PTASK + DANDELION: DATA-FLOW PROGRAMMING SUPPORT FOR HETEROGENEOUS PLATFORMS

Chris Rossbach and Jon Currey
Microsoft Research Silicon Valley
NVIDIA GTC 5/17/2012
Motivation/Overview

- GPU programming is super-important
  - (dissenters: are you at the right conference?)
  - still difficult despite amazing progress
- Technology stacks (at least partly) to blame:
  - OS thinks GPU is I/O device
  - Data movement + algorithms coupled
  - High level language support too low level
Motivation/Overview

- GPU programming is super-important
  - (dissenters: are you at the right conference?)
  - still difficult despite amazing progress

- GPU technology stacks (at least partly) to blame:
  - OS thinks GPU is I/O device
  - Data movement + algorithms coupled
  - High level language support too low level

1. These properties limit GPUs
2. OS-level abstractions are needed
3. OS support → better language support
Outline

- The case for OS support
- The case for dataflow abstractions
- PTask: better OS support for GPUs
- Dandelion: language+runtime support
- Related and Future Work
- Conclusion
Layered abstractions + OS support

* 1:1 correspondence between OS-level and user-level abstractions
* Diverse HW support enabled HAL
GPU abstractions

1. No kernel-facing API
2. Too much runtime support in OS
3. Poor composability
No OS support $\rightarrow$ No isolation

CPU+GPU schedulers not integrated!
...other pathologies abundant
Composition: Gestural Interface

- Raw images
- Capture camera images
- Geometric transformation
- Capture
- Xform
- Noisy point cloud
- Detect
- Detect gestures
- Noise filtering

- High data rates
- Data-parallel algorithms
  ... good fit for GPU
# What We’d Like To Do

> capture | xform | filter | detect &

| CPU | GPU | GPU | CPU |

- Modular design
  - flexibility, reuse
- Utilize heterogeneous hardware
  - Data-parallel components $\rightarrow$ GPU
  - Sequential components $\rightarrow$ CPU
- Using OS provided tools
  - processes, pipes
GPU Execution model

- GPUs cannot run OS:
  - different ISA
  - Memories disjoint, or have different coherence guarantees

- Host CPU must “manage” GPU execution
  - Program inputs explicitly transferred/bound at runtime
  - Device buffers pre-allocated
Data migration

#> capture | xform | filter | detect &

**User Level**
- capture
- xform
- filter
- detect

**Kernel Level**
- read()
- write()
- copy to GPU
- copy from GPU
- copy to GPU
- copy from GPU
- IRP

**HW**
- camdrv
- GPU driver
- HIDdrv

**GPU**
- Run!

PTask+Dandelion: NVIDIA GTC 5/17/2012
Outline

- The case for OS support
- The case for dataflow abstractions
- PTask: Dataflow for GPUs
- Dandelion: LINQ on GPUs
- Related and Future Work
- Conclusion
Why dataflow?

Matrix
gemm(Matrix A, Matrix B) {
copyToGPU(A);
copyToGPU(B);
invokeGPU();
Matrix C = new Matrix();
copyFromGPU(C);
return C;
}

What happens if I want the following?
Matrix D = A x B x C
Composed matrix multiplication

Matrix AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
Composed matrix multiplication

Matrix

AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A, B);
    Matrix AXBxC = gemm(AxB, C);
    return AXBxC;
}

Matrix

gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
Composed matrix multiplication

Matrix

AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}

...only to be copied right back!

We need different abstractions that help get around this...
Dataflow: runtime manages data movement

- nodes $\rightarrow$ computation
- edges $\rightarrow$ communication
- leaves flexibility for the runtime
- minimal specification of data movement
- asynchrony is a runtime concern (not programmer concern)
Outline

- The case for OS support
- The case for dataflow abstractions
- PTask: Dataflow for GPUs
- Dandelion: LINQ on GPUs
- Related and Future Work
- Conclusion
PTask OS abstractions: dataflow!

- **ptask** (parallel task)
  - Has *priority* for fairness
  - Analogous to a process for GPU execution
  - List of input/output resources (*e.g.* stdin, stdout...)

- **ports**
  - Can be mapped to ptask input/outputs
  - A data source or sink

- **channels**
  - Similar to pipes, connect arbitrary ports
  - Specialize to eliminate double-buffering

- **graph**
  - Directed, connected ptasks

- **datablocks**
  - Memory-space transparent buffers

- **data**: specify *where*, not *how*
- **OS objects** -> **OS RM possible**
PTask Graph: Gestural Interface

#> capture | xform | filter | detect &

ptask graph

mapped mem

GPU mem → GPU mem

Optimized data movement
Data arrival triggers computation
### Location Transparency: Datablocks

<table>
<thead>
<tr>
<th>Datablock</th>
<th>Main Memory</th>
<th>GPU 0 Memory</th>
<th>GPU 1 Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>V</td>
<td>M</td>
<td>RW</td>
</tr>
<tr>
<td>main</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>gpu0</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>gpu1</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

- **Logical buffer**
  - backed by multiple physical buffers
  - buffers created/updated lazily
  - mem-mapping used to share across process boundaries

- **Track buffer validity per memory space**
  - writes invalidate other views

- **Flags for access control/data placement**

---

**Enables OS-mediated IPC through GPU memory**
Datablock Action Zone

#> capture | xform | filter ...

Datablock

<table>
<thead>
<tr>
<th>space</th>
<th>V</th>
<th>M</th>
<th>RW</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>gpu</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Main Memory

GPU Memory

process
ptask
port
channel
datablock

PTask+Dandelion: NVIDIA GTC 5/17/2012
Outline

- The case for OS support
- The case for dataflow
- PTask: Dataflow for GPUs
  - Some numbers
- Dandelion: LINQ on GPUs
- Related and Future Work
- Conclusion
Implementation

Windows 7

Two PTask API implementations:

1. Stacked UMDF/KMDF driver
   - Kernel component: mem-mapping, signaling
   - User component: wraps DirectX, CUDA, OpenCL
   - syscalls $\rightarrow$ DeviceIoControl() calls

2. User-mode library
Gestural Interface evaluation

- Implementations
  - **pipes**: capture | xform | filter | detect
  - **modular**: capture+xform+filter+detect, 1process
  - **handcode**: data movement optimized, 1process
  - **ptask**: ptask graph

- Configurations
  - **real-time**: driven by cameras
  - **unconstrained**: driven by in-memory playback
Gestural Interface Performance

- **Windows 7 x64 8GB RAM**
- **Intel Core 2 Quad 2.66GHz**
- **GTX580 (EVGA)**

- Lower is better

- **Runtime**:
  - ~2.7x less CPU usage
  - 16x higher throughput
  - ~45% less memory usage

- **Throughput**:
  - 11.6% higher throughput
  - Lower CPU util: no driver program

- **Compared to hand-code**

PTask+Dandelion: NVIDIA GTC
5/17/2012
Things I didn’t show you

- PTask schedules GPUs like CPUs
- PTask provides performance isolation
- Locality-aware scheduling
- PTask transparently uses multiple GPUs
- Proof-of-concept in Linux
- Generality: micro-benchmarks
- ...

PTask+Dandelion: NVIDIA GTC 5/17/2012
Outline

- The case for OS support
- The case for dataflow abstractions
- PTask: better OS support for GPUs
- Dandelion: language+runtime support
- Related and Future Work
- Conclusion
Dandelion *Goal*

- A single programming interface for heterogeneous platforms:
  - Clusters comprising:
    - CPUs
    - GPUs
    - FPGAs
    - You name it...

- Programmer: writes sequential code once
- Runtime: partitions, runs in parallel on available resources
- ...achievable?
Dandelion goal

- Offload data-parallel code fragments
- Small cluster of multi-core + multi-GPU
- Starting point:
  - LINQ queries

(a less holy and attractive vessel: usually just as effective, depending on use case)
Wait! What’s a LINQ Query?

- **Language Integrated Query**
- Relational operators over objects:

  ```
  var res = collection.
  .Where(c => c.hasSomeProperty())
  .Select(c => new {c, quack});
  ```

- **Why is LINQ important?**
  - Expresses many important workloads easily
    - K-Means, PageRank, Sparse Matrix SVD, ...
  - LINQ Queries are **data-parallel**
  - Natural fit for data flow
Dandelion Overview

LINQ Query
(or other parallel code)

Distributed Graph Management

Local PTask Graph

Machine with CPUs and GPUs

Distribution by DryadLINQ
c.f. MapReduce/Apache Hadoop

CPU Task

GPU Task

Runtime
Dandelion: Local View

- LINQ Query (or other parallel code)
- Generate GPU code for user types and functions
- Construct dataflow graphs using templated ‘PTask Primitives’

Dandelion Compiler

Dataflow Graph

A → B
C → D

PTask Runtime

Machine with CPUs and GPUs
Example: K-Means Clustering

- Partition \( n \) points into \( k \) clusters
  - Step 0: Pick \( k \) initial cluster centers
  - Step 1: Each point \( \rightarrow \) cluster with nearest center
  - Step 2: Recompute centers (calculate cluster means)
  - Iterate steps 1 & 2 until stable

```csharp
centers = points
    .GroupBy(point => NearestCenter(point, centers))
    .Select(g => g.Aggregate((x, y) => x+y)/g.Count());
```
GroupBy

- Groups an input sequence by key
- Custom function maps input elements \( \rightarrow \) key

```csharp
List<Group<int>> res = ints.GroupBy(x => x);
```

**Example:**

```
ints = [10, 30, 20, 10, 20, 30, 10]

res = ['10', '10', '10', '30', '30', '20', '20']
```
Consider CPU Implementation

```csharp
foreach (T elem in InputSequence)
{
    key   = KeySelector(elem);
    group = GetGroup(key);
    group.Add(elem);
}
```

- Mapping to GPUs is not obvious
  - How to assign work across 1000’s of threads?
  - Synchronizing group creation is problematic
  - “Append” is problematic
Parallel GroupBy

Process each input element in parallel
- grouping ~ shuffling
- item output offset = group offset + item number
- ... but how to get the group offset?

<table>
<thead>
<tr>
<th>ints</th>
<th>10</th>
<th>30</th>
<th>20</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>res</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>30</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Start index of each group in the output sequence
Number of elements in each group
Number of groups and integer group IDs
## GPU GroupBy: Multiple Passes

### Assign group IDs

<table>
<thead>
<tr>
<th>GroupID</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>10, 20, 30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Compute group sizes

<table>
<thead>
<tr>
<th>GroupID</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>10, 20, 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group Size</td>
<td>3, 2, 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Compute start indices

<table>
<thead>
<tr>
<th>GroupID</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group ID</td>
<td>10, 20, 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group Start Index</td>
<td>0, 3, 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Write Outputs

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
<th>10</th>
<th>20</th>
<th>20</th>
<th>30</th>
<th>30</th>
</tr>
</thead>
</table>

- **Uses a lock-free hash table**
- **Looks up hash table to get group ID**
- **Atomically increments group counters**
- **Exclusive prefix sum on group counts**
- **Write output to correct location**
  - Uses atomic increment
GroupBy As Composed Primitives

10 30 20 10 20 30 10

Assign group IDs

GroupID:
0 1 2

Compute group sizes

GroupID:
0 1 2
Group Size:
3 2 2

Compute start indices

GroupID:
0 1 2
Group Start Index:
0 3 5

Write Outputs

10 10 10 20 20 30 30
K-Means Dandelion graph

PTask+Dandelion: NVIDIA GTC 5/17/2012
class KMeans {
    int NearestCenter(Vector point, IEnumerable<Vector> centers) {
        int minIndex = 0, curIndex = 0;
        double minValue = Double.MaxValue;
        foreach (Vector center in centers) {
            double curValue = (center - point).Norm2();
            minIndex = (minValue > curValue) ? curIndex : minIndex;
            minValue = (minValue > curValue) ? curValue : minValue;
            curIndex++;
        }
        return minIndex;
    }

    IQueryable<Vector> Iteration(IQueryable<Vector> points, IQueryable<Vector> centers) {
        return points
            .GroupBy(point => NearestCenter(point, centers))
            .Select(g => g.Aggregate((x, y) => x + y) / g.Count());
    }
}
K-Means in Dandelion

class KMeans {
    int NearestCenter(Vector point, IEnumerable<Vector> centers) {
        int minIndex = 0, curIndex = 0;
        double minValue = Double.MaxValue;
        foreach (Vector center in centers) {
            double curValue = (center - point).Norm2();
            minIndex = (minValue > curValue) ? curIndex : minIndex;
            minValue = (minValue > curValue) ? curValue : minValue;
            curIndex++;
        }
        return minIndex;
    }

    IQueryable<Vector> Iteration(IQueryable<Vector> points, IQueryable<Vector> centers) {
        return points
            .GroupBy(point => NearestCenter(point, centers))
            .Select(g => g.Aggregate((x, y) => x + y) / g.Count());
    }
}
K-Means Performance

- 4-core Intel Xeon W3550 @ 3.07 GHz, 6GB RAM
- NVIDIA GTX580 GPU with 1.5GB GDDR
- PTask on NVIDIA CUDA version 4.0

**Streaming** is 23x faster than single-threaded CPU...

... and think we can do even better
PageRank in Dandelion

```csharp
public struct Rank {
    public UInt64 PageID;
    public UInt64 PageRank;
    ...
}

public struct Page {
    public UInt64 PageID;
    public UInt64 LinkedPageID;
    int numLinkedPages;
    ...
}

// Join Pages with Ranks, and disperse updates
var partialRanks =
    from rank in ranks 
    join page in pages 
    on rank.PageID equals page.PageID 
    select new Rank( 
        page.LinkedPageID, 
        rank.PageRank / page.NumLinks);

// Re-accumulate Ranks
var newRanks =
    from partialRank in partialRanks 
    group partialRank.PageRank 
    by partialRank.PageID into g 
    select new Rank(g.Key, g.Sum());
```
Status

- Dandelion is a research prototype
  - Working end to end … now add more workloads
  - Current results promising … but incomplete

- Many areas of research remain unexplored
  - Optimal work partitioning and scheduling
  - When to expose/hide architectural features
  - We *will* need programmer hints sometimes. When?
Outline

- The case for OS support
- The case for dataflow abstractions
- PTask: Dataflow for GPUs
- Dandelion: LINQ on GPUs
- Related and Future Work
- Conclusion
Related Work

- Prior work: regular (scientific, HPC) apps
  -Accelerator: Array computations [ASPLOS ’06]
  -Amp/C++
  -StreamIt → CUDA [CGO ‘09, LCTES ‘09]
  -MATLAB → CUDA compiler [PLDI ‘11]

- OS support for heterogeneous platforms:
  -Helios [Nightingale 09], BarrelFish [Baumann 09], Offcodes [Weinsberg 08]

- GPU Scheduling
  -TimeGraph [Kato 11], Pegasus [Gupta 11]

- Graph-based programming models
  -Synthesis [Masselin 89], Monsoon/Id [Arvind], Dryad [Isard 07]
  -StreamIt [Thies 02], DirectShow, TCP Offload [Currid 04]

- Tasking
  -Tessellation, Apple GCD, …
Conclusion and Future Work

- Better abstractions for GPUs are critical
  - Enable fairness & priority
  - OS can use the GPU
- Dataflow: a good fit abstraction
  - system manages data movement
  - performance benefits significant
  - enables more flexible technology stack
- Proof of concept: LINQ on GPUs
- Future work
  - Automatic task partitioning across CPU and GPU
  - Finding best parallelization for a given LINQ query

Thank you! Questions?