CATEGORY: **GRAPHICS PERFORMANCE OPTIMIZATION**
POSTER **GP02**
CONTACT NAME Kerry Seitz: kaseitz@ucdavis.edu
GPU TECHNOLOGY CONFERENCE

Piko: A Design Framework for Programmable Graphics Pipelines

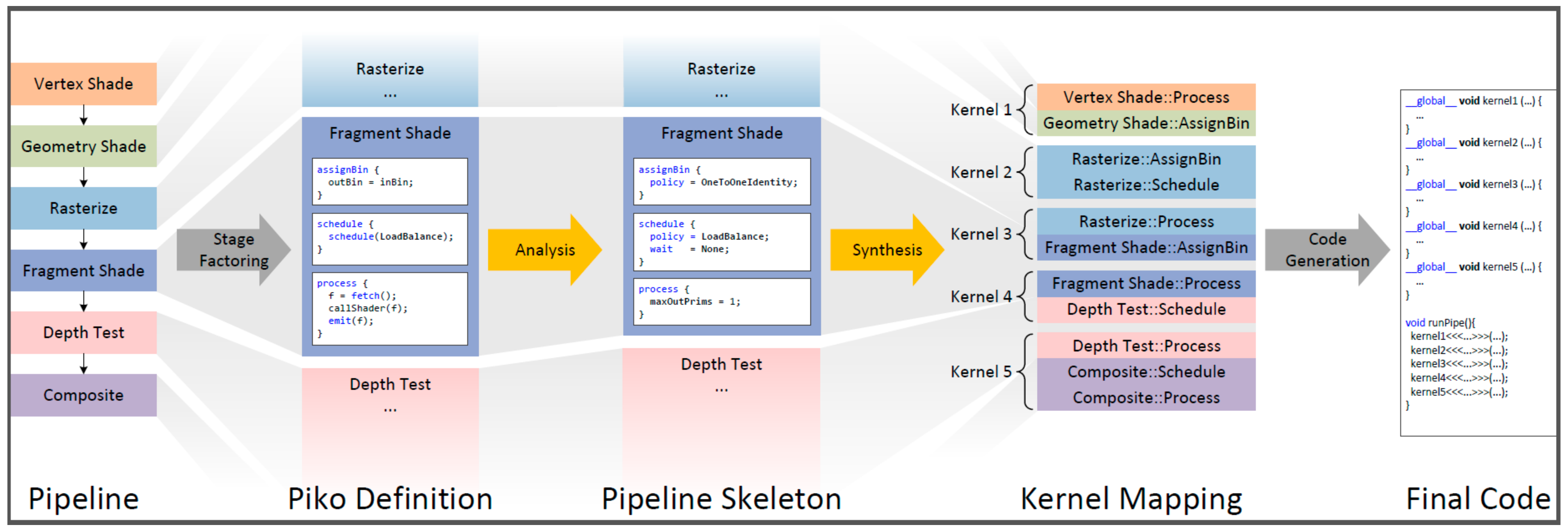
Kerry A. Seitz, Jr., Anjul Patney, Stanley Tzeng, and John D. Owens
University of California, Davis
UCDAVIS UNIVERSITY OF CALIFORNIA

Introduction

We present Piko, an abstraction for designing efficient programmable graphics pipelines. Piko is built around managing work granularity in a programmable and flexible manner, allowing programmers to build load-balanced parallel pipeline implementations, to exploit spatial and producer-consumer locality in the pipeline, and to explore tradeoffs between these considerations.

Graphics pipelines inherently consist of logically separate stages that operate on primitives. Ordinarily, a programmer needs to combine these stages together in order to achieve high performance. Reusing a stage in a different pipeline requires rewriting the stage to fit with the new pipeline. Furthermore, the pipeline needs to be altered significantly or even rewritten entirely to achieve high performance on different architectures. Piko aims to alleviate these limitations by emphasizing programmability, efficiency, and portability. In addition, Piko is flexible so that programmers can express a wide variety of current pipelines using this abstraction. Piko can also be used to develop new and unique pipelines, and it promotes such exploration through modular stage reuse.

Piko uses programmable spatial binning to divide workloads in a manner that both exploits spatial locality and exposes parallelism. The screen is divided into uniform 2D bins, and each primitive in a pipeline is assigned to a bin (or multiple bins) in a programmable fashion (e.g. based on its location in screen space or based on its bin in the previous stage). Piko then schedules these bins onto cores, where the primitives are modified and transformed based on the pipeline stages.



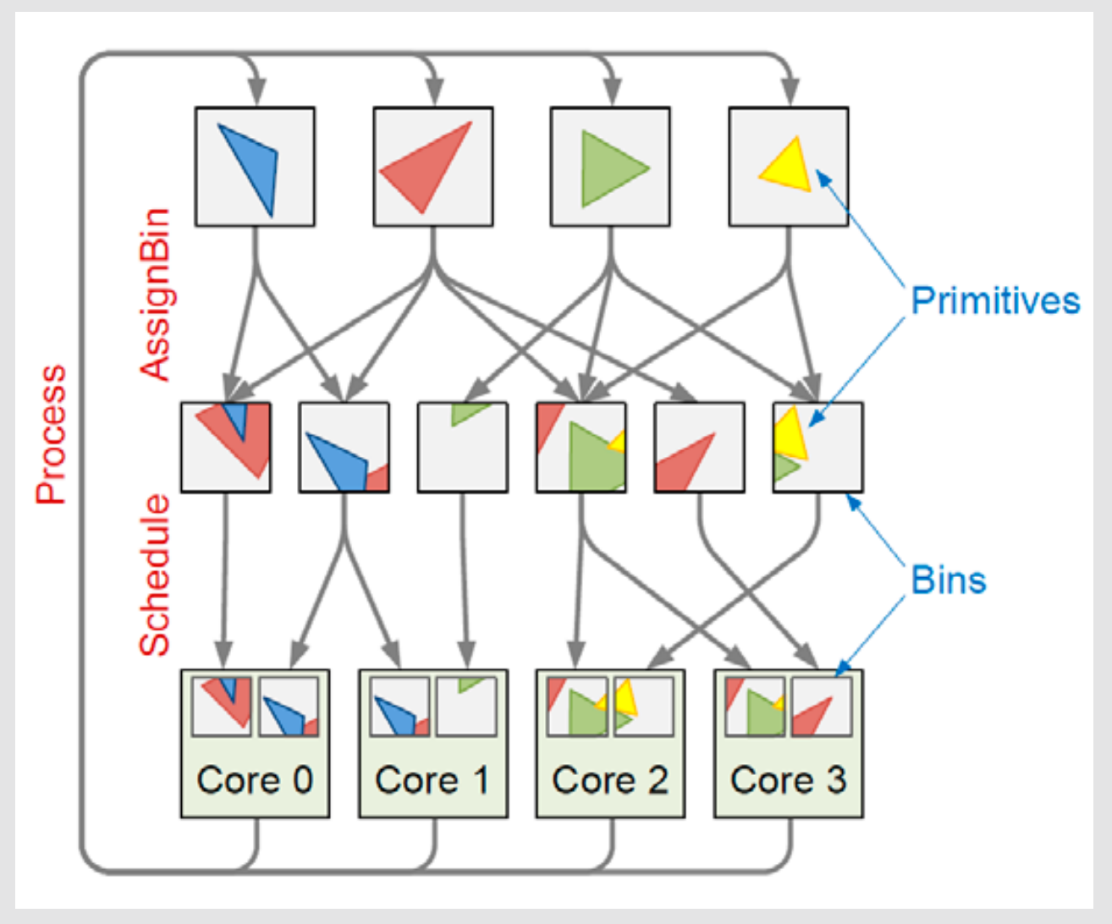
Our Piko abstraction applied to a forward raster pipeline. The user writes a pipeline definition in terms of stages factored into *AssignBin*, *Schedule*, and *Process* phases. Piko runs an analysis step to generate a pipeline skeleton, and then a synthesis step to divide the pipeline into kernels. In the example above, Piko fuses the Depth Test and Composite stages into one kernel. The output of Piko is a set of efficient kernels optimized for multiple hardware targets and a schedule for running the pipeline.

Defining a Pipeline

Piko pipelines are defined by writing and composing together modular stages. Each stage consists of three programmable phases: *AssignBin*, *Schedule*, and *Process*.

Phase	Granularity	Purpose
`AssignBin`	Per-Primitive	How to group computation?
`Schedule`	Per-Bin	When to compute? Where to compute?
`Process`	Per-Bin	How to compute?

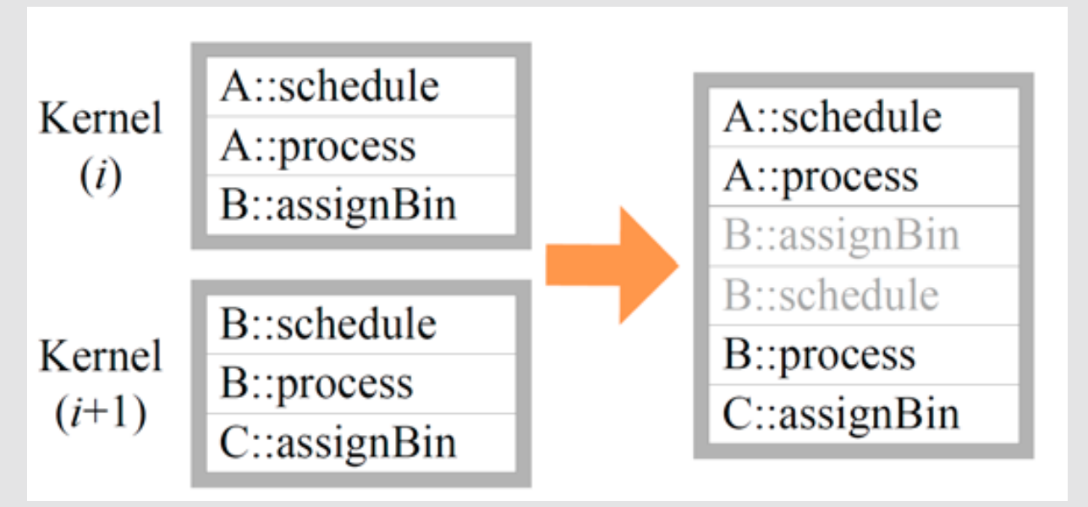
The figure to the right shows the flow of data in Piko. Primitives are first assigned to bins. These bins are then scheduled for processing on the cores of the hardware. The primitives that result are then assigned to bins in the next stage, and these steps are repeated until the primitives reach the final pipeline stage.



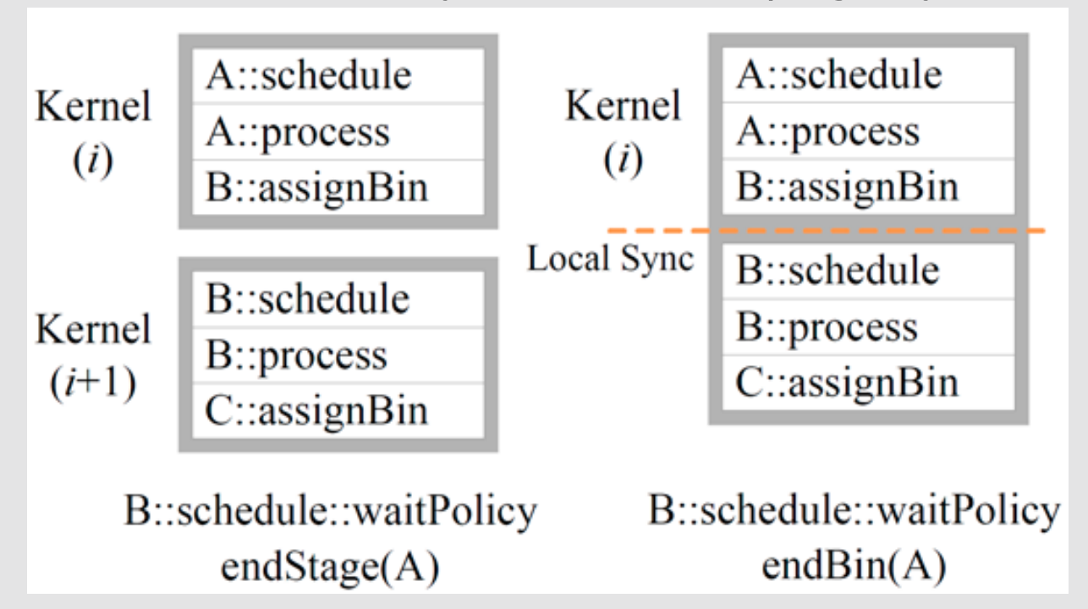
Pipeline Synthesis

Given a pipeline definition, Piko's job is to combine the logically separate stages into efficient kernels. While the *AssignBin* and *Process* phases depend only on the stage in which they reside, the *Schedule* phase is inherently pipeline- and architecture-dependent. Thus, the *Schedule* phase contains information to help Piko optimize the kernels for specific architectures. A programmer can specify whether a stage should prefer locality over load-balanced execution, and vice versa, for example. Using information about successive stages, Piko can perform various optimizations.

Kernel Fusion (exploits producer-consumer locality):



Static dependency resolution (e.g. synchronization between stages):



The results of Piko's pipeline synthesis is a kernel mapping where each phase of each stages is assigned to a kernel. Kernels can contain as few as one phase, but efficient implementation requires that kernels contain multiple phases (many times from different stages) to preserve locality. Piko also produces the order in which these kernels should be run, including which kernels should be repeated for pipelines that contain cycles.

Runtime Implementation

We implemented Piko on two different architectures, a manycore GPU (NVIDIA GTX 460) and a heterogeneous CPU / GPU processor (Intel Ivybridge Core i7-3770k). We program the GPU using CUDA and the heterogeneous processor using OpenCL, which can target both devices on the chip. Both architectures use atomics to update bins and share similar kernel code (differences are in API format).

Manycore GPU: The most important concern for the GPU is to keep all lanes of all cores busy. Thus, we prefer a load balance scheduling policy (as opposed to a locality based one) to maximize throughput. To maximize the efficiency of Piko implementations on the GPU, we make the following design decisions for runtime:

- Warp-synchronous Design: We use multiple per core, but instead of operating on one bin at a time, we assign a separate bin to each warp. This allows us to exploit locality within a single bin, but at the same time we avoid losing performance when bins do not have a large number of primitives.
- Compile-Time Customization: We use CUDA templates to specialize kernels at compile-time.

Heterogenous CPU/GPU: The CPU / GPU on a chip design allows both devices to share a common memory system and allows work to be executed on both devices simultaneously. Piko allows bins to be partitioned for each processor. We find the bin partition by profiling the kernel and adjusting the split ratio accordingly so that each device takes roughly the same amount to time to execute.
The Piko scheduler for Ivybridge (IVB) aggressively attempts to fuse pipeline stages together whenever possible. This is because this processor is better at locality based scheduling, especially when the work is done on the CPU. The CPU has a more developed cache hierarchy than its GPU counterpart (IVB L1 and L2 caches are reserved for driver use) and thus can handle producer-consumer locality more efficiently.

Results

We used the above scenes to evaluate our Piko implementation of a triangle rasterizer. Fairy Forest (left) has 174K triangles with many small and large triangles. Buddha (middle) has 1.1M very small triangles. Mecha (right) has 250K small- to medium-sized triangles. Large triangles produce more fragments during raster and thus cause more irregular parallelism in the system.

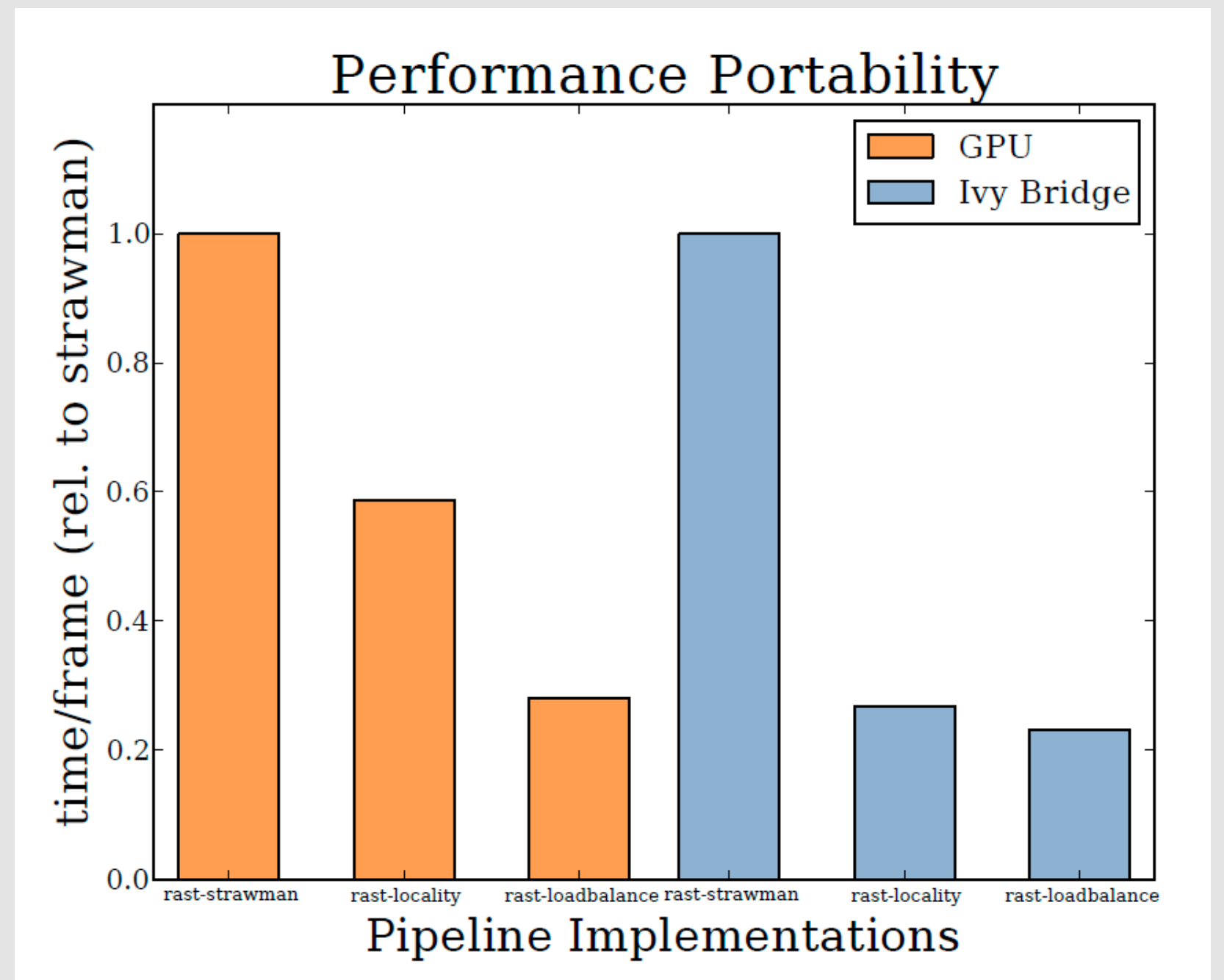
Below are the results from rendering these scenes. We tested four different Piko version of a triangle rasterizer: a strawman pipeline (no Piko optimization), freepipe (based on FreePipe*), a pipeline that prefers locality, and a pipeline that prefers load-balance.

Pipeline	Fairy Forest ms/frame	Mecha ms/frame	Buddha ms/frame
GPU rast-strawman	74.4	25.3	14.3
GPU rast-freepipe	185.3	50.6	10.6
GPU rast-locality	42.6	48.7	43.4
GPU rast-loadbalance	20.3	22.3	32.0
IVB rast-strawman	325.07	147.85	101.6
IVB rast-freepipe	603.89	298.51	86.6
IVB rast-locality	51.97	141.15	87.3
IVB rast-loadbalance	91.45	158.18	125.9

* Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. 2010. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 75-82.

Results (cont.)

The below graph shows the portability of the Piko triangle rasterizer. Due to the differences in the capabilities of the two devices, we have normalized the y-axis to the performance of the strawman pipeline. Notice how the locality-preserving rasterizer improves in relative performance when we move to an architecture with a more capable cache hierarchy.



Future Work

- Non-uniform bin sizes (which may offer better load balance)
- Spatial binning in higher dimensions (e.g. for voxel pipelines and volume rendering)
- Design a language better suited to describing Piko pipelines
- Develop more pipeline optimizations to further exploit locality and load-balance for a variety of hardware targets