Parallel Programming Models for Graphics

Tim Foley
Stanford/Intel
13th April, 2010
Problem Statement

• Building a high-performance application
  • Game engine (physics, graphics, audio, ...)
  • Software renderer (ray tracing, micropolygon, ...)

• Need to select tools, techniques
  • Productivity
  • Performance
  • Portability
Scope

• Games as representative application
  • High performance, visual quality
  • Use multicore, throughput, heterogeneous HW
  • Visibility, illumination, physics, simulation

• Not covering every possible approach
  • Explicit threads/locks
  • Message-passing/actors/CSP
  • Transactions
What goes into a game frame?
Rendering computation graph for *Battlefield: Bad Company* provided by DICE.
A modern game is a mix of...

Data-parallel algorithms
A modern game is a mix of...

Task-parallel algorithms and coordination
A modern game is a mix of...

Standard and extended graphics pipelines

- Input Assembly
- Vertex Shading
- Primitive Setup
- Geometry Shading
- Rasterization
- Pixel Shading
- Output Merging

Pipeline Flow
Data-Parallel  Task-Parallel  Pipeline-Parallel
Structure of this talk

• Start with a hypothetical workload

• Decompose it to expose parallelism

• Apply each of our three models, noting:
  • Key idea
  • Mental model
  • Applicability
Workload
Typical Game Workload

• Subsystems given % of overall time “budget”
• Input, Miscellaneous: 5%
• Physics: 30%
• AI, Game Logic: 10%
• Graphics: 50%
• Audio: 5%

<table>
<thead>
<tr>
<th>I</th>
<th>Physics</th>
<th>AI</th>
<th>Graphics</th>
</tr>
</thead>
</table>

• GPU Workload:

“Rendering”
Parallelism Anti-Pattern #1

- Assign each subsystems to a SW thread

  thread 0  
  
  thread 1  
  Physics  
  Physics  
  thread 2  
  AI  
  AI  
  thread 3  
  Graphics  
  Graphics  

- Problems
  - Communication/synchronization
  - Load imbalance
  - Preemption leads to thrashing

- Don’t do this
Parallelism Anti-Pattern #2

• Group subsystems into HW threads

  thread 0: I Physics AI A I Physics AI A
  thread 1: Graphics Graphics

frame N

• Problems
  • Communication/synchronization
  • Load imbalance
  • Poor scalability (4, 8, ... HW threads)

• Don’t do this either
Work granularity

- **Physics**
  - Forward integration
  - Collision detection (coarse, fine)
  - Contact group (create, solve)

- **Graphics**
  - Frustum culling
  - Animation
  - Command buffer generation
Dependencies

- Suppose ordering constraints are known

- Visualize as a graph
Distribute work to threads

• Respecting the dependencies

thread 0: I, P, P, P, G, G, G

thread 1: P, AI, A, G, G

thread 2: P, G, G, G

thread 3: G, G

• This is the crux of task-parallel programming
Task Parallelism
Key Idea

• Achieve scalability for heterogeneous or irregular work, by expressing dependencies directly

• Lightweight cooperative scheduling
Mental Model

• Think of task as asynchronous function call
  • “Do F() at some point in the future...”
  • Optionally “… after G() is done”

• Can be implemented in HW or SW

• Usually cooperative, not preemptive
  • Task runs to completion – no blocking
  • Can create new tasks, but not wait
Implementation

• One worker thread per HW thread context
  • Cooperative scheduling – no preemption
  • Pull tasks from queue of work requests

• Work stealing
  • Each thread maintains its own work queue
  • When empty, steal work from random “victim”
  • Minimizes communication/contention

• Can implement on a wide variety of platforms
  • Atomic operations
  • Memory coherence model
Task Granularity

- Coarse-grained tasks easy to identify

- Can schedule poorly
  - Coarse-grained dependencies
  - “Bubble” waiting for predecessor to clear
Task Granularity

- Fine-grained tasks pack well
  - Tune average task size to strike a balance

- More scheduling overhead
  - Tune average task size to strike a balance
Tasks and Software Threads

- Task sounds a lot like OS software thread
  - Create with function to execute + arguments
  - Waits on a queue to be scheduled to a HW thread
  - May trigger events (e.g. other tasks) on completion

- Cooperative vs. preemptive scheduling
  - Task “owns” HW thread for its duration
  - Preemption can’t interrupt time-sensitive task

- Task programs more restricted
  - No blocking operations, synchronization inside task
  - Task “join” usually restricted and lightweight
Applicability

- Task system can handle almost any workload
  - Irregular, dynamic, unpredictable work
  - Viable model for top-level program

- Tends to ignore structure, locality of workload
  - Batches of identical tasks
  - Tasks sharing access to one data structure
  - Producer-consumer locality
Examples / Further Reading

- Cilk [MIT -> Cilk Arts -> Intel]
- OpenCL [Khronos]
- ConcRT [Microsoft]
- Grand Central Dispatch [Apple – C/ObjC]
- TBB [Intel]
- Task Parallel Library (TPL) [Microsoft – C#]
- SPURS [Sony – PS3]
- Custom
  - ex. “Saints Row Multiprocessing Architecture”
    - [Randal Turner, Volition Inc. – GDC 2007]
Data Parallelism
Data Parallelism

• Test every model in scene against frustum

• Want to exploit structure
  • All cull tasks run the same program
  • Data is in contiguous array
  • Dependencies are trivial and statically known
Key Ideas

• Run a single kernel over many elements

• Per-element computations are independent

• Exploit throughput architecture
  • Amortize per-element costs with SIMD/SIMT
  • Hide memory latency with lightweight threads
  • Details in previous lecture
Mental Model

• Execute N independent work items
  • aka: elements, fragments, threads

• All work items run the same program: kernel

• Work item uses data determined by $0 \leq i < N$
  • $[0,N)$ is the domain of computation
Domain of Computation

• Determines number and “shape” of work items

• Often based on input/output data structure
  • Not required – domain and data may be decoupled

• Many domain “shapes” possible
  • Regular
  • Nested
  • Irregular
Simple Data Parallelism

- **Data structure**
  - Regular array

- **Kernel**

- **Domain of computation**
  - 1-D interval \([0,N)\)

Program:

```c
void k(int i) {
    B[i] += A[i];
}
```

Computation:

```
k(0) K(1) K(2) K(3) K(4) K(5)
```
Simple Data Parallelism

- **Data structure**
  - N-D array

- **Kernel**

- **Domain of computation**
  - N-D interval

```c
void k(int i, int j) {
    B[i][j] += A[i][j];
}
```
Shapes need not match

• Data structure
  • N-D array
  • 1-D array

• Kernel

• Domain of computation
  • 1-D interval

```
void k(int i) {
    for(int j = 0; j < M; j++)
        B[i] += A[i][j];
}
```
Many possible syntaxes

<table>
<thead>
<tr>
<th>Kernel Language</th>
<th>Parallel “Loop”</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kernel void k(</code></td>
<td><code>par_for(int i = 0; i &lt; N; i++)</code></td>
</tr>
<tr>
<td><code>float* C) {</code></td>
<td></td>
</tr>
<tr>
<td><code>C[id] = A[id] + B[id];</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td></td>
</tr>
<tr>
<td><code>k&lt;N&gt;(A, B, C);</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Operations</th>
<th>Parallel Functional Map</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Stream&lt;float&gt; A, B, C;</code></td>
<td><code>fun k(a, b) = a + b</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>C = A + B;</code></td>
<td><code>C = par_map(k, A, B)</code></td>
</tr>
</tbody>
</table>
Advanced Data Parallelism

• Hierarchical domains
  • Allow work items to communicate
  • Useful for sums, sorts, scans

• Irregular domains
  • Nested or “ragged” data structures
“Flat” Domains

• Kernel temporaries, scratch data are:
  • Private: inaccessible to other work items
  • Transient: inaccessible after work item completes

• Flat domains expose work-item locality

• Optimization
  • Allocate to core local resources (registers, scratchpad)
  • Low-latency access
  • No coherence traffic for read/write
Communication

• Need to communicate intermediate results
  • Each work item computed value, now want sum

• Write to main memory, launch a new kernel?
  • Don’t exploit locality, rest of memory hierarchy

• Employ a hierarchy of domains
Hierarchical Domains

- Domain composed of smaller domains
  - Each level has its own scratch memory
  - Might be tied to memory hierarchy
    - ex. Registers, L1$, L2$, DRAM

- Work item can access
  - Kernel parameters
  - Own scratch memory
  - Scratch memory of hierarchy ancestors
Hierarchical Domains

- Communicate through parent item scratch
  - ex. Each element computes value “a”
  - Add local value into shared “sum”

- Data races now possible
  - Atomic operations
  - Synchronization barriers, fences

- Also possible for global memory...

- Sacrifice “pure” data-parallel
  - Work items no longer truly independent
“Pure” Data Parallel

- Hierarchical of data-parallel dispatches
  - No communication, synchronization between elements
Current APIs, languages compromise

- Hierarchy, but no internal fork/join
  - Implement similar semantics with barrier
Irregular Domains

• “Ragged array” data structure
  • N-D array- / grid-of-lists

• Used for
  • Bucketing: particles in a cell
  • Collision: potential collides
  • ...

\[
\{{\{A0,A1\}, \{B0,B1,B2\}, {}\}, \{D0,D1\}, \{E0\}, \{F0\}\}
\]
Irregular Domains

- Must choose in-memory representation
  - Pointer per bucket?

- Performance

- Required operations
  - Apply kernel to each bucket?
  - Apply kernel to each element?
A simple representation

Logical

Physical

Count:

Offset:

Storage:
Apply to each element

Count:

Offset:

Storage:

A0  A1  B0  B1  B2  D0  D1  E0  F0
Apply to each bin

Count:

Offset:

Storage:
Irregular/Nested Data Parallelism

• Key insight: represent irregular structure as flat storage and indexing arrays
  • Many other representations possible

• Allows efficient regular data-parallel implementation of some irregular algorithms
  • Many examples in the literature
Parallelism Anti-Pattern #3

• You have N elements to process, P processors

• Given N/P work to each processor?
  • Lots of up-front communication
  • Poor load balancing
  • Small N? Large P?
    • Overhead of distribution may be greater than actual work

• Don’t do this
  • Rule of thumb: strive to be oblivious of P

• Divide into batches of approximately K elements
  • Pick K such that overhead is small
Examples / Further Reading

- NESL [Bleloch]
- BrookGPU [Buck et al. Stanford]
- Sh [McCool et al. Waterloo], RapidMind
- Glift [Lefohn et al. UC Davis]
- CUDA [NVIDIA]
- OpenCL [Khronos]
- Data-Parallel Haskell [Chakravarty et al. UNSW]
Pipeline Parallelism
Key Idea

- Algorithm is ordered sequence of stages
  - Each stage emits zero or more items for next

- Buffer items between stages
  - Increase throughput by running stages in parallel
  - Re-balance load between stages

- Exploit producer/consumer locality
  - On-chip FIFOs
  - Efficient bus between cores
  - ...

GPU Rendering Pipeline (DirectX 10)

- Pipeline of
  - Fixed-function stages
  - Programmable stages
    - Data-parallel kernels
- Stages run in parallel
  - Even on “unified” cores
- Queues between stages
  - Often in HW
Why pipelines?

- **Variable-rate amplification**
  - Rasterization: 1 triangle in, 0-N fragments out
  - Ray tracer: 1 hit in, 0-N secondary/shadow rays out
  - Load imbalance
Pipelines can cope with imbalance

• Re-balance load between stages
  • Buffer up results for next stage

• Optimize for locality
  • Keep buffers small, local
  • On-chip FIFOs, busses, caches, scratchpads

• Prioritization
  • Given choice between buffers, which to service?
  • Favor earliest stages
    • Operate on large batches of work – minimize overhead
  • Favor latest stages
    • Drain buffers as fast as possible – minimize storage
User-defined pipelines

• Standard practice for console developers
  • Custom Cell/RSX graphics pipelines on PS3

• Pipeline-oriented programming models research
  • GRAMPS [Sugerman et al. 2009]

• Challenges
  • Bounding intermediate storage
  • Scheduling algorithms
Examples / Further Reading

- GRAMPS [Sugerman et al. Stanford]
- Playstation EDGE [Sony]
Composition / Comparison
Blurring the Lines

• Task-parallel to assign work to cores/threads
  • Then data-parallel for SIMD, latency hiding
  • This is effectively how OpenCL work groups function

• Task-parallelism with multiple queues, priorities
  • Starts to look a lot like pipeline parallelism
  • Queues of waiting tasks serve as inter-stage buffers

• Custom scheduling on “data-parallel” API
  • “Persistent threads”
  • OptiX (NVIDIA)
Pick the right tool for the job

• No one model is best for all apps
  • Or even all parts of one app

• Real-world parallel apps use combinations

• Example: the graphics “pipeline”
  • Pipeline-parallel buffering between stages
  • Programmable stages run data-parallel
  • Task-parallel sharing of unified cores
Task Parallelism

• **Strengths**
  • Scales with irregular/dynamic/heterogeneous problems
  • Viable parallelism approach for global app structure

• **Weaknesses**
  • Limited ability to exploit workload structure
  • No automatic latency-hiding
  • No automatic use of SIMD/SIMT
  • Producer/consumer parallelism not explicit
  • Load balancing relies entirely on task-stealing
Data Parallelism

• **Strengths**
  • Easy to get high utilization of throughput architecture
  • Implicit use of SIMD/SIMT
  • Implicit memory latency hiding

• **Weaknesses**
  • Works best for large, homogeneous batches
  • Work efficiency drops with irregularity
  • Work efficiency drops with small batches
  • Core resources (working set) divided by # of elements
Pipeline Parallelism

• **Strengths**
  - Copes with irregular data amplification
  - Can exploit producer-consumer locality

• **Weaknesses**
  - Best scheduling strategy is workload-dependent
  - No general-purpose tools for current HW
Summary

• Task-, data- and pipeline-parallelism
  • Three proven approaches to scalability
  • Applicable to many problems in visual computing

• Look for these to surface throughout the course
  • Architectures
  • Tools, languages, APIs
  • Algorithms and case studies
Questions?

tfoley@graphics.stanford.edu