REALIZING HIGH-PERFORMANCE PIPELINES USING PIKO

Kerry Seitz, Anjul Patney, Stanley Tzeng, John D. Owens
OUTLINE

- Motivation
- Related Work
- Abstraction
- Language
- Compiler
- Results
MOTIVATION
FLEXIBILITY

- Evolving application scenarios
  - Deferred shading on GPUs
  - Ray tracing in Reyes

- Diversifying architectures
  - Discrete GPUs
  - Heterogeneous CPU-GPU architectures
  - Mobile GPUs

- Limited programmability of shaders
GRAPHICS PIPELINES

- Functional description for graphics algorithms
- Intuitive and flexible
- Does not directly translate to implementations
IMPLEMENTATIONS OF GRAPHICS PIPELINES

Vertex Shade
  ↓
Rasterize
  ↓
Fragment Shade
  ↓
Composite

VS → VS → VS → VS → VS → VS
Rast → Rast
FS → FS → FS → FS → FS → FS
Comp → Comp → Comp → Comp
IMPLEMENTATIONS OF GRAPHICS PIPELINES

- Parallelism
- Varying work granularity
- Coherence
- Data locality
PIKO - BASIC IDEA

- Augment traditional pipelines to expose these characteristics

- Within each stage, we group computation using spatial tiles

- We propose three programmable phases per stage
  - AssignBin
  - Schedule
  - Process
RELATED WORK
GRAMPS

• A flexible pipeline abstraction

• Similarities
  – Pipelines as graphs
  – Expression for intra-stage parallelism
  – Use of “queue sets”

• Differences
  – Specialization of stages
  – Scheduling granularity

[Sugerman et al. 2009]
HALIDE

• Programmable scheduling for image processing

• Similarities
  – Decouple schedule from algorithm
  – Flexible image-space work granularity

• Differences
  – Image processing applications (pixels)
  – Central Scheduler
  – Shorter pipelines

[Ragan-Kelley et al. 2012]
SOFTWARE PIPELINES ON GPUS

- OptiX
- CudaRaster
- Line Sampled DOF
- RenderAnts
- VoxelPipe
- Image-Space Photon Mapping
- FreePipe
SOFTWARE PIPELINES ON GPUS

• Similarities
  – Multi-kernel approach
  – Use of spatial-bins or tiles

• Differences
  – Focus on single pipeline
SOFTWARE PIPELINES ON GPUs

Strawman
Multikernel

Off-chip Memory

A
A
B
B
C
C
C
C

Cores
Pipeline Progress

Strawman
Persistent threads

Off-chip Memory

A
B
C
A
B
C
A
B
C

Cores
Pipeline Progress

Strawman
Piko Optimized

Off-chip Memory

A
B
C
A
B
C
A
B
C

Cores
Pipeline Progress
ABSTRACTION
Why

- Is everywhere
- Is intuitive
- Is helpful
  - locality
  - parallelism

Forms a link between algorithm and implementation
SPATIAL TILING FOR PARALLELISM
SPATIAL TILING FOR LOCALITY

[Diagram showing spatial tiling with cores 1 to 4]
EXPRESSING PIPELINES USING TILES

- Add programmable tiling to a pipeline

- What does the programmer tell us?
  - How to group data into bins? “AssignBin”
  - When and where to schedule each bin? “Schedule”
  - What to compute for each bin? “Process”
LANGUAGE
PIKO LANGUAGE

- Basically C++
PIKO LANGUAGE

- Basically C++

- Piko API
  - Stage class
  - Pipe class
  - Math library (using NVIDIA’s libdevice library)
PIKO LANGUAGE

- Basically C++

- Piko API
  - Stage class
  - Pipe class
  - Math library (using NVIDIA’s libdevice library)

- Added *Directives* for optimization information
  - Policy - express common patterns
  - Hints - can theoretically be derived from code
DIRECTIVES - POLICIES

- AssignBin
  - OneToAll
  - OneToManyRange
  - OneToOneIdentity
DIRECTIVES - POLICIES

- AssignBin
  - OneToAll
  - OneToManyRange
  - OneToOneIdentity

- Schedule
  - LoadBalance (dynamic)
  - DirectMap (static)
  - Serialize
  - All
    - TileSplitSize
  - EndStage(X)
  - EndBin
DIRECTIVES - POLICIES

- **AssignBin**
  - OneToAll
  - OneToManyRange
  - OneToOneIdentity

- **Process**
  - None

- **Schedule**
  - LoadBalance (dynamic)
  - DirectMap (static)
  - Serialize
  - All
    - TileSplitSize
  - EndStage(X)
  - EndBin

- **Schedule**
  - None

- **Schedule**
  - LoadBalance (dynamic)
  - DirectMap (static)
  - Serialize
  - All
    - TileSplitSize
  - EndStage(X)
  - EndBin
DIRECTIVES - HINTS

- AssignBin
  - MaxOutBins
DIRECTIVES - HINTS

- AssignBin
  - MaxOutBins

- Schedule
  - None
DIRECTIVES - HINTS

- AssignBin
  - MaxOutBins

- Process
  - MaxOutPrims

- Schedule
  - None
class FragmentShaderStage : public Stage<8, 8, 64, piko_fragment, piko_fragment> {
};
EXAMPLE STAGE

class FragmentShaderStage : public Stage<8, 8, 64, piko_fragment, piko_fragment>
{
public:
    void assignBin(piko_fragment f) {
        int binID = getBinFromPosition(f.screenPos);
        this->assignToBin(f, binID);
    }
};
class FragmentShaderStage : public Stage<8, 8, 64, piko_fragment, piko_fragment> {
    public:
    void assignBin(piko_fragment f) {
        int binID = getBinFromPosition(f.screenPos);
        this->assignToBin(f, binID);
    }

    void schedule(int binID) {
        specifySchedule(LOAD_BALANCE);
    }
};
```cpp
class FragmentShaderStage : public Stage<8, 8, 64, piko_fragment, piko_fragment>
{
public:
    void assignBin(piko_fragment f) {
        int binID = getBinFromPosition(f.screenPos);
        this->assignToBin(f, binID);
    }

    void schedule(int binID) {
        specifySchedule(LOAD_BALANCE);
    }

    void process(piko_fragment f) {
        cvec3f material = genvec3f(0.80f, 0.75f, 0.65f);
        cvec3f lightvec = normalize(genvec3f(1,1,1));
        f.color = material * dot(f.normal, lightvec);
        this->emit(f);
    }
};
```
EXAMPLE PIPE

class RasterPipe : public PikoPipe
{

};
EXAMPLE PIPE

class RasterPipe : public PikoPipe
{
    VertexShaderStage vertexShader;
    RasterStage raster;
    FragmentShaderStage fragmentShader;
};
```cpp
class RasterPipe : public PikoPipe
{
    VertexShaderStage vertexShader;
    RasterStage raster;
    FragmentShaderStage fragmentShader;

public:
    RasterPipe() {
        PikoPipe::pikoConnect(vertexShader, raster);
        PikoPipe::pikoConnect(raster, fragmentShader);
    }
};
```
COMPILER
THREE PHASES

- Analysis
  - Clang
THREE PHASES

- Analysis
  - Clang

- Optimization
  - LLVM
  - Kernel Planner
THREE PHASES

- Analysis
  - Clang

- Optimization
  - LLVM
  - Kernel Planner

- Code Generation
  - NVPTX backend
  - C++ runner code
ANALYSIS

- Parse Clang AST twice
  - Gather stage information
  - Gather pipe information
ANALYSIS

- Parse Clang AST twice
  - Gather stage information
  - Gather pipe information

- Produce *Pipeline Skeleton*
  - Stage policies and hints
  - Connections between stages
OPTIMIZATION - KERNEL PLANNER

- Resolves ordering dependencies
  - Explicit barriers
  - Cycles
OPTIMIZATION - KERNEL PLANNER

- Resolves ordering dependencies
  - Explicit barriers
  - Cycles

- Map phases to kernels
  - Simplest - each phase is a separate kernel
  - Optimized - merge phases together where appropriate
    - E.g. process + assignBin
INTER-STAGE OPTIMIZATIONS

- Kernel Fusion
- Scheduler Elimination
- Static Dependency Resolution
KERNEL FUSION

- **Goal**
  - Exploit producer-consumer locality

- **Opportunity**
  - Same work granularity (tile size)
  - No synchronization
SCHEDULER ELIMINATION

- **Goal**
  - Eliminate software scheduling overheads

- **Opportunity**
  - Most pipelines simply require a load-balanced schedule
CODE GENERATION

- C++ Source for kernel runner
  - Use Kernel Plan
  - CUDA Driver API
CODE GENERATION

- C++ Source for kernel runner
  - Use Kernel Plan
  - CUDA Driver API

- NVPTX LLVM Backend for device code
  - Link with libdevice
  - Inline PTX
LIBDEVICE

- Provide own header for math functions

- E.g.

```c
extern "C"
double __nv_fmin(double x, double y);

namespace piko {
    double fmin(double x, double y) {
        return __nv_fmin(x, y);
    }
}
```
LIBDEVICE

- Provide own header for math functions

- E.g.

```cpp
extern "C"
double __nv_fmin(double x, double y);

namespace piko {
    double fmin(double x, double y) {
        return __nv_fmin(x, y);
    }
}
```

Gets resolved when the libdevice LLVM module is linked
 INLINE PTX EXAMPLES

- Atomic Add

```c
int myAtomicAdd(int* v1, int v2) {
  int res;
  asm("atom.add.s32 %0, [%1], %2;" : "=r"(res) : "1"(v1), "r"(v2));
  return res;
}
```
INLINE PTX EXAMPLES

- **Atomic Add**

  ```c
  int myAtomicAdd(int* v1, int v2) {
    int res;
    asm("atom.add.s32 %0, [%1], %2;" : "=r"(res) : "1"(v1), "r"(v2));
    return res;
  }
  ```

- **Ballot**

  ```c
  int myBallot(int pred) {
    int res;
    asm __volatile__ ("{
      \n      \n      .reg .pred \t%p1;
      \n      \n      setp.ne.u32 \t%p1, %1, 0; \n      \n      \n      "vote.ballot.b32 \t%0, %p1; \n      \n      \n      "}" : "=r"(res) : "r"(pred));
    return res;
  }
  ```
RESULTS
EXPERIMENTAL SETUP

- A feed-forward rasterization pipeline
- We studied its efficiency, flexibility, and portability
RAST-STRAWMAN (RSM)
RAST-FREEPIPE (RFP)

Vertex Shade
Geometry Shade
Rasterize
Fragment Shade
Depth Test
Composite

Vertex Shade
Geometry Shade
Rasterize
Fragment Shade
Depth Test
Composite
RAST-LOCALITY (RLC)

Vertex Shade
Geometry Shade
Rasterize
Fragment Shade
Depth Test
Composite

Vertex Shade
Geometry Shade
Rasterize
Fragment Shade
Depth Test
Composite
RAST-LOADBALANCE (RLB)

Vertex Shade
Geometry Shade
Rasterize
Fragment Shade
Depth Test
Composite

Vertex Shade
Geometry Shade
Rasterize
Fragment Shade
Depth Test
Composite
RESULTS - FRAMEWORK

We tested Piko against two pipeline implementations:

- Strawman - 1 Kernel per pipeline stage.
- Freepipe - 1 Kernel for the whole pipeline.

We configure Piko to produce two types of pipelines:

- Loadbalance - Pikoc does not merge pipeline stages.
- Locality - Pikoc merges pipeline stages as much as possible.
RESULTS - ALTERNATIVE HARDWARE

- Implemented piko-generated pipelines on heterogeneous architectures that have CPU and GPU on the same chip.
- We tested Piko on Intel’s Ivy Bridge.
- Since Pikoc does not generate OpenCL backend code, we used Pikoc’s pipeline skeleton and hand-coded the pipelines.
IS IT FLEXIBLE?

Locality

Ability to explore multiple design choices for one pipe
IS IT EFFICIENT?

- Yes
  - rast-loadbalance is 4x faster than rast-strawman
  - Combination of tiling and opportunistic kernel fusion

- No
  - RLB is 3-4x slower than state of the art (cudaraster)
  - Aggressive optimizations not pursued
  - Indirect optimizations unavailable to Piko
### IS IT PORTABLE?

<table>
<thead>
<tr>
<th>GPU Cores</th>
<th>CPU Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 0 1 2 3 4 5 6 0 1 2 3 4</td>
<td>1 2 3 4 5 6 0 1 2 3 4 5</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 0 1 2 3 4</td>
<td>3 4 5 6 0 1 2 3 4 5 6 0</td>
</tr>
<tr>
<td>2 3 4 5 6 0 1 2 3 4 5 6</td>
<td>5 6 0 1 2 3 4 5 6 0 1 2</td>
</tr>
<tr>
<td>4 5 6 0 1 2 3 4 5 6 0 1</td>
<td>0 1 2 3 4 5 6 0 1 2 3 4</td>
</tr>
<tr>
<td>6 0 1 2 3 4 5 6 0 1 2 3</td>
<td>0 1 2 3 0 1 2 3 0 1 2 3</td>
</tr>
<tr>
<td>1 2 3 4 5 6 0 1 2 3 4 5</td>
<td>0 1 2 3 0 1 2 3 0 1 2 3</td>
</tr>
<tr>
<td>3 4 5 6 0 1 2 3 4 5 6 0</td>
<td>0 1 2 3 0 1 2 3 0 1 2 3</td>
</tr>
<tr>
<td>5 6 0 1 2 3 4 5 6 0 1 2</td>
<td>0 1 2 3 0 1 2 3 0 1 2 3</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 0 1 2 3 4</td>
<td>0 1 2 3 0 1 2 3 0 1 2 3</td>
</tr>
</tbody>
</table>

**Discrete GPU**

**Hybrid CPU-GPU**
Is it portable?

- NVIDIA GPU
- Intel Ivy Bridge

Locality works better for CPU + GPU

Load balance works for both architectures
This talk introduced Piko:

- User writes pipeline stages split into three components: assignBin, schedule, and process
- Pikoc will fuse pipeline stages together into an optimal set of CUDA kernels
- Pikoc runs with a LLVM backend so that we can translate to other architectures as well (in the future)
FUTURE WORK

- Additional Features into Piko:
  - Printf (now in Piko!)
  - Support for more GPU instructions (i.e. funnel shift, shuffle, etc.)

- Additional Pipeline Implementations:
  - Raytracing, hybrid pipelines

- Additional Target Architectures:
  - Potential Tegra K1 target
  - HSAIL LLVM IR Integration
THANK YOU