Stream Processing with CUDA™
A Case Study Using Gamebryo's Floodgate Technology
Dan Amerson, Technical Director, Emergent Game Technologies
Purpose

• Why am I giving this talk?
• To answer this question:
• Should I use CUDA for my game engine?
Agenda

• Introduction of Concepts
• Prototyping Discussion
• Lessons Learned / Future Work
• Conclusion
What is CUDA: Short Version?

• Compute Unified Device Architecture.
• Hardware and software architecture for harnessing the GPU as a data-parallel computing device.
• Available on recent NVIDIA hardware.
  – GeForce8 Series onward.
  – See CUDA documentation for more details.
What is Floodgate?

• Emergent’s stream processing solution within Gamebryo.
• Designed to abstract parallelization of computationally intensive tasks across diverse hardware.
• Focus on ease-of-use and portability. Write once. Run anywhere.
• Tightly integrated with Gamebryo’s geometric update pipeline.
  – Blend shape morphing.
  – Particle quad generation.
  – Etc.
Stream Processing Basics

- Data is organized into streams.
- A kernel executes on each item in the stream.
- Regular data access patterns enable functional and data decomposition.
- Compatible with a wide range of hardware.

```
NiSPBeginKernelImpl(Sum2Kernel)
{
    NiUInt32 uiBlockCount = kWorkload.GetBlockCount();

    // Get Streams
    NiInputAlign4 *pkInput1 = kWorkload.GetInput<NiInputAlign4>(0);
    NiInputAlign4 *pkInput2 = kWorkload.GetInput<NiInputAlign4>(1);
    NiInputAlign4 *pkOutput = kWorkload.GetOutput<NiInputAlign4>(0);

    // Process data
    for (NiUInt32 uiIndex = 0; uiIndex < uiBlockCount; uiIndex++)
    {
        // out = in1 + in2
        pkOutput[uiIndex].uiValue =
            pkInput1[uiIndex].uiValue + pkInput2[uiIndex].uiValue;
    }
}
NiSPEndKernelImpl(Sum2Kernel)
```
Floodgate and CUDA: Like Peanut Butter and Chocolate

• CUDA’s computational model is slightly more flexible than Floodgate.
  – Floodgate does not currently support scatter and gather.

• Floodgate was designed to subdivide work and target an array of processors.

• The GPU is a very powerful array of stream processors.

• A CUDA backend for Floodgate would allow high-performance execution on the GPU.
Prototyping

• We prototyped this integration using a very simple sample.

• The sample morphed between two geometries.
  – Single-threaded - 115 FPS.
  – Floodgate with 3 worker threads - 180 FPS.

• Original prototype development used CUDA v1.1 on a GeForce 8800GTX.
  – Currently running on v2.0 and 9800GTX.
Phase 1: Naïve Integration

- Each Floodgate execution:
  - Allocated CUDA memory.
  - Uploaded input data.
  - Ran kernel.
  - Retrive results.
  - Upload results to D3D.
  - Deallocate CUDA memory.
Phase 1: Naïve Integration - Results

• Performance scaled negatively. – 50 FPS
• Transferring the data across PCIe bus consumed too much time.
• The gain in computation speed did not offset the transfer time.
Phase 1: Naïve Integration - PCIe Transfer

• Transfer times were measured for a 240KB copy.
• Average transfer was .282ms.
• With naïve transfers, we can’t exceed 148FPS at this rate.
  – We’re seeing about .81 GB/s by this data.
  – This is much lower than peak transfer for PCIe.
Phase 2: Limiting Input Data Transfer

• Iterate on the naïve implementation.
  – Allocate input and output data in CUDA memory at application init.
  – Upload input data once.
  – Retrieve results and upload to D3D each frame.
Phase 2: Limiting Input Data Transfer - Results

• Performance improves dramatically.
  – 145 FPS

• Performance exceeds single threaded execution.

• Does not exceed the multithreaded Floodgate execution.
  – Single-threaded engines or games can benefit from this level of CUDA utilization.
Phase 3: Using D3D Input Mapping

• Since results were positions in a D3D VB, we can write directly to it.
  – cudaD3D9MapVertexBuffer(void**, IDirect3DVertexBuffer9*)
  – More robust APIs exist in CUDA v2.0.

• With this mapping in place, no per-frame memory transfers need occur.
Phase 3: Using D3D Input Mapping - Results

• Fastest performance of any configuration.
  – 400 FPS

• Exceeds performance of a quad-core PC with 3 worker threads.
Lessons Learned - The Hard Way

• A UPS is critical!
Lessons Learned / Future Work

• Seamless build system integration.
• Build CUDA dependency scheduling into the execution engine directly.
• Heuristics for task decomposition.
• Automatic detection of good candidates for GPU execution.
Build System Integration

• The prototype used custom Floodgate subclasses to invoke CUDA execution.

• Ideally, we’d like to have a more seamless build integration.
  – Cross-compile for CUDA and CPU execution.
  – Hide this complexity from developers on other platforms.
Build System Integration - Floodgate

• We currently wrap kernels in macros to hide per-platform differences.
  – NiSPBeginKernelImpl(NiMorphKernel)
  – NiSPEndKernelImpl(NiMorphKernel)
Build System Integration - CUDA

• Macros for PC would need to create:
  – CUDA invokable __global__ function.
  – Create the equivalent __host__ function.
  – Provide the entry-point used by the system at runtime.

• True cross-compilation would also need to define or typedef types used by nvcc.
  – Necessary for code compiling for other targets.
Beyond Simple Tasks: Workflows

- Simple stream processing is not always sufficient for games.
- With Floodgate, streams can inputs and outputs creating a dependency.
- Schedule entire workflows from dependencies.
Automatically Managing Dependencies

• CUDA has two primitives for synchronization.
  – cudaStream_t
  – cudaEvent_t

• Floodgate mechanisms for dependency management fit nicely.
  – Uses critical path method to break tasks into stages.

• Each stage synchronizes on a stream or event.
  – cudaStreamSynchronize
  – cudaEventSynchronize
Task Subdivision for Diverse Hardware

• One of the primary goals of Floodgate is portability.

• System automatically decomposes tasks based on the hardware.
  – Must fit into the SPU local store for Cell/PS3.
  – Optimize for data prefetching on Xbox 360.

• A CUDA integration would want to optimize the block and grid sizes automatically.
Brief Aside on CUDA Occupancy Optimization

- Thread count per block should be a multiple of 32.
  - Ideally, at least 64.
- The number of blocks in the grid should be multiple of the number of multiprocessors.
  - Preferably, more than 1 block per multiprocessor.
  - Number of multiprocessors varies by card.
  - Can be queried via cudaGetDeviceProperties.
Brief Aside on CUDA Occupancy Optimization - Cont.

• There are a number of other factors.
  – Number of registers used per thread.
  – Amount of shared memory used per thread.
  – Etc.

• We recommend that you look at the CUDA documentation and occupancy calculator spreadsheet for full information.
Automatic Task Subdivision for CUDA

• A stream processing solution like Floodgate can supply a base heuristic for task decomposition.
  – Balance thread counts vs. block counts.
  – Thread count ideally >= 192.
  – Block count multiple of multiprocessor counts.
  – Potentially feed information from .cubin into build system.

• Hand-tuned subdivisions will likely outperform, but it is unrealistic to expect all developers to be experts.

• Streams with item counts that aren’t a multiple of the thread count will need special handling.
Detecting Good Candidates for the GPU

• Not all game tasks are well-suited for CUDA and GPU execution.

• Automatable Criteria
  – Are the data streams GPU resident?
  – Are any dependent tasks also suited for CUDA?
  – Does this change if the GPU supports device overlap?

• Manual Criteria
  – Are you performing enough computation per memory access?
Conclusions

• CUDA provides a very powerful computing paradigm suitable to stream processing.
• Memory transfer overhead via PCIe can dominate execution time for some tasks.
• Suitable tasks are significantly faster via CUDA than a quad-core PC.
• Identifying and handling such tasks should be partially automatable.
To Answer the Question...

• Should I use CUDA for my game engine?
• You should definitely start now, but...
• ...widespread use is probably further out.
Thank you

• Questions are welcome.
• Thank you to:
  – Randy Fernando and NVIDIA developer relations.
  – Vincent Scheib
• amerson@emergent.net