Introduction to Parallel Programming Models

Tim Foley
Intel Corp
Overview

• Three models of parallelism
  – Seen in modern game engines
  – Applicable in multiple situations

• After this talk, you should be able to
  – Identify kinds of parallelism in your workloads
  – Select tools that will exploit that parallelism
What goes into a game frame?
Computation graph for Battlefiled: Bad Company provided by EA 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

Pipeline Flow:

1. Input Assembly
2. Vertex Shading
3. Primitive Setup
4. Geometry Shading
5. Rasterization
6. Pixel Shading
7. Output Merging
Data-Parallel  Task-Parallel  Pipeline-Parallel
Structure of this talk

• For each of our models
  – Key idea
  – Mental model
  – Applicability

• Composition
  – How these models combine in the real world
Data Parallelism
Key Idea

• Run a single kernel over many elements

• Can exploit throughput architecture well
  – Amortize per-element cost with SIMD/SIMT
  – Hide memory latency with lightweight threads
Mental Model

- Launch N independent work items
- All running the same program code (kernel)
- Data operated on is function of $0 \leq i < N$

- Domain of computation
  - Determines number/shape of work items
  - Often based on input/output data structure
Simple Data-Parallelism

- Data structure
  - Regular array

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

- Computation domain
  - 1D interval
Simple Data-Parallelism

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

- **Kernel**

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

- **Computation domain**
  - N-D interval
Shapes need not match

- **Data structure**
  - N-D array
  - 1D array

- **Kernel**

```java
void k(int i) {
    B[i] += A[i,0] + A[i,1];
}
```

- **Computation domain**
  - 1D interval
More advanced data-parallelism

• Nested domains
  – Allow work items to communicate
  – Used for sums, scans, sorts, …

• Irregular domains
  – Nested data structures
“Flat” domains

- Flat domain exposes work-item locality
  - Scratch data is private and transient
  - Can keep in registers or caches

- May not exploit rest of memory hierarchy
  - Can’t share temporaries through cache, local memory, etc.
Nested domains

- A domain composed of smaller domains
  - Levels map to memory hierarchy
    - e.g. registers, L1$, L2$, main memory
    - Scratch memory at each level
- Allows work items to share/communicate
  - Useful for “collective” operations: sort, scan, …
  - Need barriers or other synchronization construct
Irregular Domains

• Data structure
  - “ragged” array
  - N-D array- / grid-of-lists

• Multiple possible representations
A simple representation

Count:

Offset:

Storage:
Irregular data parallelism

• Key insight: represent irregular structure as flat index and storage arrays
  – Multiple representations possible

• Allows efficient DP implementation of some irregular algorithms
  – Pay attention for examples this afternoon
Pipeline Parallelism
Key Idea

- Increase throughput by running multiple stages of an algorithm in parallel

- Exploit producer-consumer locality
  - On-chip FIFOs
  - Efficient bus between cores
GPU Pipeline (DX10)

- Pipeline of
  - Fixed-function stages
  - Programmable stages
    - Data-parallel kernels
  - Stages run in parallel
    - Even for unified cores
- Queues between stages
  - Often in HW

Diagram:
- Input Assembly
- Vertex Shading
- Primitive Setup
- Geometry Shading
- Rasterization
- Pixel Shading
- Output Merging
Why pipelines?

- **Variable rate amplification**
  - Rasterizer: 1 tri 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
  – Specialized inter-stage FIFOs
  – On-chip caches, busses or scratchpads
Host/GPU pipeline

- Graphics command stream
  - Host packs, GPU consumes in parallel

- Distribute pack work across N host cores
  - Common technique in console graphics
  - Will eventually translate to desktop

Host:

GDP:

... Prepare Frame N  Prepare Frame N+1  Prepare Frame N+2 ...

... Render Frame N-1  Render Frame N  Render Frame N+1 ...
Build your own pipeline?

- Sounds simple
  - Pick per-stage kernels, wire up dataflow
- Challenging in practice
  - Scheduling stages to cores
  - Bounding intermediate storage
- Active area of research
  - See Stanford GRAMPS for a recent effort
Task Parallelism
Key Idea

- Achieve scalability for heterogeneous and irregular work by expressing dependencies directly

- Lightweight cooperative scheduling
Why tasks?

- Start with sequential workload
Why tasks?

- Identify data- and pipeline-parallel steps
Why tasks?

- Identify data- and pipeline-parallel steps
- Assume perfect scaling
Why tasks?

- Cost now dominated by sequential part
  - The part not suited to data- or pipeline-parallelism
- Oh yeah... that's just Amdahl's Law
Using tasks

• If we know dependencies between the steps
Using tasks

- If we know dependencies between the steps
- We can distribute the work across cores
  - Respecting the dependencies
Finite # of cores

- It looks more like this
  - Multiple kinds of work fill in the “cracks”
Task systems

• Standard practice for PS3 games
  – Gaining currency on other consoles, desktop

• One worker thread per HW context
  – Cooperative scheduling
  – Pull tasks from an incoming queue
  – Load balance using “work stealing” [Cilk]
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
- More scheduling overhead
  - Tune task size to strike a balance
Tasks take-away

• Can’t write sequential app with parallel pieces
  – Amdahl’s Law will bite you every time
• Must involve parallelism from the top down

• Task systems
  – Handle the code that won’t fit other models
    – Heterogeneous, irregular
    – Dynamically generated work, dependencies
  – Provide scalability and load balancing
Composition
Picking the right tools

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

• Real-world parallel apps use combinations
  – Case in point: the graphics “pipeline”
    – Pipeline-parallel buffering between stages
    – Programmable stages run data-parallel
    – Task-parallel sharing of shader cores
Data Parallelism

• Strengths
  – Good utilization of throughput architecture
  – Implicit use of SIMD/SIMT
  – Implicit memory latency hiding

• Weaknesses
  – Works best for large, homogeneous problems
  – Work efficiency drops with irregularity
Pipeline Parallelism

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

- **Weaknesses**
  - Best scheduling strategy workload-dependent
  - No general-purpose tools for current HW
Task Parallelism

• **Strengths**
  - Copes well with irregular/dynamic problems
  - Viable parallelism approach for a whole app

• **Weaknesses**
  - No automatic support for latency-hiding
  - Must explicitly write for SIMD
Summary

• Data-, pipeline- and task-parallelism
  – Three proven approaches to scalability
  – Applicable to many problems in graphics

• Look for these to surface as we discuss
  – Tools
  – Architectures
  – Algorithms
Acknowledgements

- EA DICE

- Co-organizers: Jason, Justin, Mark

- Stanford: Kayvon Fatahalian

- Intel: Aaron Lefohn, Andrew Lauritzen, Tim Mattson
Questions?