
xna
Gamefest
MICROSOFT GAME TECHNOLOGY CONFERENCE 2008
Microsoft

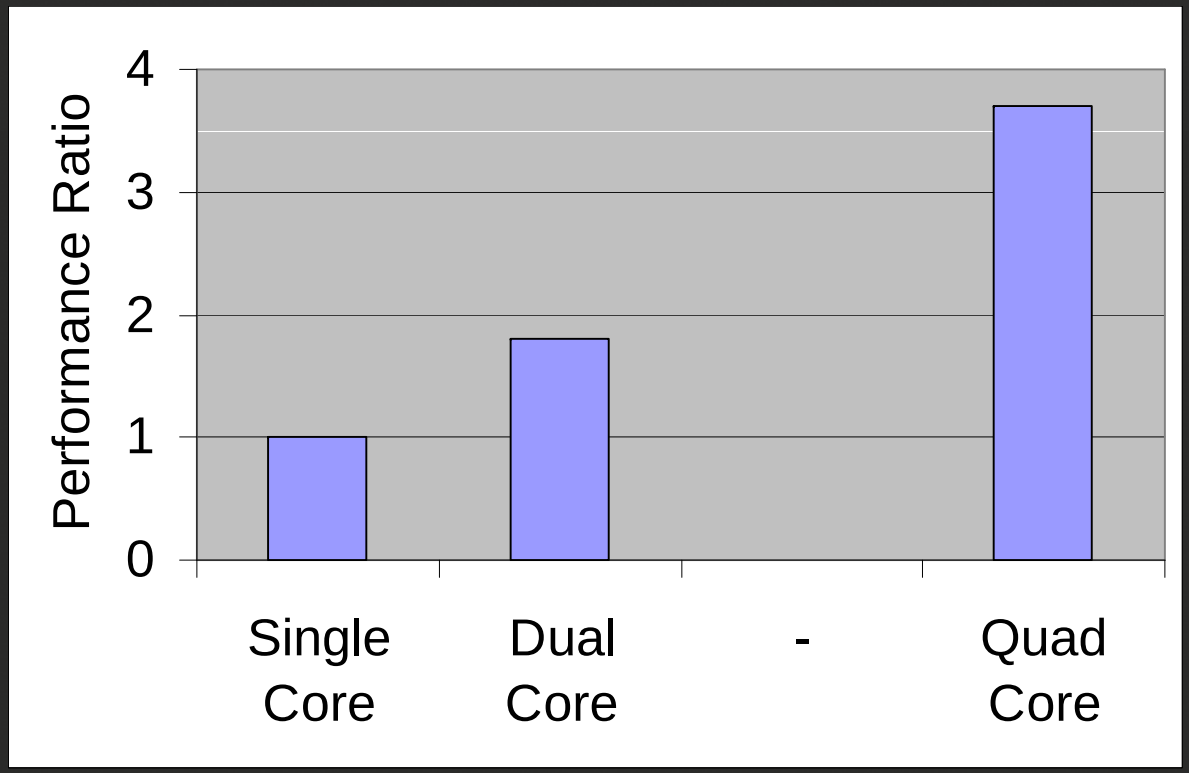
Practical Parallel Rendering with DirectX 9 and 10

Windows PC Command Buffers

Vincent Scheib
Architect
Gamebryo
Emergent Game Technologies

Introduction

- Take advantage of multiple cores with parallel rendering
- Performance should scale by number of cores



Observed data from this project, details follow

Presentation Outline

Motivation and problem definition

- Command buffers
  - Requirements
  - Implementation
  - Handling effects and resources
- Application models
- Integrating to existing code
- Prototype results
- Future work

Motivation

- Take advantage of multi core machines
  - 40% machines have 2+ physical CPUs (steamJul08)
- Rendering can have high CPU cost

- Direct3D 11 display lists coming, but want support for Direct3D 9 and 10 now
  - Currently 81% DX9 HW, 9% DX10 HW (steamJul08)
  - Rough DX9 HW forecast: 2011 ~30% (emergent)
  - Asia HW trends lag somewhat

Multithreaded DX Device?

- DirectX 9 and 10 primarily designed for single-threaded game architectures
- Multithreaded mode incurs overhead
  - Cuts FPS roughly in half on DX9 for a CPU render call bound application
- DX is Stateful
  - Requires additional synchronization for parallel rendering

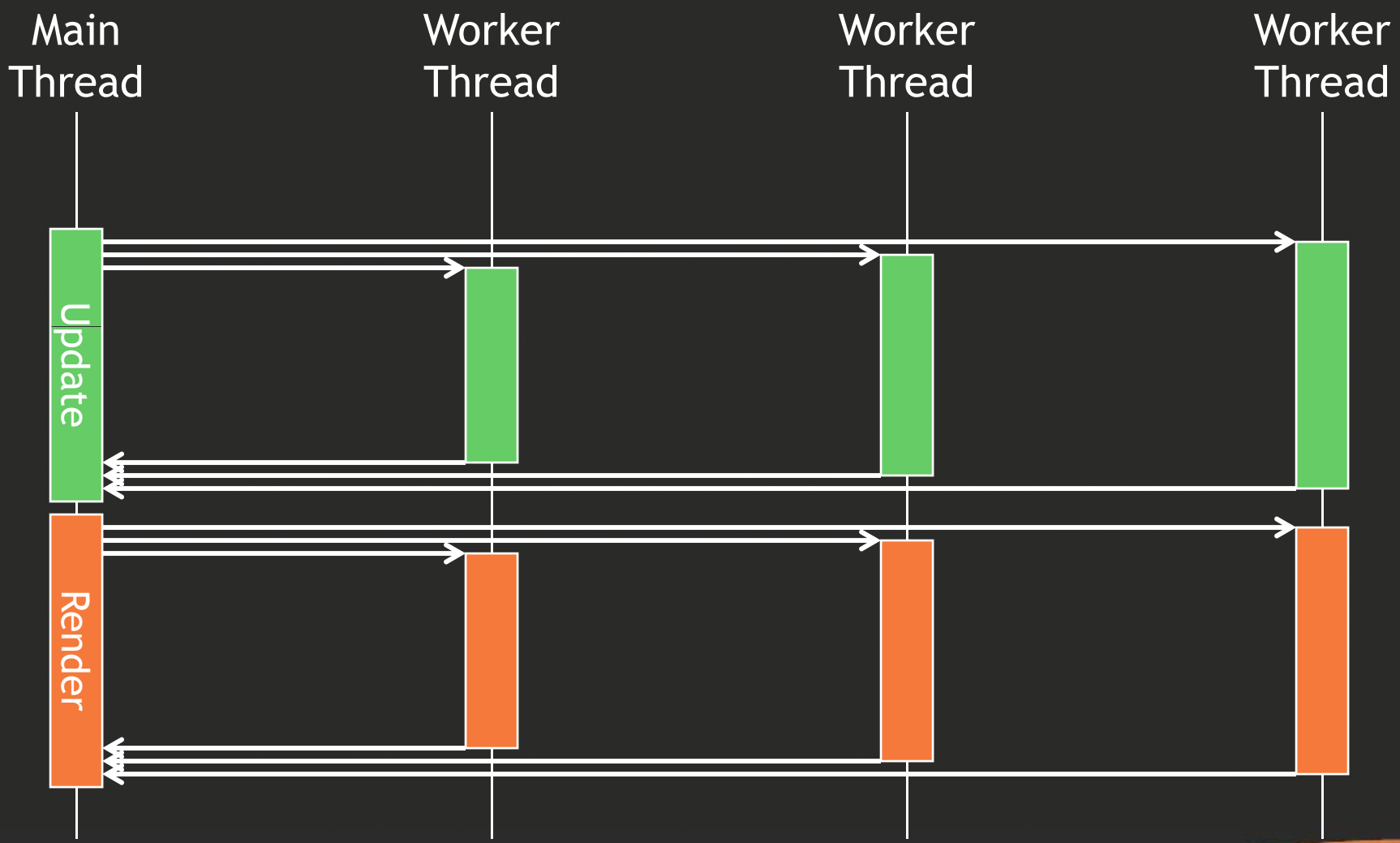
Ideal Scenario

- One thread per hardware thread
- Application manages dispatching work to multiple threads
- Rendering data completely prepared, ready to be sent to single-threaded D3D device
  - Function calls, conditionals, and final matrix multiplies are wasted time on a D3D device thread

Reality

- Update()
  - Seldomly generates coherent data in API specific format.
- Render()
  - Still works between calls to DX API

Going Wide


Main Thread
Worker Thread
Worker Thread
Worker Thread
Update
Render

Command Buffers

- Record calls to D3D
  - Store in a command buffer
  - Can be done concurrently on multiple threads, to multiple command buffers
- Playback D3D commands
  - Efficiently on main thread
  - Exact data for DX API
  - Coherent in memory
- Clean and modular point to integrate to application

Command Buffer Requirements

- Minimal modifications to rendering code
  - Most code uses pointer to D3DDevice
  - Parameters from stack, e.g., D3DRECT
  - Support most of the device API
    - Draw calls, setting state, constants, shaders, textures, stream source, and so on
  - Support effects
- Playback does not modify buffer
- Playback is ideal performance

Command Buffer Allowances

- No support for:
  - Create methods
  - Get methods
  - Miscellaneous other functions that return values
    - QueryInterface, ShowCursor

Command Buffer: Nice to Have

- Buffers played back multiple times
- Optimization of buffers
  - Remove redundant state calls
    - Offload main thread by doing this on recorder threads
  - Reordering of sort independent draw calls

Design: Recording

- Wrap every API call
  - Unsupported calls, return error
  - Supported calls
    - Store enumeration for call into buffer
    - Store parameters into buffer
    - Make copies of non-reference counted objects such as D3DMATRIX, D3DRECT, shader constants, and so on

Design: Playback

- Playback, read from buffer, and
  - select function call pointer from table given token
  - each playback function unpacks parameters buffer

Recording Example

```
virtual HRESULT STDMETHODCALLTYPE DrawPrimitive(
    D3DPRIMITIVETYPE  PrimitiveType,
    UINT              StartVertex,
    UINT              PrimitiveCount)
{
    m_pCommandBuffer->Put(CBD3D_COMMANDS::DrawPrimitive);

    m_pCommandBuffer->Put(PrimitiveType);
    m_pCommandBuffer->Put(StartVertex);
    m_pCommandBuffer->Put(PrimitiveCount);
    return D3D_OK;
}
```

Playback Example

```
void CBPlayer9::DoDrawPrimitive()
{
    D3DPRIMITIVETYPE  arg1;
    m_pCommandBuffer->Get(&arg1);

    UINT arg2;
    m_pCommandBuffer->Get(&arg2);

    UINT arg3;
    m_pCommandBuffer->Get(&arg3);
    if(FAILED(m_pDevice->DrawPrimitive(arg1, arg2, arg3)))
        OutputDebugStringA(__FUNCTION__ " failed in playback\n");
}
```

Effects: Problem

- Effect takes pointer to device at creation
- Effect then creates resources
- At render, effect should use our recorder

- Our recording device cannot create resources

Effects: Solutions

1. Create FX with command buffer device
   - Fails: needs real device for initialization
2. Wrap and record FX calls and play them back
   - Inefficient
3. Give FX EffectStateManager class to redirect calls to command buffer, give it real device for initialization
   - Disables FX use of state blocks
4. Create redirecting device
   - Acts as real device at init, command buffer device at render time

Resource Management

- Multiple threads wish to:
  - Create resources (e.g., background loading)
  - Update resources (e.g., dynamic geometry)
- App must use playback thread only to modify resources
  - App specific logic
    - Deferred creation, double buffering
  - Support in command buffers (next slide)

Resource Management (2)

- Command buffer library could encapsulate details
  - (This is Future Work)
- Gamebryo Volatile Type Buffers
  - D3DUSAGE: WRITEONLY | DYNAMIC
    D3DLOCK: NOOVERWRITE, DISCARD
  - Lock() is stored into command buffer
  - Memory allocated from command buffer, returned from Lock()
  - At playback, true lock is performed
- Gamebryo Mutable Type Buffers:
  - CPU read and infrequent access
  - Backing store required, copied on each Lock()

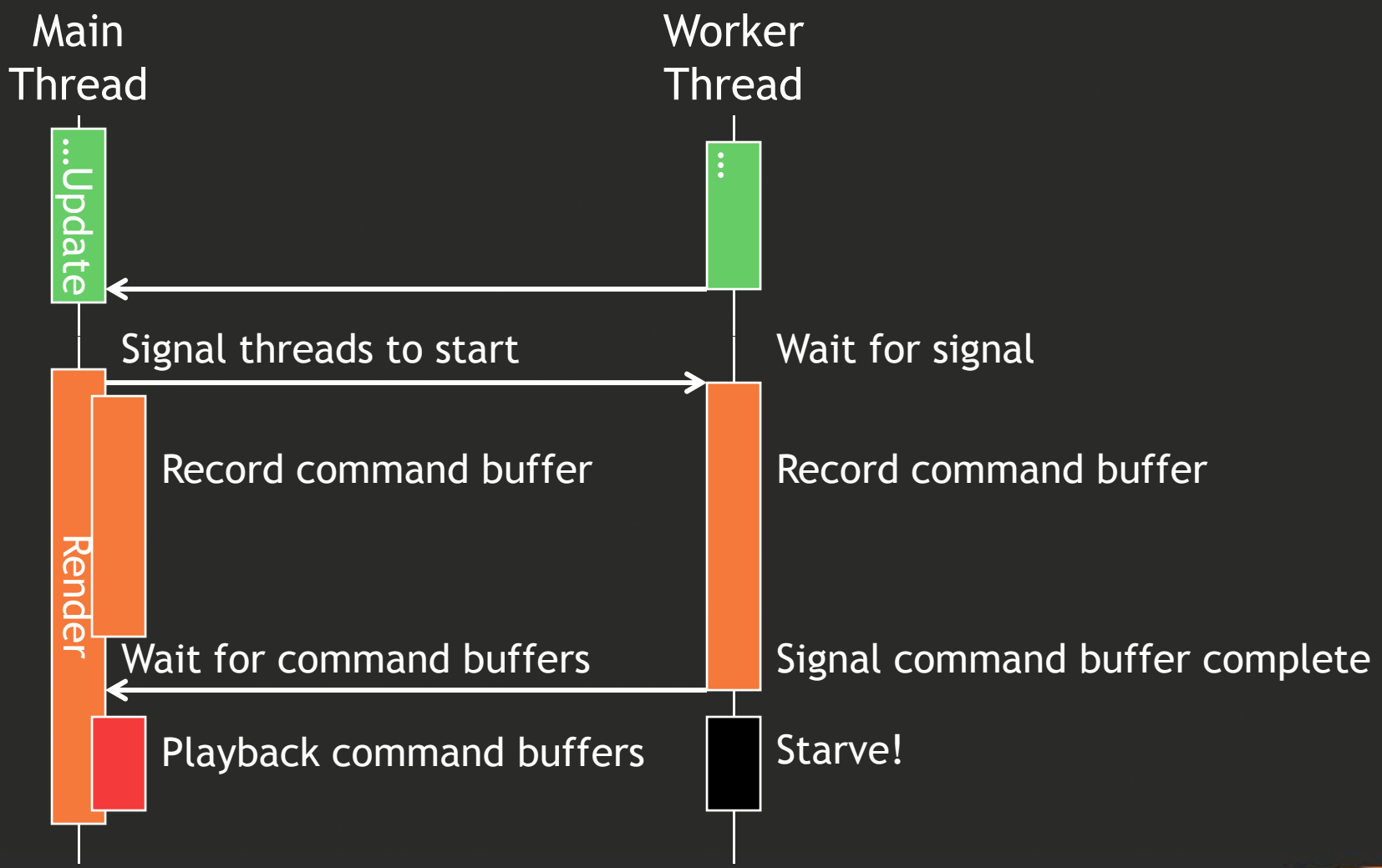
Implementation Considerations

- Ease of changing implementation
  - Macros provide implementation
  - Preprocessor & Beautifier produce debuggable code
  - Many macro permutations required (~40) for different argument count and return type
    - Generated from Excel
  - Function overloading to store non ref counted parameters
    - Everything but shader constants then stored with same function signature.

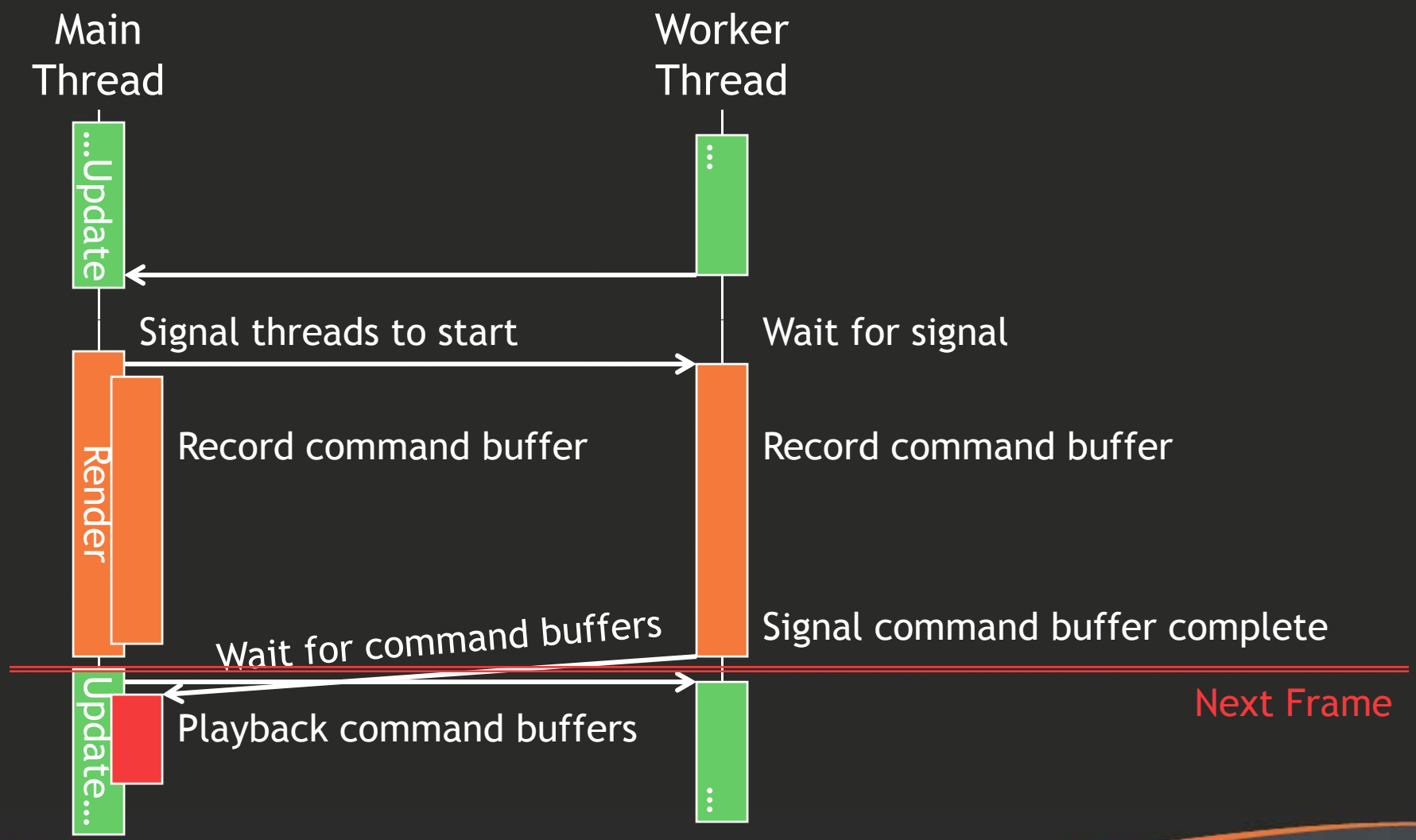
Application Models

- Command buffers can be used in various ways by applications
  - Fork and join
  - Fork and join, frame deferred
  - Work queue
  - …
- Record once, play back several times

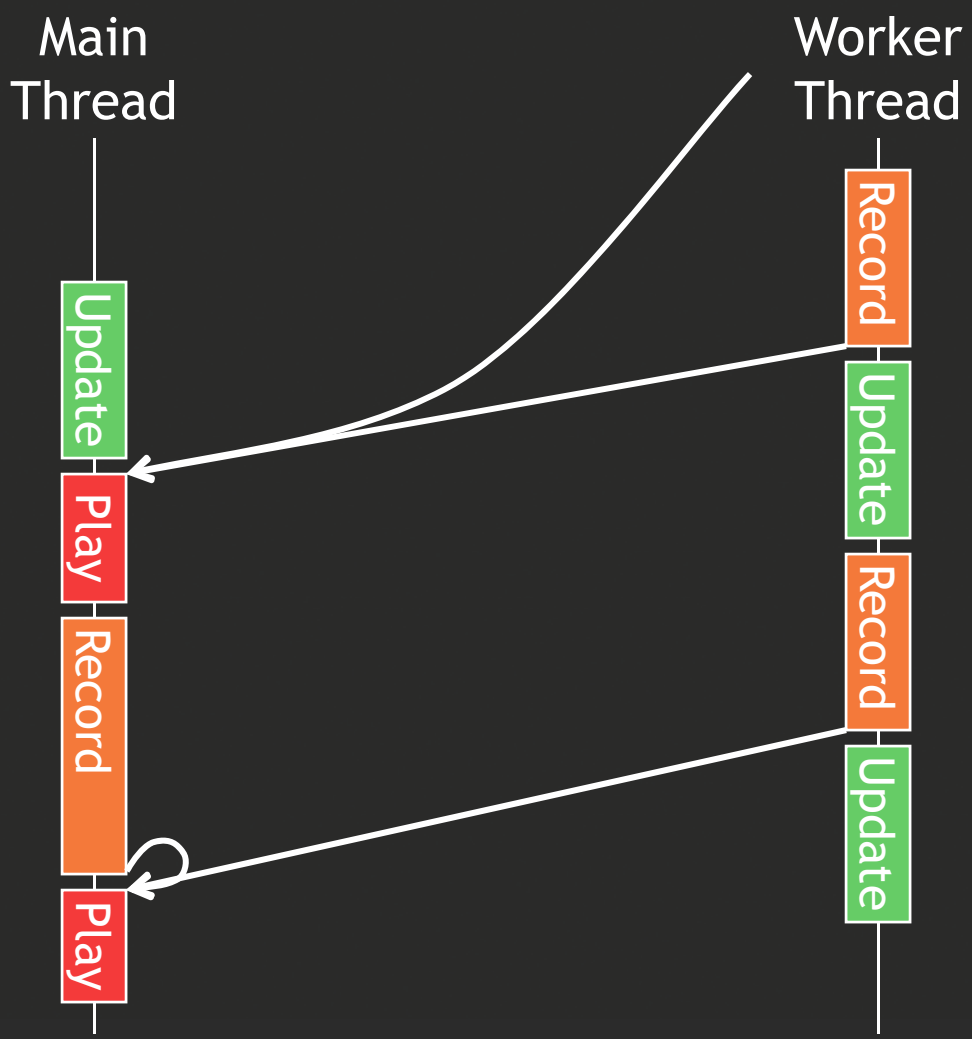
Fork & Join


Main Thread
Worker Thread
...Update
Signal threads to start
Wait for signal
Record command buffer
Record command buffer
Render
Wait for command buffers
Signal command buffer complete
Playback command buffers
Starve!

Fork & Join, Frame Deferred



Work Queue


Main Thread
Worker Thread
Update
Play
Record
Play
Record
Update
Record
Update
Update

Adapting to an Existing Codebase

- Refactor code to take pointer to device that can be changed easily
  - Easy if pointer passed on stack
  - Thread local storage if used from heap
- Add ownership of recording devices, playback class, and pool of command buffers
- Determine application model, and add high-level logic to parcel out rendering work.
- Manage resources over recording and playback

Integration into DX Samples

- Instancing
  - Effects, shader constants
- Textures tutorial
  - Simple, added multithreading
- Stress test
  - Fork and join multithreading, with optional:
    - Frame delay of playback
    - Draw call count
    - CPU and memory access
    - Recorder thread count

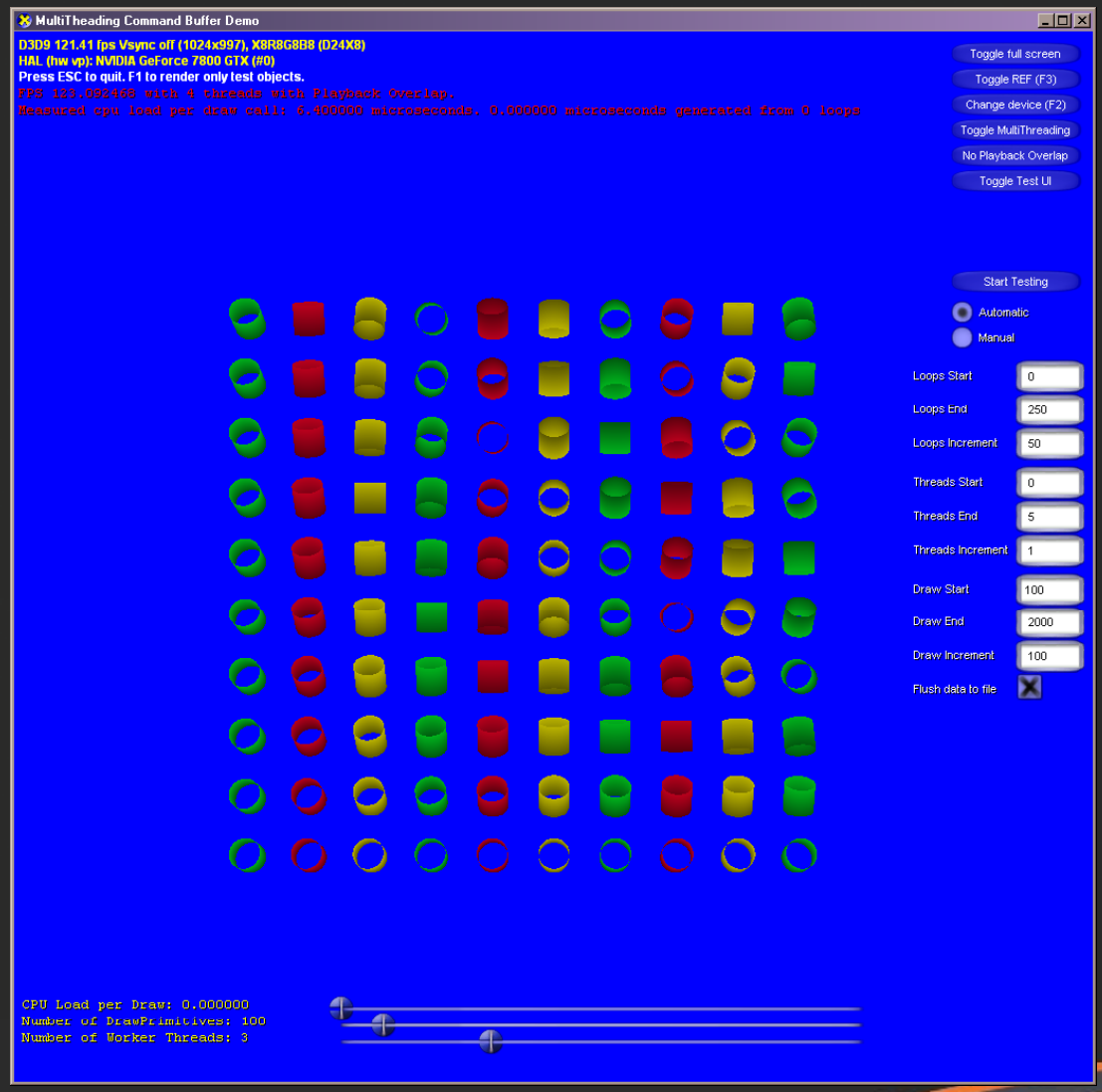
Stress Test Information

- Render call contains:
  - Matrices computed with D3DX calls * 3
  - SetTransform * 3
  - SetRenderState
  - SetTexture
  - SetTextureStageState * 8
  - SetStreamSource
  - SetFVF
  - DrawPrimitive

CPU Busy Loops

- Draw call CPU cost varies in real applications
- Stress test simulates cost with CPU Busy Loops
  - Scattered reads from a large buffer in memory
  - Perform some logic, integer, and floating point operations
- Gamebryo render on DX9: 100-200 µs
  - (on a Pentium 4, 3 GHz, nVidia 7800)
- Stress test can simulate Gamebryo render calls with 0-200 loops.

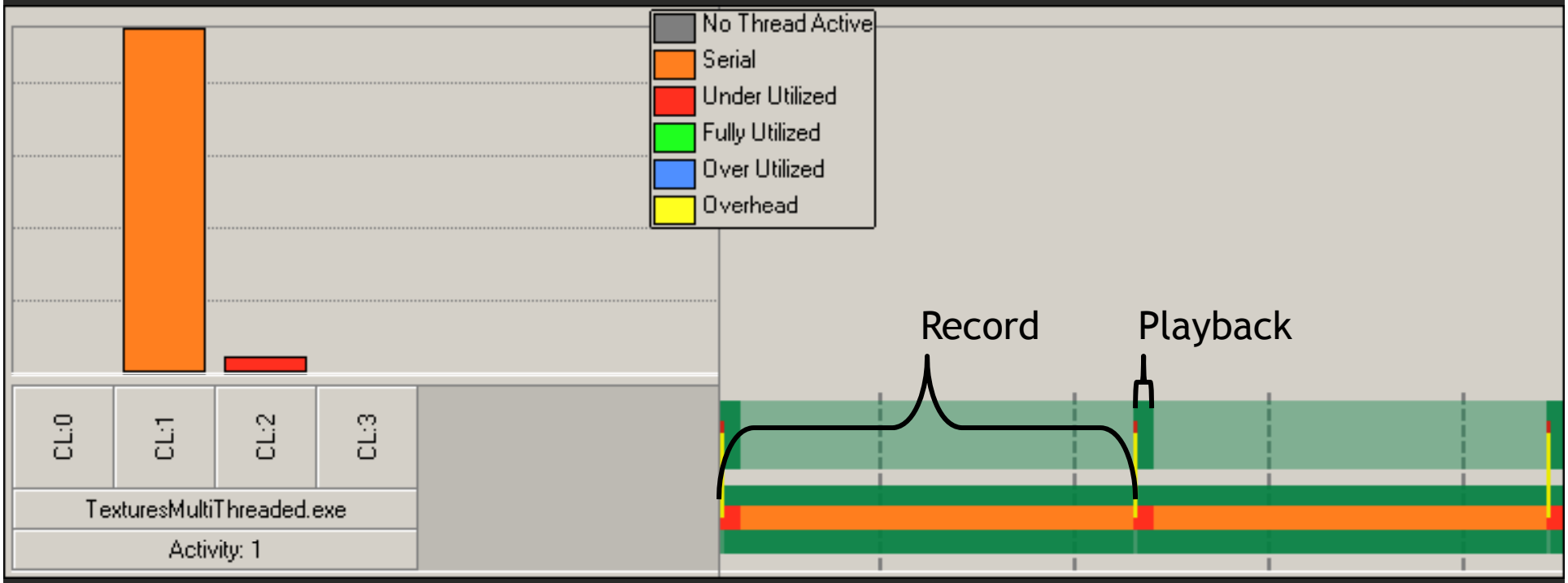
DX Sample Stress Test Demo



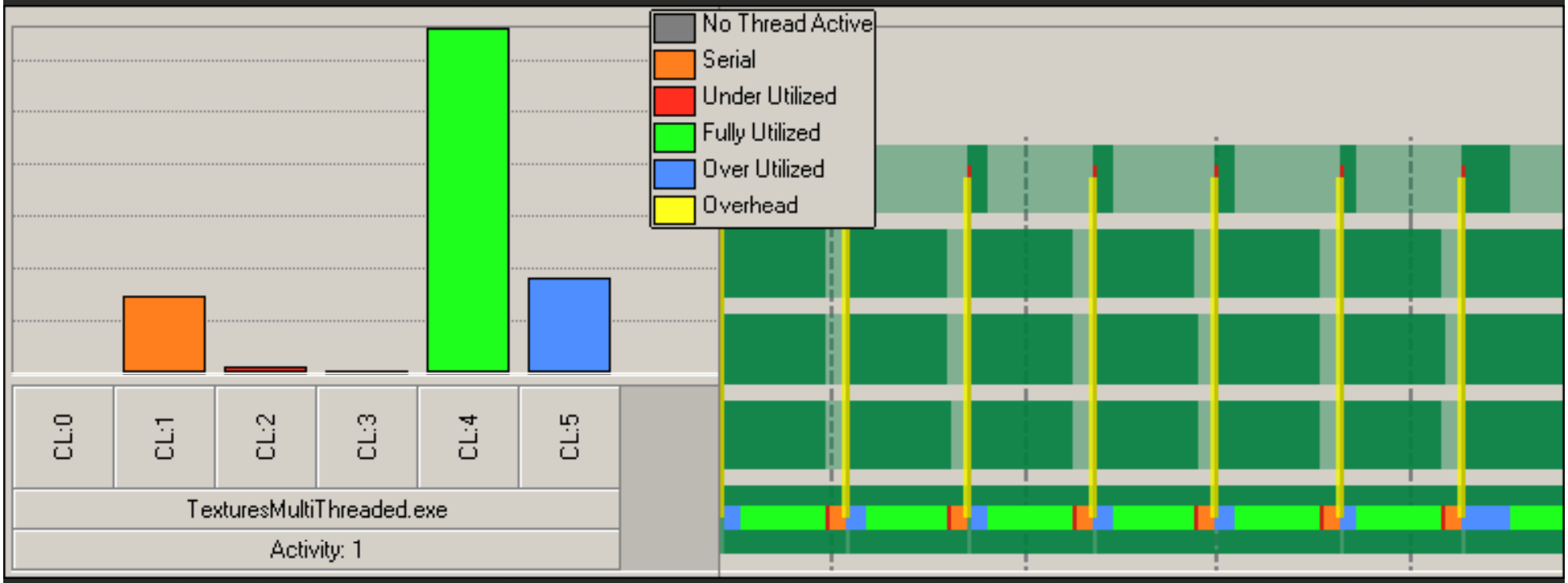
DX Call Cost vs. Recorder Cost

- Render call cost with DirectX device is 13 times as expensive as command buffer recorder
  - DX: 92µs
  - Recorder: 7µs
    - (on a Pentium 4, 3 GHz, nVidia 7800)

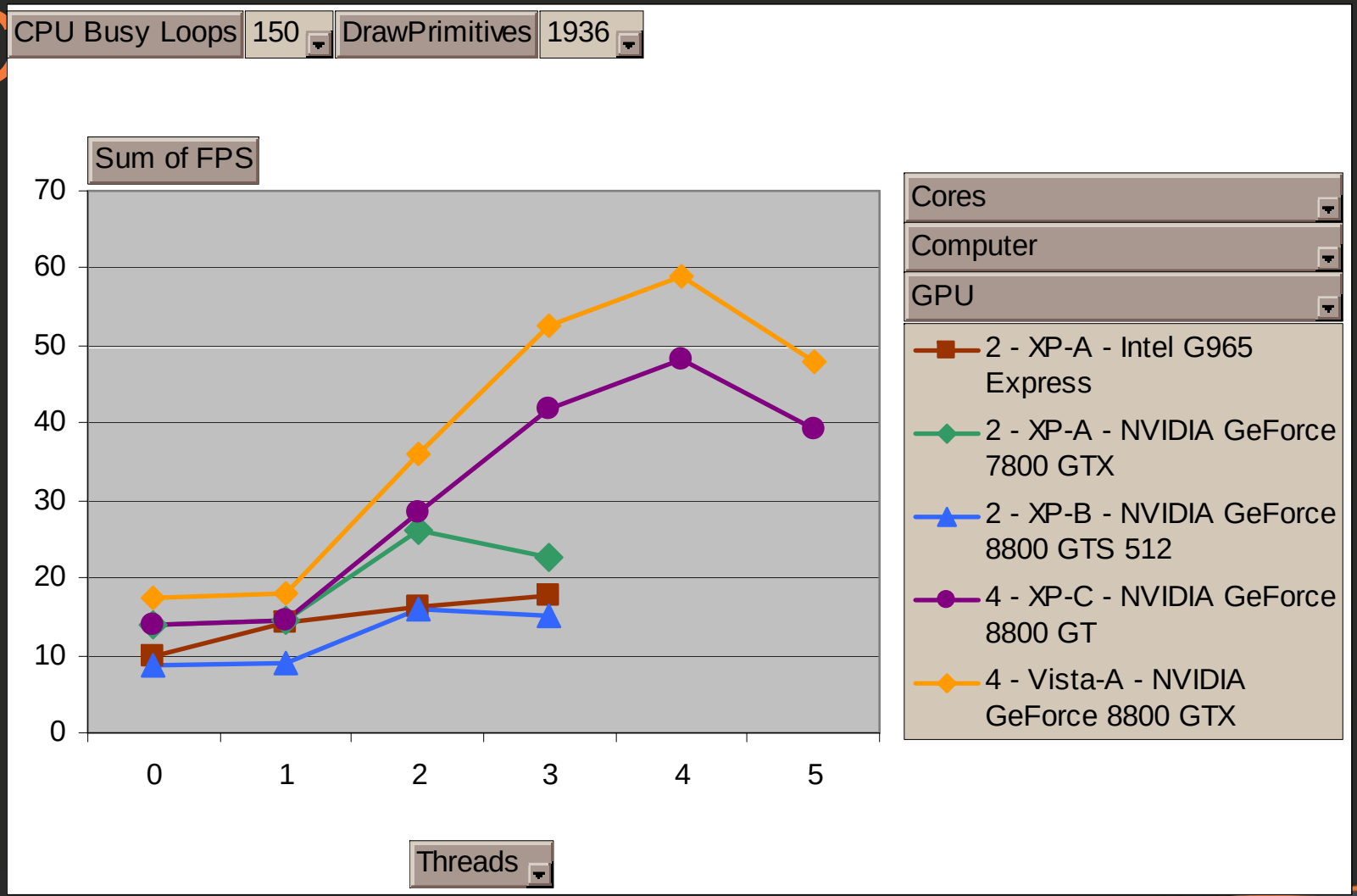
Thread Profiler Quadcore 1 Recorder Thread



Thread Profiler Quadcore 4 Recorder Threads



FPS by Threads and C

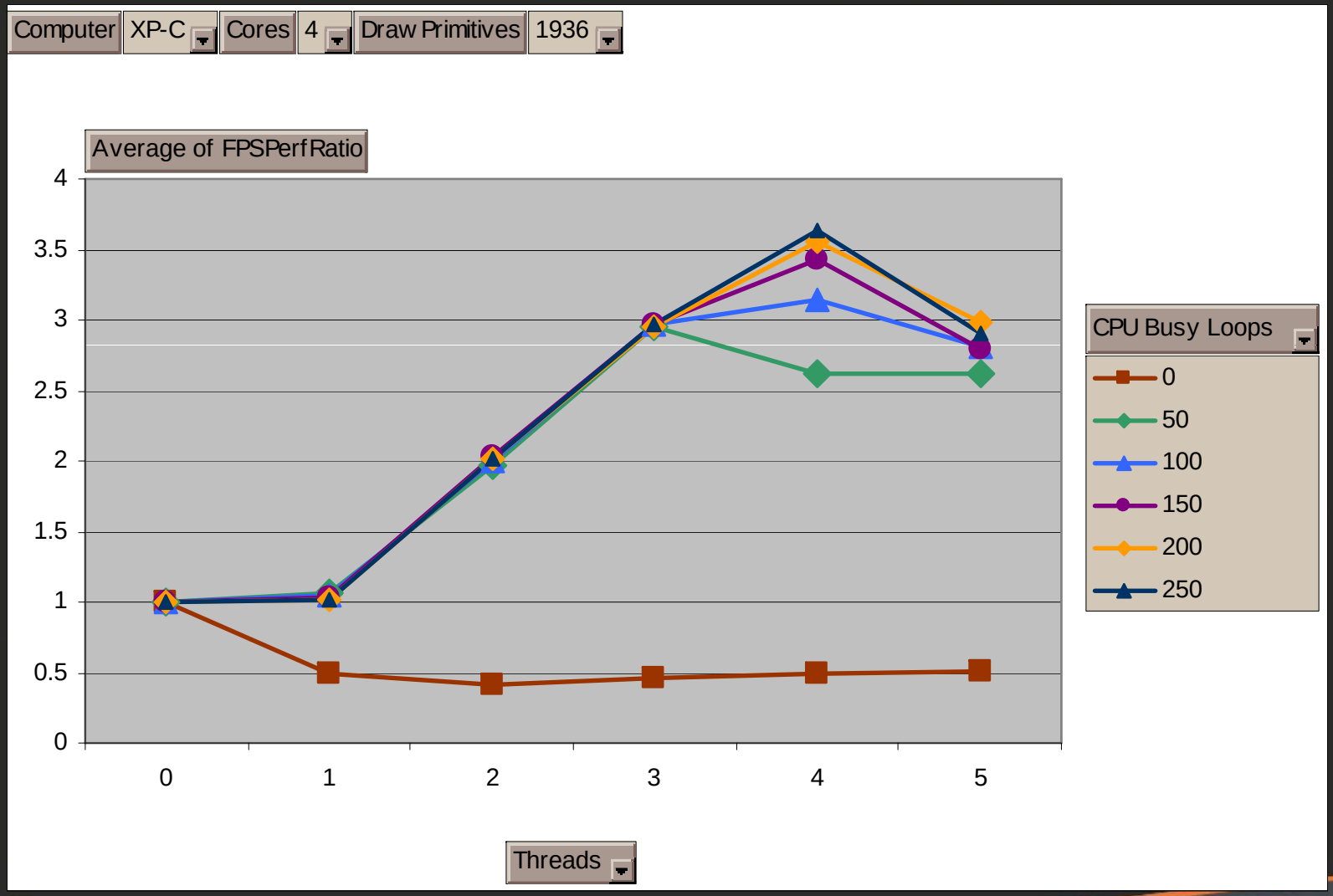


Definition: Performance Ratio

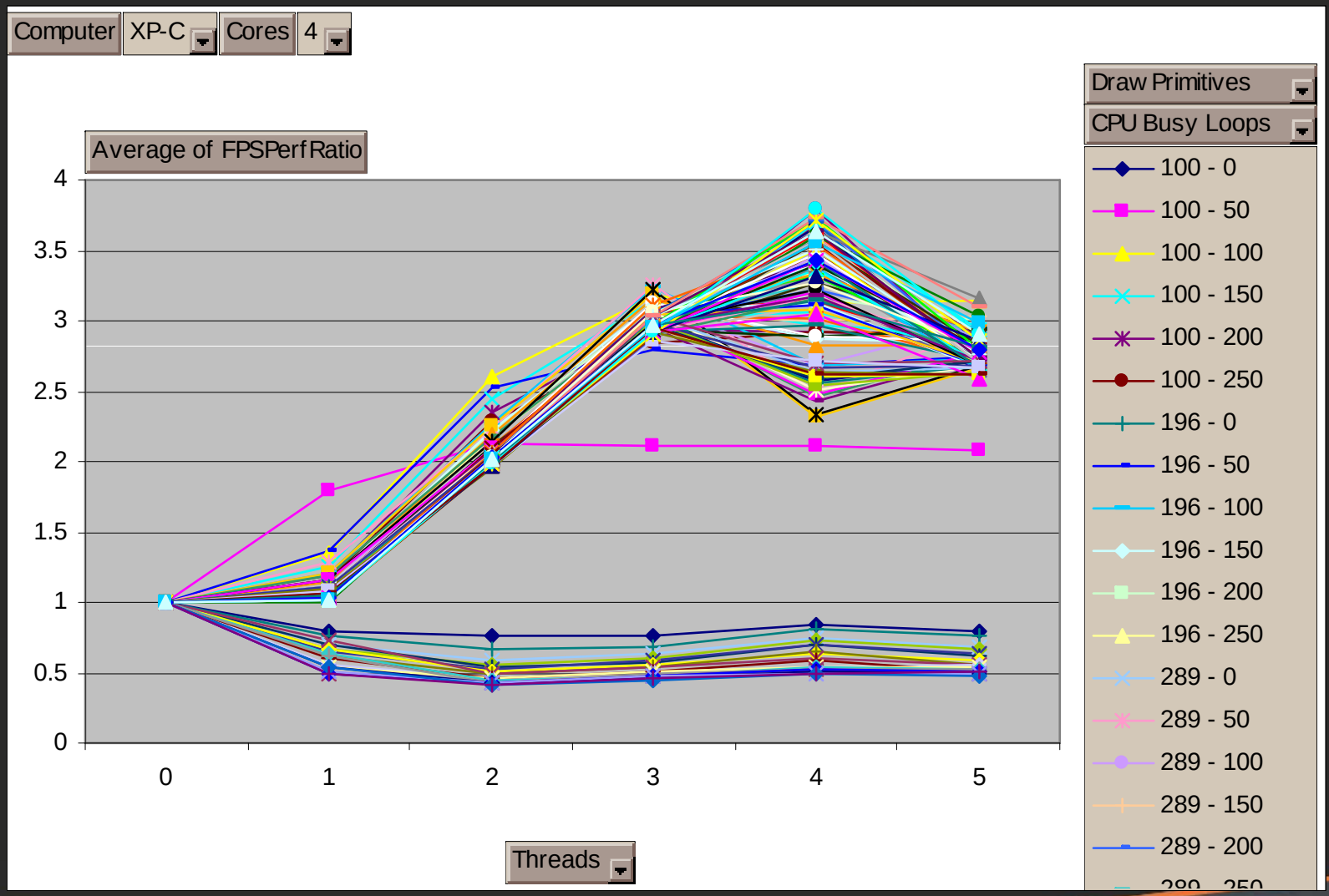
- Charts that follow use
  Performance Ratio = FPS_{test} / $FPS_{baseline}$

- Normalized result
- Useful for comparisons while varying
  - Number of draw calls
  - CPU busy loops

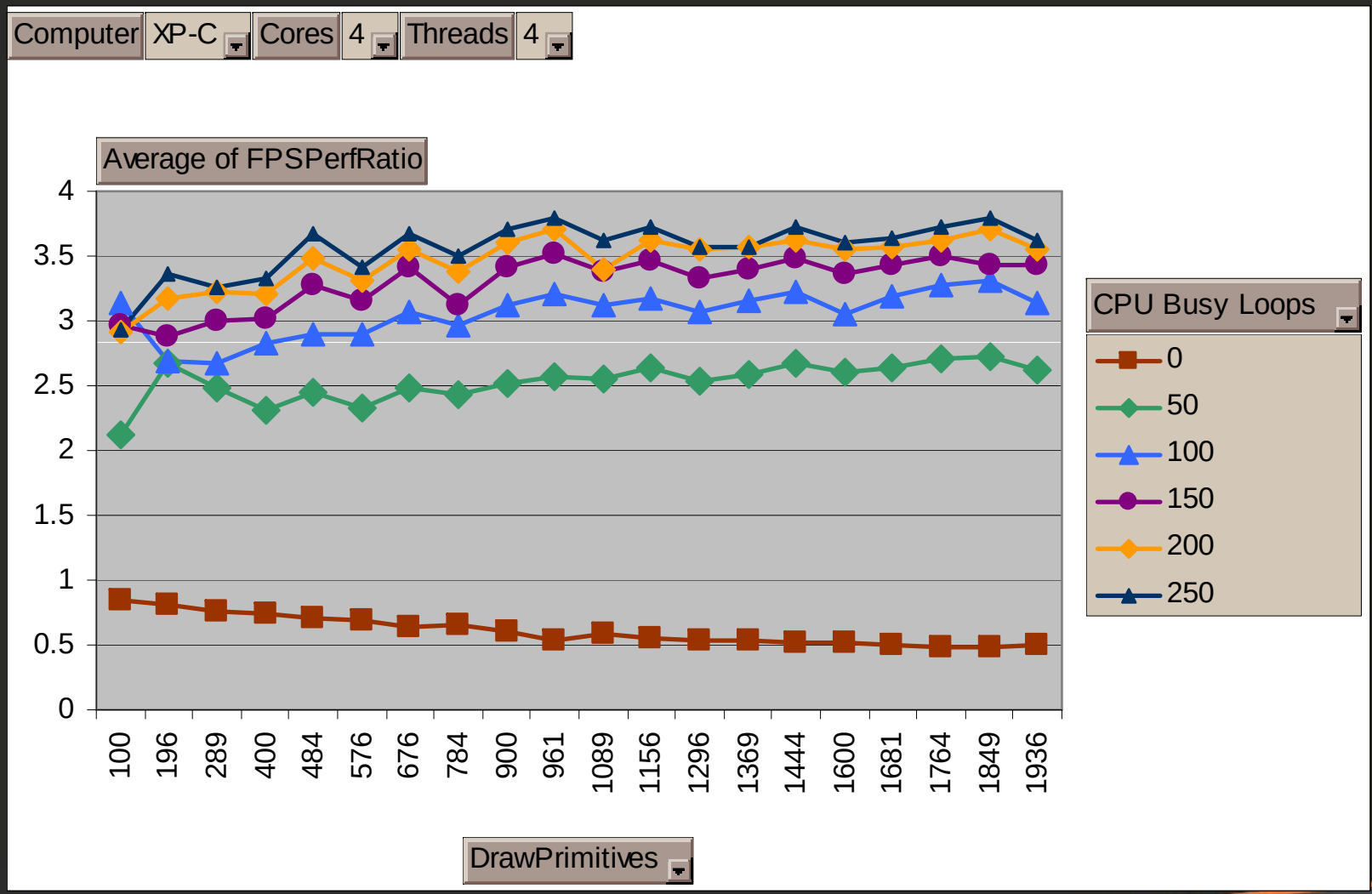
Perf by Threads & Busy Loops



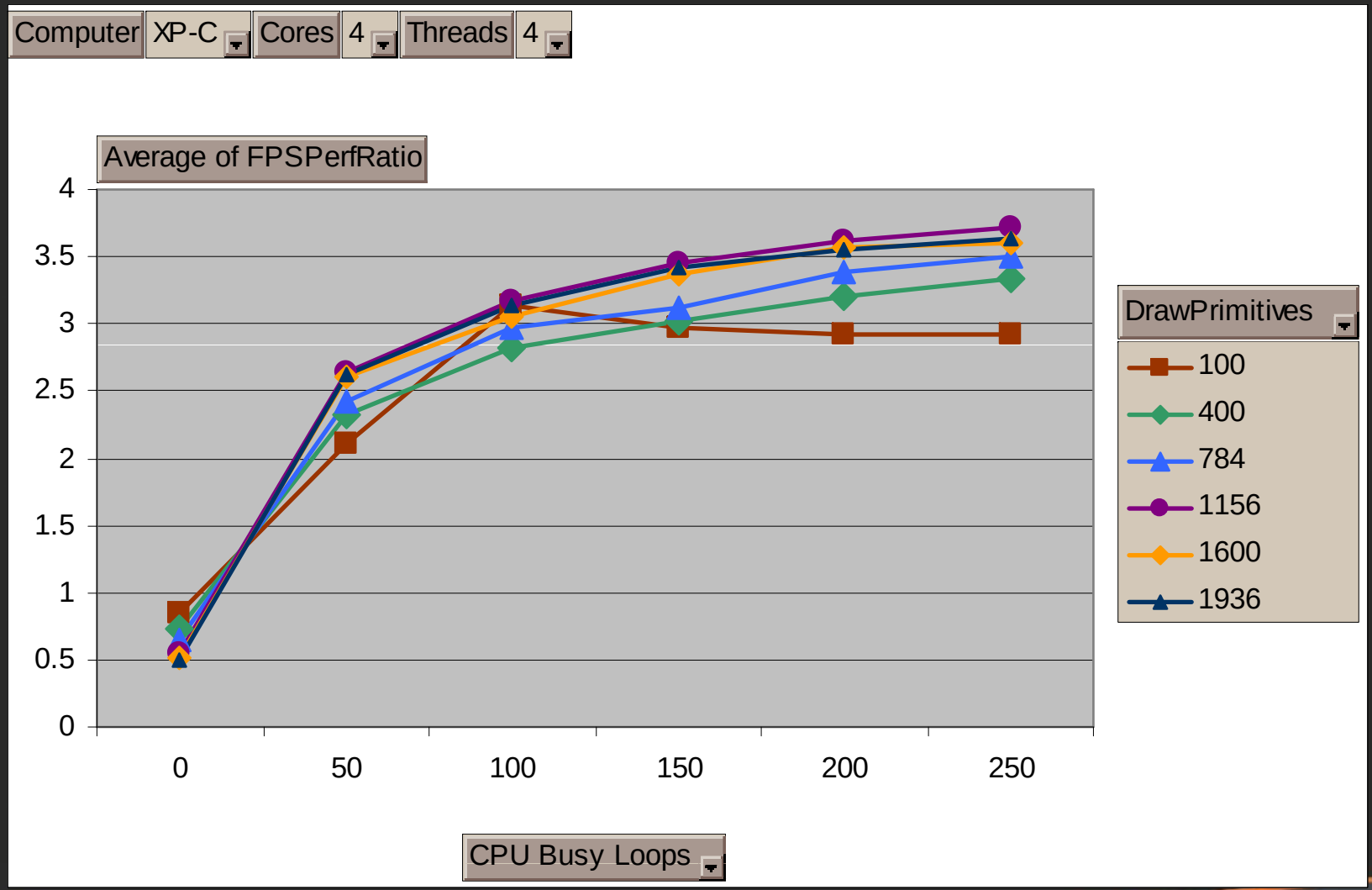
Perf by Threads & Busy Loops



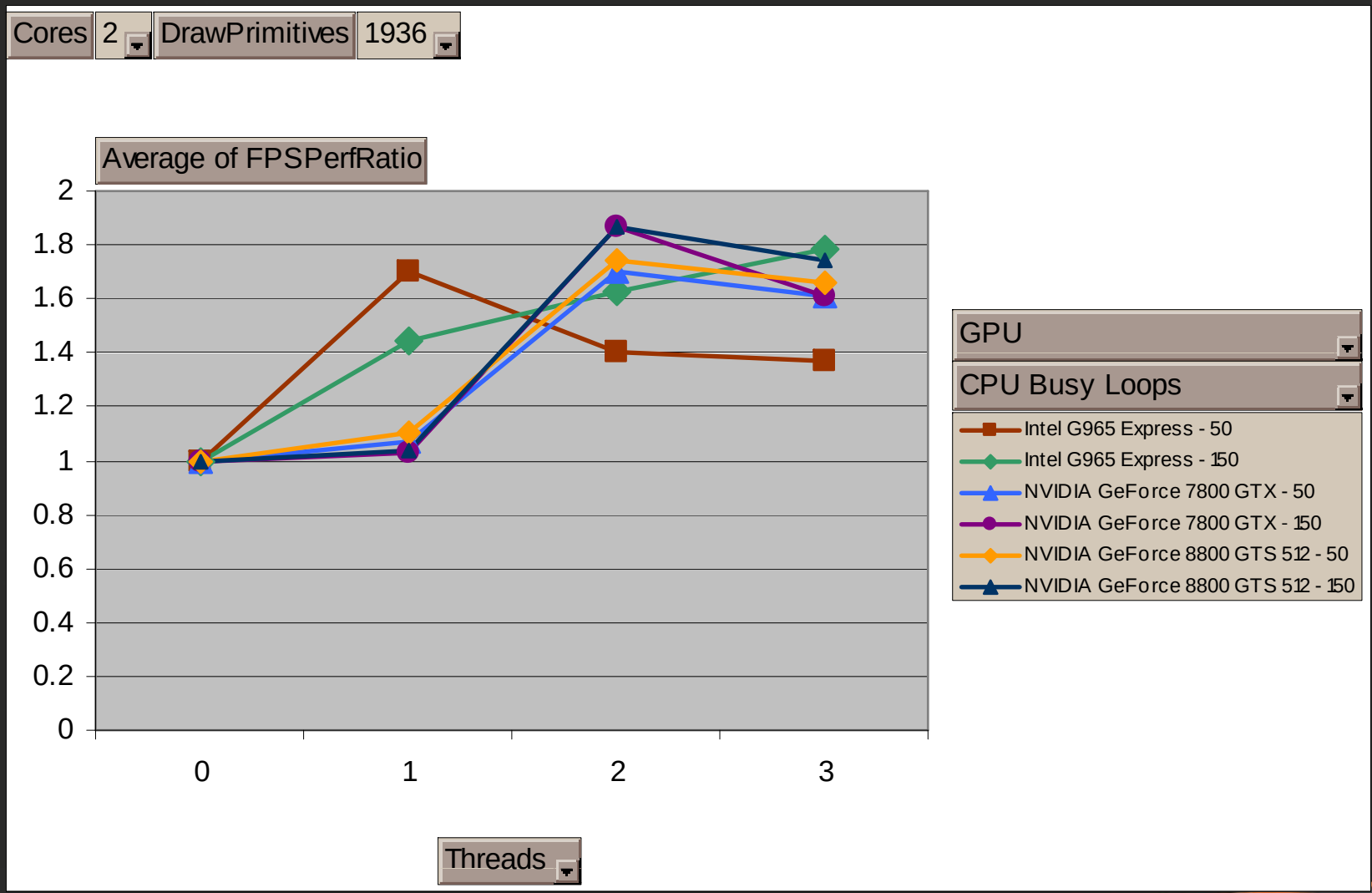
Perf by Draws & Busy Loops



Perf by Busy Loops & Draws



Dual Core Results



Future Work

- Resource management facilitated through command buffer, instead of application logic
- Optimization of command buffers by reordering order independent draw calls
- DirectX10

Open Source Library

- Emergent has open sourced the command buffer library
  - Command buffer serialization
  - Recording device
  - Playback class
  - Redirecting device
  - EffectStateManager
  - DX9 only so far
- http://emergent.net/GameFest2008

Thank You. Questions?

- Presented by:
  - Vincent.Scheib@emergent.net

Co-Developer:

  - Bo.Wilson@emergent.net
- Consultation:
  - Tim.Preston@emergent.net
  - Dan.Amerson@emergent.net
- http://emergent.net/GameFest2008

www.xnagamefest.com

EMERGENT GAME TECHNOLOGIES