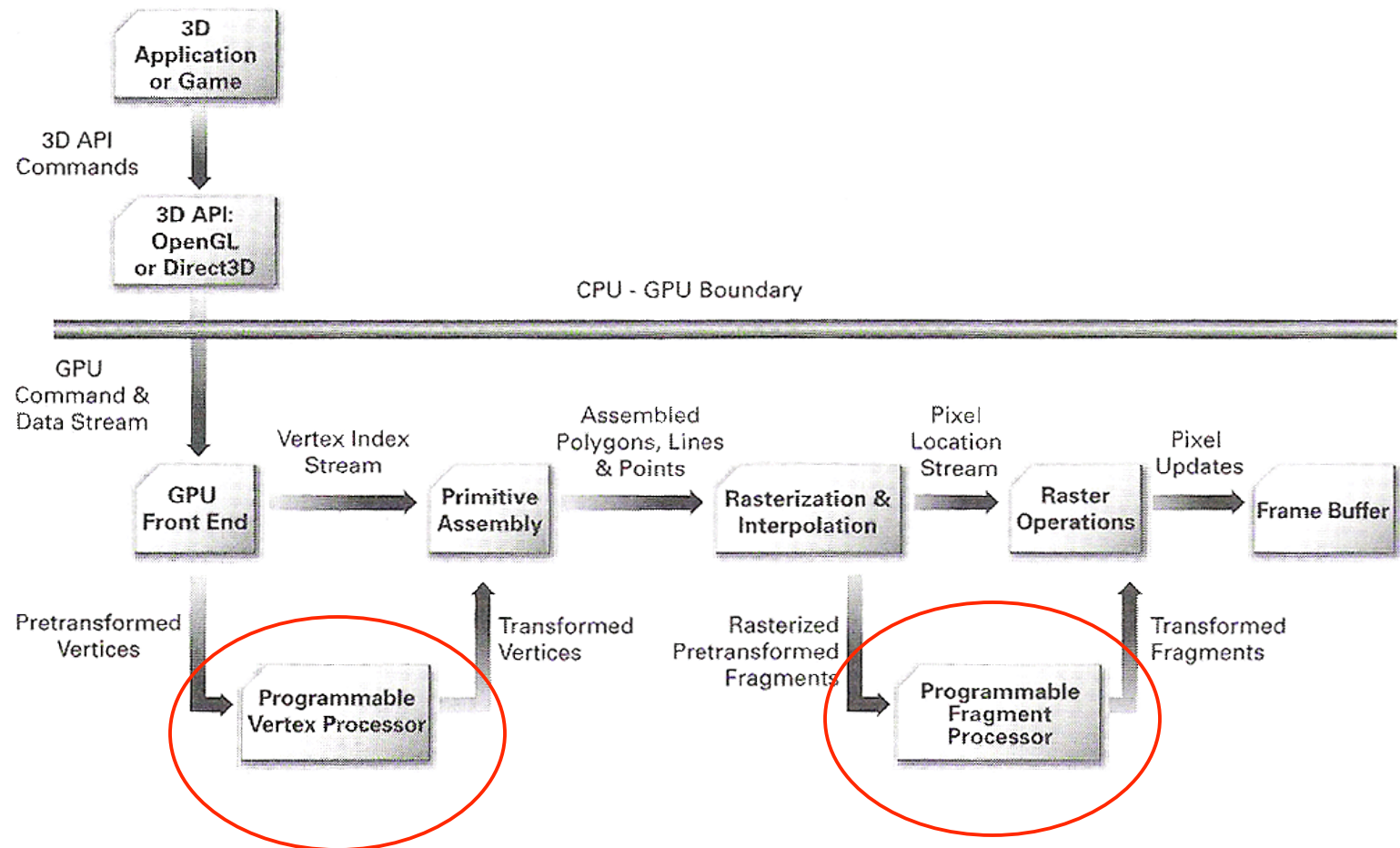
Pixel Processing Unit

- Eight operations per cycle per pipeline for pixel shading
- 16 Pixel pipelines each containing
 - two shader units with
 - superscalar architecture
 - one floating point texture processor
- 32 bit pipeline

Graphics Pipeline



Programmable Graphics Pipeline



Calculations in graphics
pipeline for **general
computing** performed in
vertex processor and in
fragment processor

Vector Math

- Both processors fetch instructions in a loop until program terminates
- Necessary operations:
 - Add, multiply, multiply-add, dot product, minimum and maximum
 - Branching and relative addressing added recently

Applications

- GPU can be applied to:
 - Order Statistics
 - Matrix Multiplication
 - Stream Processing
 - Other (Sorting, Databases, geometry problems, fast wavelet transform ...)

Order Statistics

- On CPU: Selecting k -th largest element in $O(n)$ time (worst case linear time)
- On GPU: selecting k -th largest element in $O(\log n)$ time

Matrix Multiplication

- On CPU, naïve algorithm runs in $O(n^3)$, Strassen's algorithm in $O(n^{2.81})$
- How about GPU?
 - Strassen's algorithm in parallel

Matrix Multiplication

- Using Strassen's algorithm in parallel, matrix multiplication runs in

$$O(n^{\log 7/3}) = n^{0.93}$$

Stream Processing

- Streams expose parallelism in the application → good match to the GPU computing model.
- Stream elements can be processed in parallel using data-parallel hardware

Brook for Stream Processing

- Brook- language for streaming processors
- Extension of C
- Native support for streams
- Adapted to graphics hardware
- Free of graphics constructs

Streams

- A collection of data which can be operated on in parallel in one pass
- Shape of the stream refers to its dimensionality
- Similar to C arrays, but accessible only via kernels and stream operators

Brook Streams

- Shape of the stream refers to its dimensionality
- Similar to C arrays, but accessible only via kernels and stream operators

```
float s<10, 10>;
```

```
streamRead(s, data_s); /* Fill stream s<> with the data at *data_s */
```

```
streamWrite(s, data_s); /* Fill the data at *data_s from stream s<> */
```

Stream Shape Matching

- For different shape streams, kernels resize the input stream to match the output stream:
 - Repeating(123 to 111222333)
 - Striding (123456789 to 13579)

Kernels

- Brook kernels are functions that operate on streams
- Kernel performs an implicit loop over the elements of the stream
- Different from vector operators
- One input stream, many output streams

Calling a Kernel:

```
kernel void k(float s<>, float3 f, float a[10][10],  
              out float o<>);  
  
float a<100>;  
float b<100>;  
float c<10,10>;  
streamRead(a, data1);  
streamRead(b, data2);  
streamRead(c, data3);  
// Call kernel "k"  
k (a, 3.2f, c, b);  
streamWrite(b, result);
```

Brook Abstractions

- Memory is managed via streams: named, typed and shaped data objects consisting of collections of records
- Data-parallel operations executed on GPU are specified as calls to parallel functions called kernels
- Many-to-one reductions on stream elements are performed in parallel by reduction functions

Programming for a GPGPU

What's Needed

- Compiler (C++ or Java)
- 3D Graphics interface library (OpenGL, Direct3D)
- Shader language and compiler (Cg, HLSL)
- Requirements: know thine target

Our Tools

- Mac OS/X 10.3.8
- Xcode 1.5 (C++)
- OpenGL interface libraries
- Cg Runtime 1.3 (framework for Xcode)
- Development platform: Mac PowerBook G4 (ATI Mobility Radeon 9700, 4 pixel shaders, 2 vertex shaders, 128MB VRAM)
- Our target shader: pixel/fragment

Simple Fragment App. Flow

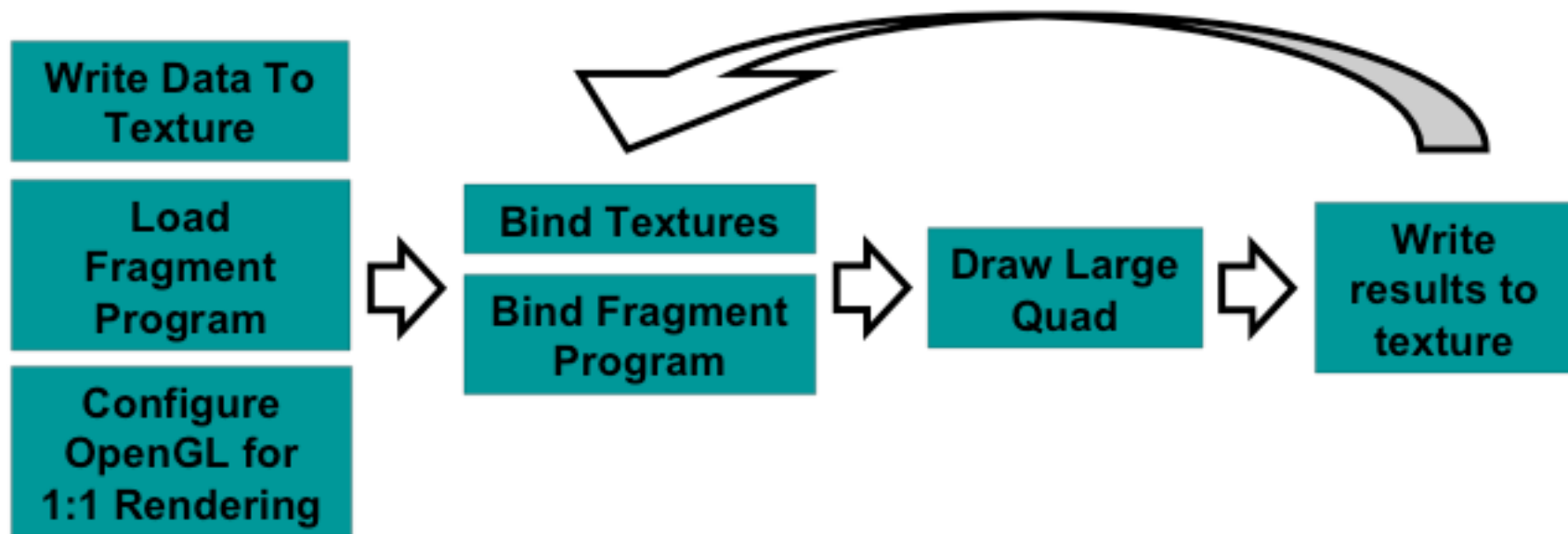


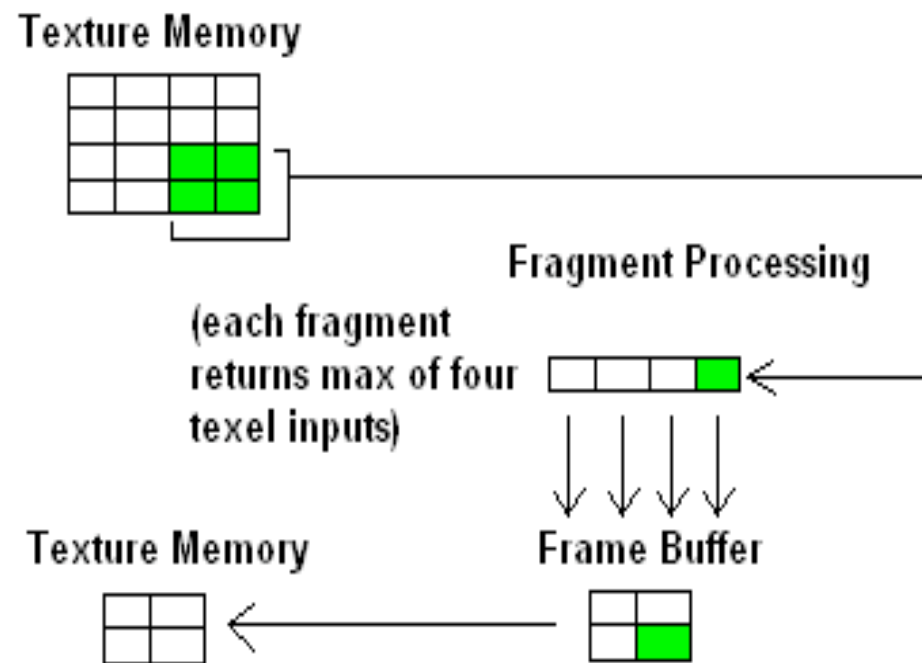
Figure from of Aaron Lefoun's tutorial, Viz 2004

Example: Reduction (max)

- Goal: Find maximum element in an array of n elements
- Approach: each fragment processor will find max of 4 adjacent array elements (each pass processes 16 elements)
- Input: array of n elements stored as 2D texture
- Output: array of $n/4$ elements to frame buffer (each pass overwrites the array)

GPU Reduction

1. Store array as 2D texture
2. Max() comparison runs as fragment program
3. Each fragment compares 4 texels and returns max
4. Frame buffer stores max from each fragment (buffer quarters original array size)
5. Frame buffer overwrites previous texture



The C code

- Initialize
 - Create random array of n elements and write to texture memory: $\text{sqr}(n) \times \text{sqr}(n)$
`glTexImage2D()`
 - Load Cg fragment
`cgCreateProgramFromFile()`
- During a single rendering pass
 - OpenGL Display() event
 - Bind texture and Cg program
`cgGLBindProgram()`
`glBindTexture()`

More C Code

- Iterate $\log_4 n$ times

- Create viewport:

```
glViewport(0, 0, sqr(n)/2, sqr(n)/2);  
gluOrtho2D(0.0, sqr(n)/2, 0.0,  
sqr(n)/2);
```

*****the viewport is 1/4 the texture size**

The Cg Code

- “: COLOR” indicates that the output from the function will be a value to the frame buffer
- texcoord ranges from (0,0) to (sqr(n), sqr(n))
- img is the 2D texture array

```
float4 main( float2 texcoord : TEXCOORD0,  
             uniform sampler2D img ) : COLOR {  
  
    float4 a, b, c, d;  
    a = tex2D( img, texcoord );  
    b = tex2D( img, texcoord + float2(0,1) );  
    c = tex2D( img, texcoord + float2(1,0) );  
    d = tex2D( img, texcoord + float2(1,1) );  
  
    return max( max(a,b), max(c,d) );  
}
```

And Still More C Code

- **Project a quad onto the viewport**

`glBegin(GL_QUADS)`

`glTexCoord2f(0,0 ... $\text{sqr}(n)$, $\text{sqr}(n)$)`

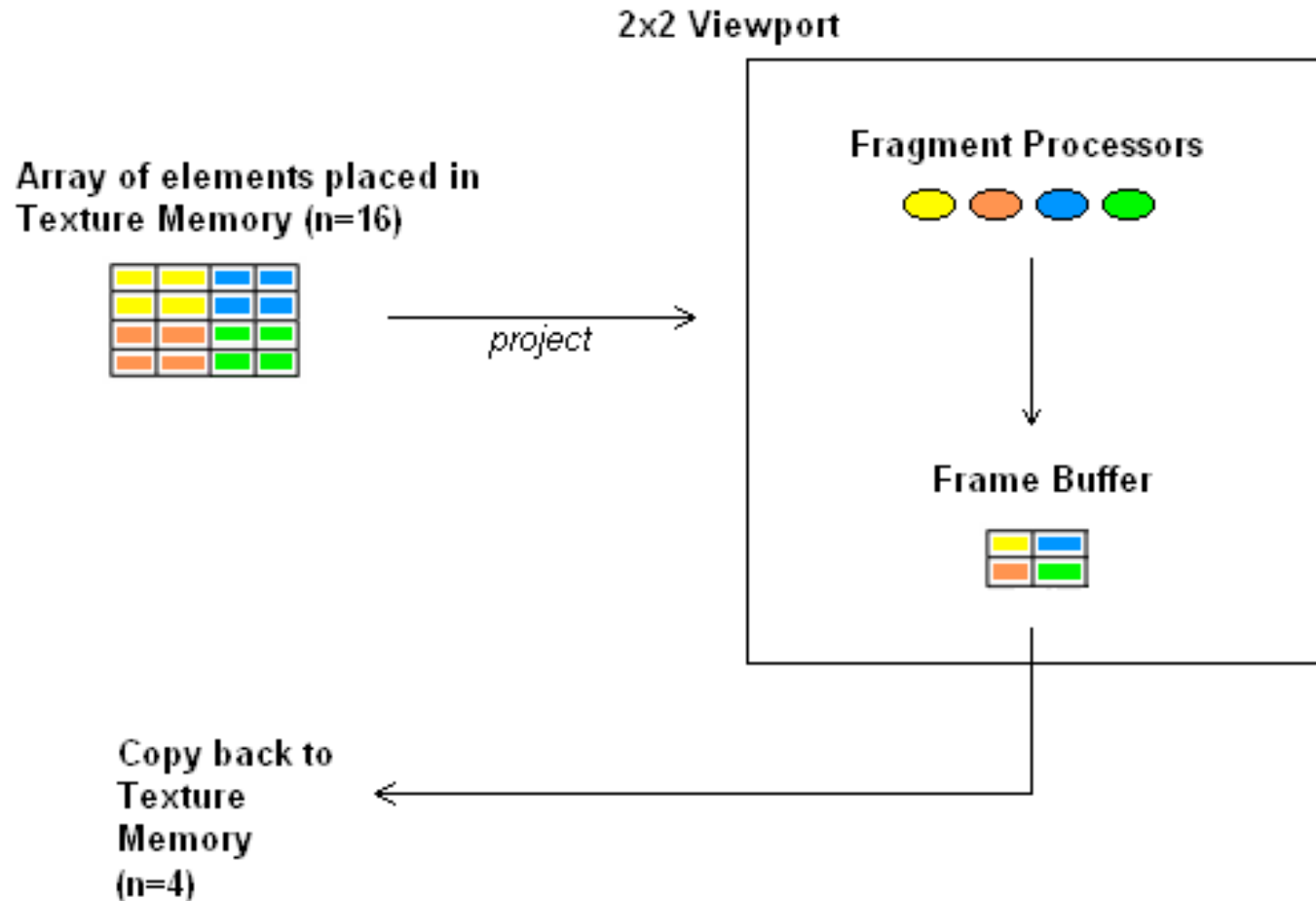
`glVertex2f(0,0 ... i , i)`

- Copy contents of frame buffer to texture

`glCopyTexSubImage2D()`

- Done: texture memory holds 1 element (the max)

Another look at Reduction Loop



Experimentation

4 Problems

- Reduction: Max
- Sorting
- Searching
- Matrix Multiplication

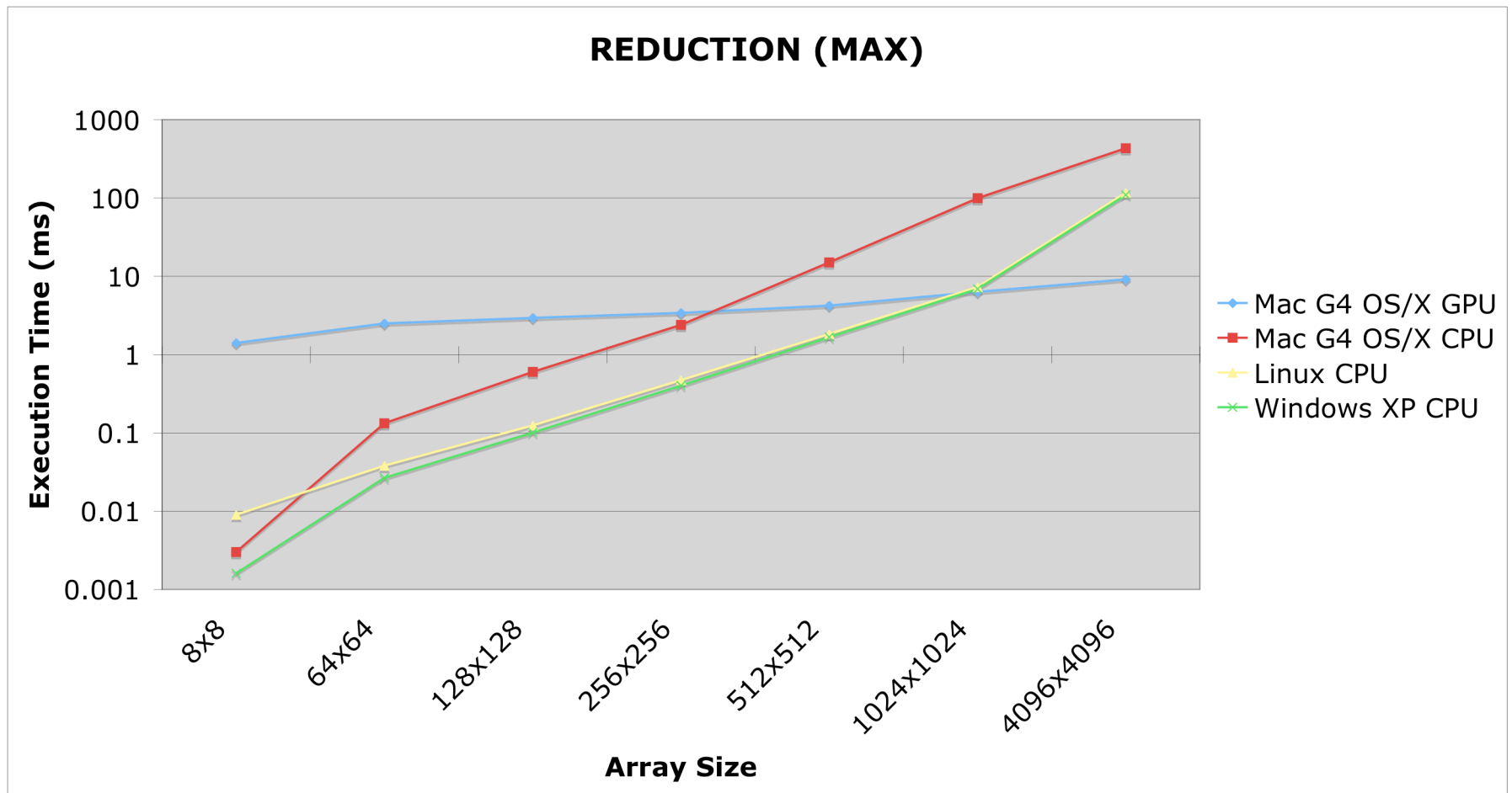
4 Platforms

- GPU: Mac PowerBook G4 (ATI Mobility Radeon 9700, 4 pixel shaders, 2 vertex shaders, 128MB VRAM)
- CPU1: Mac PowerBook G4 (PowerPC G4 1.5GHz, 1GB 333MHz DDR SDRAM)
- CPU2: Linux (Intel Celeron 2.4GHz, 1GB RAM)
- CPU3: Windows XP (Intel Pentium 4 2.6GHz, 1GB RAM)

Reduction: Max

- Find maximum element in an array of n elements
- CPU implementation: standard array traversal in $O(n)$
- GPU implementation: $O(\log_4 n)$

Results: Reduction

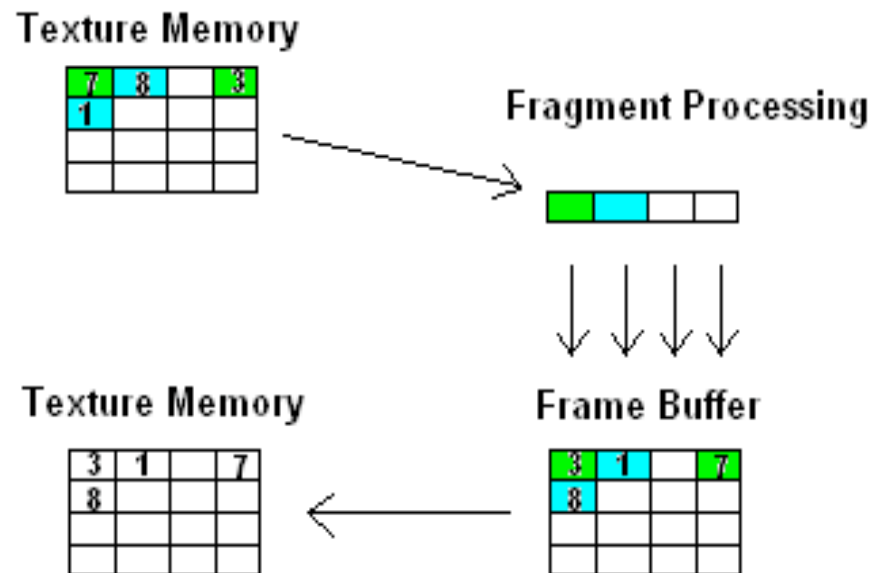


Sorting

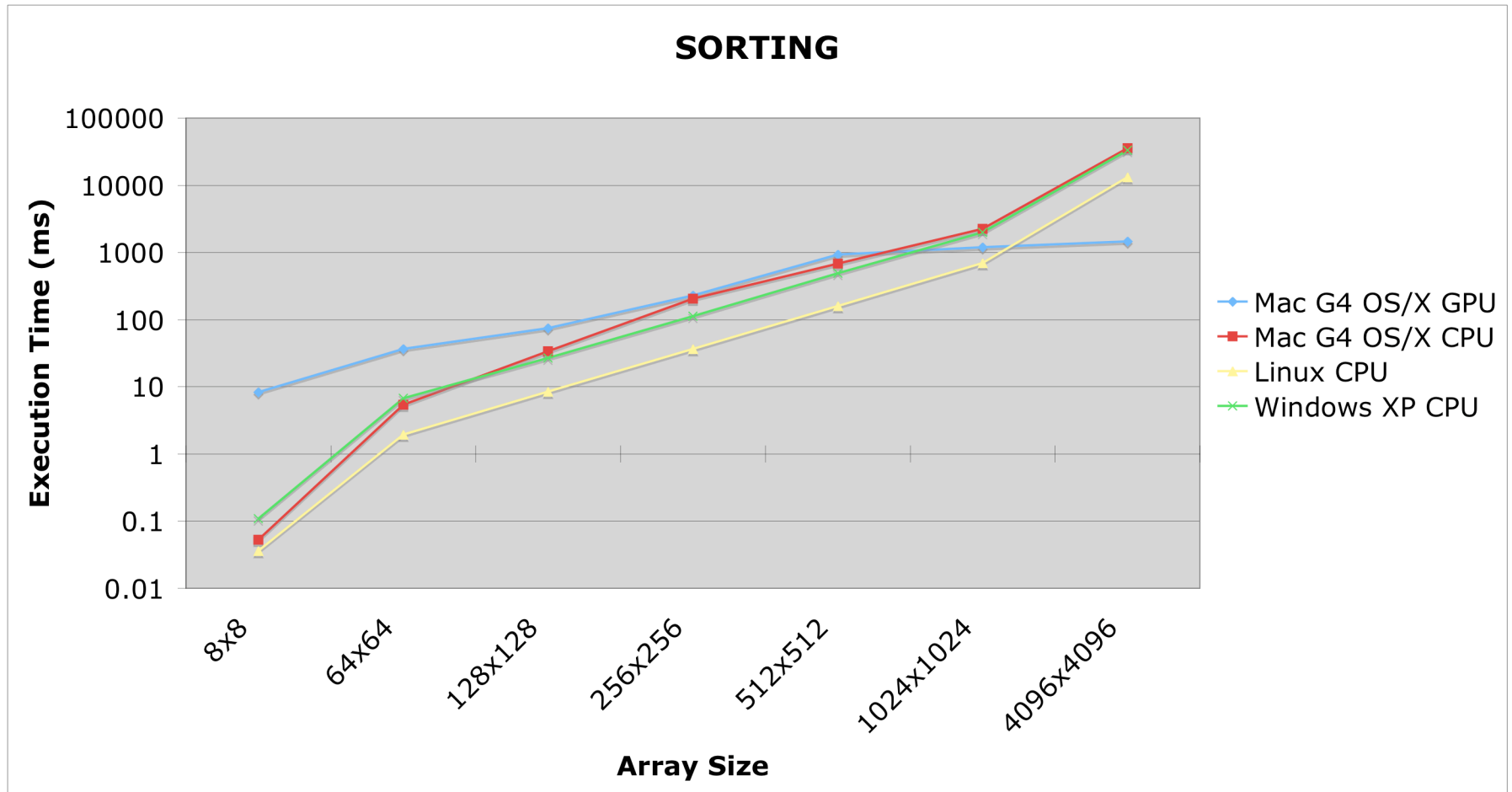
- Sort an array of n floats
- CPU implementation: standard merge sort in $O(n \lg n)$
- GPU implementation: bitonic merge sort in $O(\lg n \lg n)$

GPU Bitonic Sort

1. Store array as 2D texture
2. Bitonic sort runs as fragment program
3. Each array position sorts vs. partner position determined by current stage and step of bitonic sort
4. Sorted array position updated in frame buffer
5. Frame buffer overwrites texture



Results: Sorting

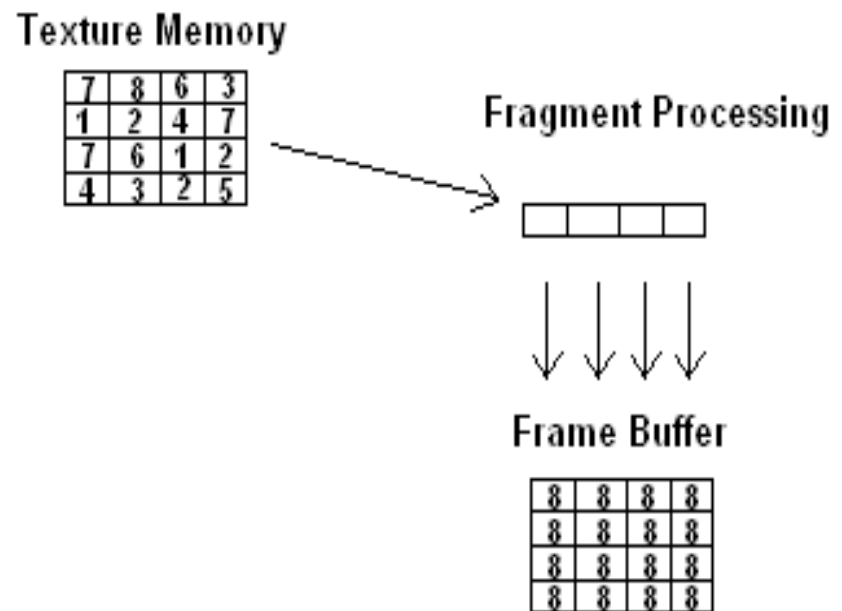


Searching

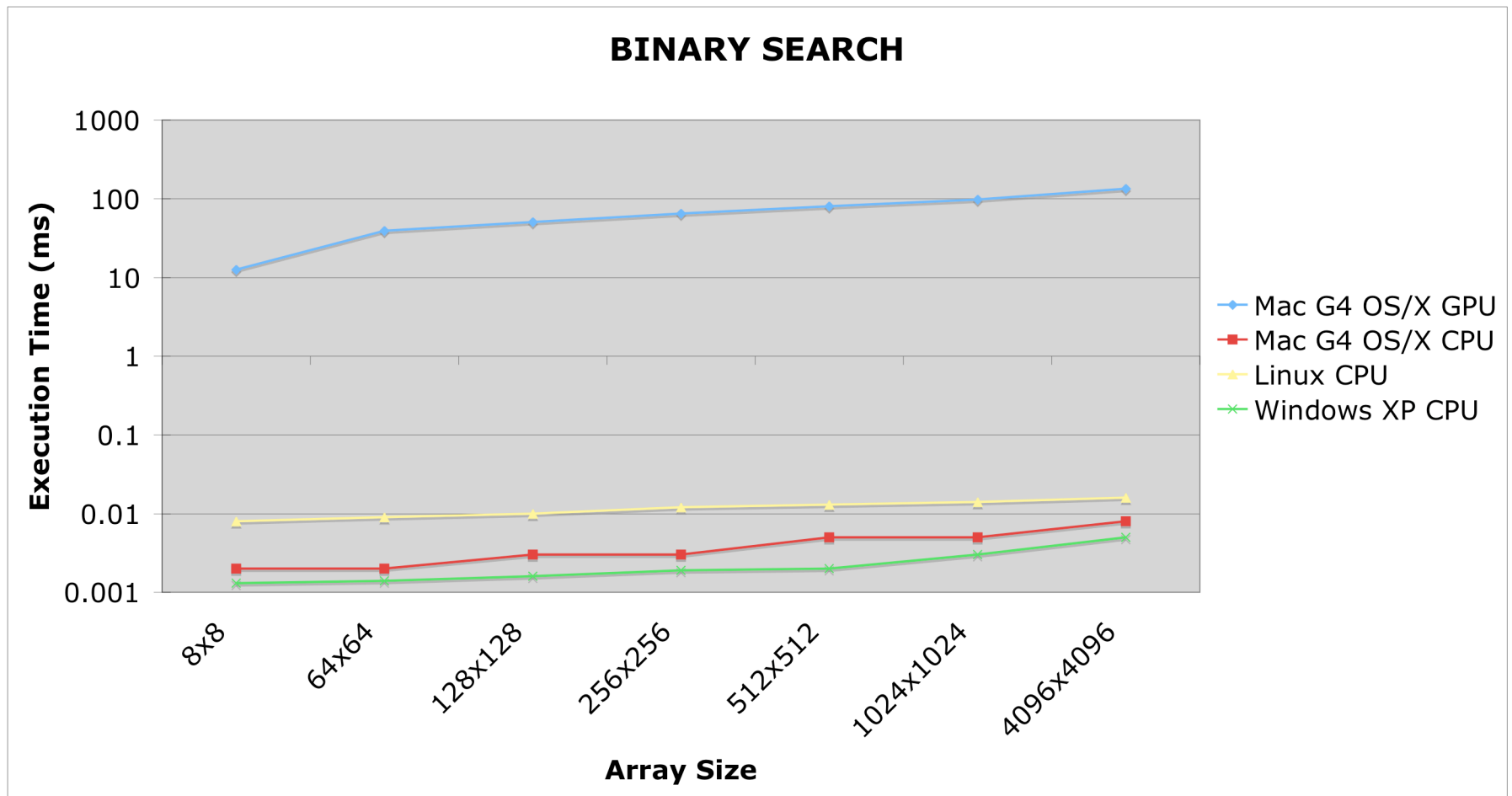
- Find an arbitrary element x in an array of n elements
- Assume array is pre-sorted
- CPU implementation: standard binary search in $O(\lg n)$
- GPU implementation: binary search in $O(n \lg n)$

GPU Binary Search

1. Store array as 2D texture
2. Search runs as fragment program on each element in array (inefficient)
3. Search result output to frame buffer



Results: Searching

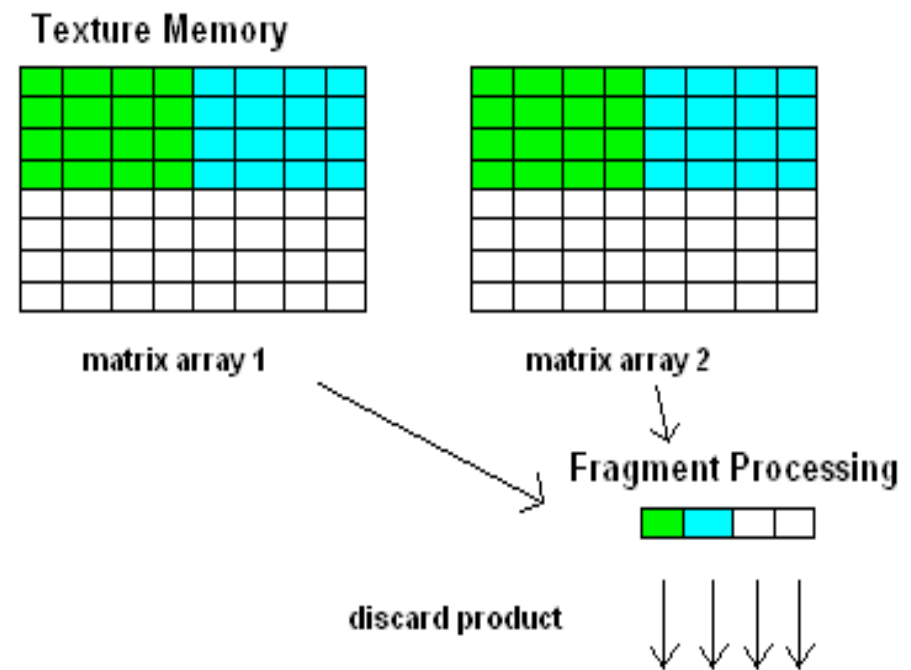


Matrix Multiplication

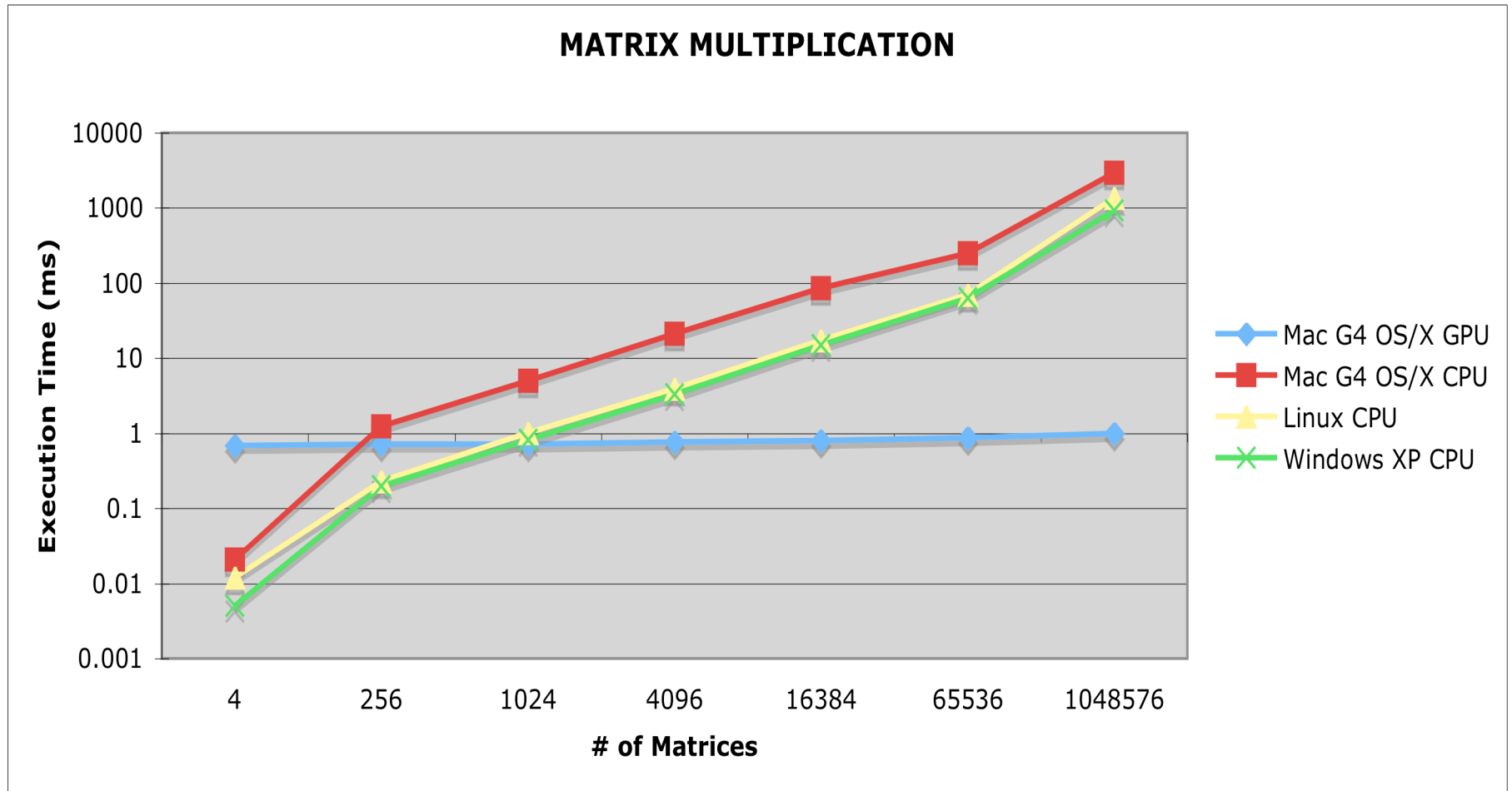
- Multiply 4x4 matrices from 2 arrays of n matrices, discarding the product

GPU Matrix Multiplication

1. Store each matrix array as a 2D texture
2. Matrix multiplication runs as fragment program
3. Each fragment loads a 4x4 matrix from each texture and calls a hardware matrix multiplication operation



Results: Matrix Multiplication



Conclusion

GPU easily outperforms CPU when:

- Problem is suited to parallelism
- Data set is large (but not larger than video memory)
- GPU instruction set can accommodate the needs of problem in an efficient manner

GPU Pros

- GPU hardware more powerful than CPU (ops/sec, memory bandwidth, and no multi-tasking)
- GPU hardware evolving much faster than CPU
- GPU technology is affordable and ubiquitous
- Built for parallel processing
- Development tools becoming high-level

GPU Cons

- Problem setup takes more time than CPU
- Limited instruction set
- Best suited for parallel problems
- Memory smaller than main memory
- Slow to write back to memory
- Limited return values from vertex and fragment programs
- Limited development tools
- Standards still in flux
- Variety of GPU platforms and feature sets

References

- BrookGPU:
<http://graphics.stanford.edu/projects/brookgpu/index.html>
- General-Purpose Computation Using Graphics Hardware: www.gpgpu.org
- Randima Fernando and Mark J. Kilgard, The Cg Tutorial, Pearson Education, Inc., Boston, MA., 2003
- nVidia Corporation, GPU Gems, Pearson Education, Inc., Boston, MA., 2004