Implementing the Graphics Pipeline on a Heterogeneous Multicore

Jiawen Chen Jonathan Ragan-Kelley

1 Introduction

General-purpose microprocessors (CPUs) are undergoing a radical change of direction, from traditionally narrow designs optimized for single-threaded ILP, towards exploiting far greater thread-level- and data-parallelism.

Meanwhile, more than a decade ago, real-time graphics emerged among the first successful commodity applications of data-parallel processors. In contrast to microprocessors, graphics processors (GPUs) leveraged highly-specialized, application-specific architectures. More recently, GPUs have become programmable, dominated by general-purpose processing units which look increasingly similar to highly parallel CPUs.

As the two architectures become similar, there are strong efficiency and economic advantages to convergence. Shared design, verification, and fabrication efforts produce massive economies of scale. At the same time, shared physical hardware, buses, and memories increase static hardware utilization across all applications: since there is no specialized hardware going unused, in principle, adding general-purpose transistors and bandwidth for one application improves the performance of all applications. Finally, unified resources smooth out dynamic load imbalance to dramatically increase total throughput in intensive, variable data-rate applications like graphics.

Nevertheless, key parts of the graphics pipeline still benefit significantly from fixed-function logic implementations because they are massively arithmetically parallel, can be deeply pipelined due to due to a lack of inter-computation dependencies, and require minimal control overhead.

We explore the convergence of graphics processor architectures with conventional microprocessor architectures. A key point in this design space are heterogeneous manycore processors, with graphics running as a hybrid software/hardware workload across different types of cores. We implement a basic programmable graphics pipeline on a simple, scalable multicore CPU architecture aiming towards GPU-competitive performance in graphics applications. We use graphics as a motivating workload for effectively combining general purpose processors with specialized and fixed-function elements, and understanding the more general issues in efficiently implementing complex parallel applications on such an architecture.

2 The Graphics Pipeline

We first introduce a simplified, logical version of the modern graphics pipeline (Fig. 1). We consider the pipeline to consist of four stages: vertex processing, rasterization, pixel processing, and raster operations. The application submits vertices into the pipeline, which typically contain several attributes, including position, normal, color, and other *interpolants*. Vertices are processed in a 1-in-1-out fashion by a user-programmable vertex shader. Rasterization interprets triplets of the processed vertices as triangles and interpolates vertex attributes at each pixel to output a (variable) number of *fragments*. Each fragment contains a screen space position and the value of each vertex attribute linearly interpolated across the triangle at that point. These fragments are fed to the fragment shader, which executes a user-supplied program that computes color and depth based on the interpolated information. Finally, the fragment color and depth are output to the raster operations unit (*raster ops* or *ROPs*), which tests depth against the framebuffer to determine visibility, and conditionally updates depth and blends color into the framebuffer.

Semantically, the pipeline executes *in-order*. The incoming vertex stream has obvious ordering requirements: triples of vertices are interpreted as a triangle by the rasterizer. If multiple vertex shaders execute in

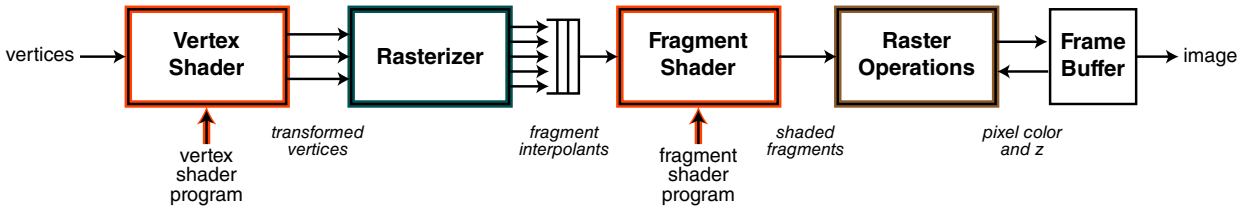

Figure 1: *A simple logical graphics pipeline. The application specifies a stream of vertices describing the 3D scene geometry (left), along with vertex and fragment shader programs and parameters, to ultimately generate an image (right). The vertex and fragment shader stages (orange) are the only programmable elements. They execute general-purpose user-defined programs, but can only read from memory. The raster operations perform read-modify-write operations against the framebuffer memory. They are configurable (brown), but not general-purpose programmable, and are the only stage which writes memory. Rasterization is not programmable (green), performs no memory access, and has extremely high arithmetic intensity.*

parallel and out of order, the rasterizer must reorder its inputs. Similarly, fragments must commit in order to the raster ops to ensure correct blending (e.g. for transparency).

3 High-level Design

We start out with a scalable, throughput-oriented manycore architecture in the style of the MIT RAW processor [Taylor et al. 2002] (Fig. 2, left).

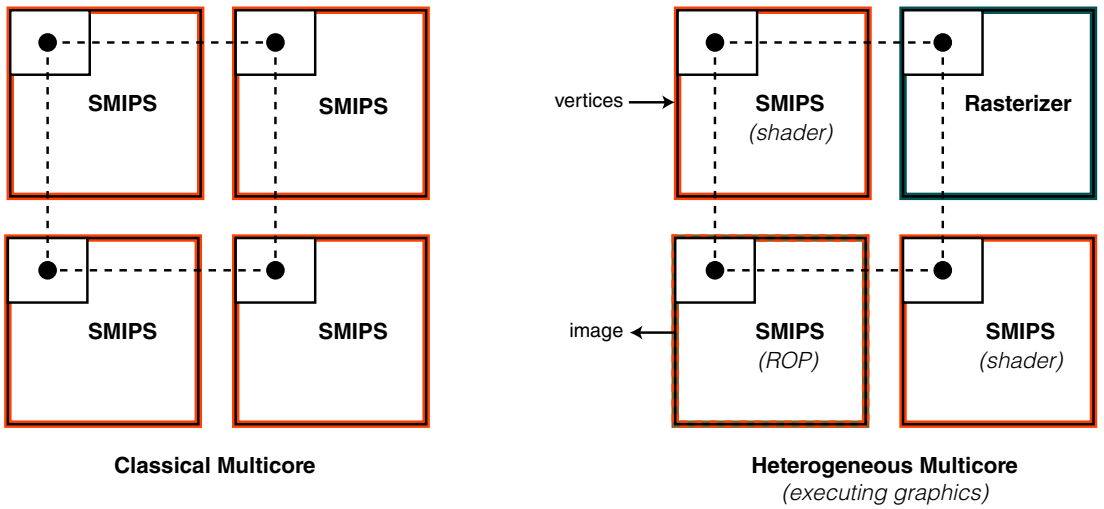

Figure 2: *Tiled chip-multiprocessors. Left: a classical multicore CPU, with many identical general-purpose processors (orange). Right: a heterogeneous multicore with one general-purpose processor replaced by a fixed-function rasterizer block (green) for efficiency, shown executing the graphics pipeline in Fig. 1. The ROP tile executes a configurable fixed-function pipeline stage in software on a general-purpose processor tile (dotted orange/brown).*

This design employs a 2D tiled network where tiles are decoupled both electrically in hardware, and semantically in the Bluespec definition, through `Connectable Get/Put` interfaces to their immediate neighbors. This enables electrical scalability by replacing global wires with local, neighbor-to-neighbor network

links, as well as transparently allowing tiles to operate in independent clock domains. It further enables tiles to be built and verified independently, and synthesized in new configurations.

Pure software graphics implementations on these architectures are known to be disproportionately dominated by the cost of some traditionally fixed-function stages, particularly rasterization [Chen et al. 2005], preventing them from even approaching GPU-competitive performance in equivalent area. We therefore explore a simple extension to such a design which replaces a general-purpose processor tile with a fixed-function rasterizer block to offload inefficient rasterization onto dense dedicated logic, in exchange for some reduction in peak general-purpose throughput. Rasterization is considered appropriate for fixed-function offload because rasterizers are small in modern processes, given the potential throughput gain, and they are never idle in modern GUI systems like Windows Vista and Mac OS X which perform all window compositing through the graphics pipeline.

4 Hardware Implementation

Our implementation builds on the existing SMIPSv2 core with a number of new hardware elements.

4.1 On-Chip Network

The packet-switched network is laid out as a 2D grid of tiles, with each tile connected to its north, east, south, and west neighbors (Fig. 3). Messages are routed according to 2D addresses, defining the row and column of the destination tile.

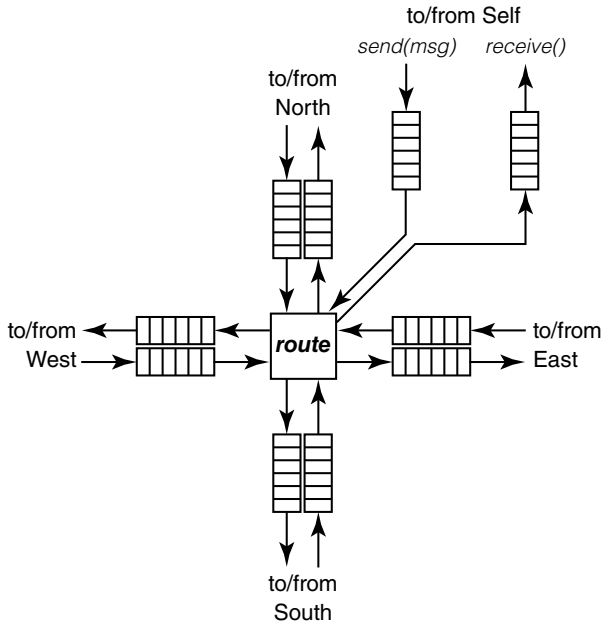

Figure 3: *Our tile network router consists of five paired send/receive queues connected by* `GetPut` *to four neighbors, and to the local tile via blocking local* `send` *and* `receive` *methods.*

Each tile implements the GetPut-based ITile interface:

```
interface IChannel#( numeric type payloadSz );
  interface Get#( NetMsg#( payloadSz ) ) send;
  interface Put#( NetMsg#( payloadSz ) ) receive;
endinterface

interface ITile#( numeric type payloadSz );
  interface Channel#( payloadSz ) north;
  interface Channel#( payloadSz ) south;
  interface Channel#( payloadSz ) east;
  interface Channel#( payloadSz ) west;
endinterface
```

4.1.1 Tile Router

Tile communications are performed by the mkNetwork module (Fig. 3) which implements the ITile interface on behalf of the containing tile, performs global tile-to-tile routing, and implements the local block-to-network interface.

The mkNetwork router has one primary family of rules, routeFromQueue, which read a packet from a given queue and push it in the appropriate direction according to its address. There is one routeFromQueue for each input queue (north, east, south, west, self) and input queues are selected in a round-robin fashion to ensure fairness and avoid deadlock. Routing destinations are selected in strict rows-then-columns order (messages are sent all the way to the correct east/west coordinate before beginning to move north/south).

The router provides two Action methods, send and receive, which directly expose the sendSelfQ and receiveSelfQ to the block. It additionally exposes sendSelfQ and receiveSelfQ full/empty checks to allow client modules to avoid blocking.

The network consists of a single, wide data channel, and an optional parallel control channel. The ITile interfaces and mkNetwork block are entirely parameterized on desired channel width, allowing simple design-space exploration and reuse with alternate network and buffer sizes. The parallel, high-priority control channel was essential to avoiding deadlock and performing efficient work scheduling in our original design featuring a parallel, unified shader processor pool, but is unnecessary in the non-unified pipeline we successfully implemented.

In our graphics pipeline, the network queues effectively implement the buffering between pipeline stages necessary to smooth out instantaneous load imbalance due to the variable-rate data amplification common in graphics. For simplicity, and given our focus on evaluating a specific application, packet size is fixed and equal to the network channel width, which we define to be large enough to fit our largest packets (post-shading vertices and pre-shading fragments).

4.1.2 Software interface

Since most of our design consists of SMIPS processor tiles running software components of the graphics pipeline, we extend the SMIPSv2 implementation to support software control of the tile router interface.

For simplicity, we implement register-mapped control using the COP2 register interface (CTC2/CFC2, MTC2/MFC2). C instructions map to the control channel, while M instructions map to the data channel. Since the network channels are much wider than the registers in the SMIPS core, we add 4 additional *waystation* registers to the core. The waystation registers (incoming and outgoing, for each channel), match the network channel widths. We take advantage of the coprocessor interface and use the register index to select individual 32-bit words in the waystation registers.

Fetch instructions (MFC2/CFC2) read 32-bit words from the waystation, while send instructions (MTC2/CTC2) write 32-bit words to the waystation. A network receive is initiated by reading from COP2 register 0, which dequeues the head of the receive queue and copies the data into the incoming waystation. Analogously,

a network send is completed by writing to register `COP2` register `0` enqueues the contents of the outgoing waystation onto the tail of the send queue and clears the outgoing waystation. (This effectively limits the usable network width to 31 words.)

The SMIPS core interacts with the network through the `INetwork` method interface. When the processor initiates a read or completes a write, it calls `INetwork.send()` and `INetwork.receive()`. `send()` and `receive()` are blocking: if the method cannot be invoked due to full/empty queues or other constraints, the `execute` rule cannot fire and the processor stalls. This effectively implements synchronous network I/O in SMIPS using only implicit Bluespec scheduling. We implement four additional `MFC0` virtual registers to perform full/empty tests on both the send and receive queues, allowing programs to avoid blocking on send or receive or perform modest flow control.

While easy to use and straightforward to implement, this interface presents a performance bottleneck, given the narrow register/data path in SMIPS relative to the network packets, requiring many instructions to perform a single useful send. Still, it is much simpler than most alternatives, and the narrow SMIPS datapath generally presents throughput challenges for graphics data even outside the network interaction, making this a deeper and more general problem which is beyond the initial scope of this project.

4.1.3 Synthesis

We synthesized our network interface as part of both the SMIPS and Rasterizer tiles, which we synthesized independently of one another. Our initial design included long (ten element) queues to allow relatively deep buffering to smooth load spikes through the pipeline. However, because of the width of our the network, this generated very large register banks which Design Compiler was unable to synthesize on a 32-bit machine with the maximum of 2gb installed RAM supported by Athena. We could only consistently synthesize our tiles before running out of memory with smaller, 3-element FIFOs.

These were nevertheless relatively large. Each router contains five send and five receive queues, each network-width wide, creating $10 \times width \times n$ bits of data registers for a single router with n-element queues. These queues are $0.183mm^2$ post-place and route, requiring just under $1.9mm^2$ for a single tile router. A real-world design would likely use SRAM blocks to support large buffers more efficiently, assuming latency-insensitive applications like graphics. Still, though large, the network is fully registered at all interfaces only a minimum of combinational logic, which means it is never on the critical path.

4.1.4 Analysis and Future Work

Our design employs a very simple, explicitly-exposed point-to-point network. It consumes significant area due to its very wide queues, but is under-utilized due to the high overhead of filling packets in software, one 32-bit word at a time.

An improved design would significantly reduce the network queue area by narrowing the physical network, accommodating large logical packets via multi-cycle sends and receives. At the same time, a high-performance design should decouple large network transactions from the narrow SMIPS data path by making communication asynchronous, likely through a memory-centric or memory-mapped processor-network interface. Such an interface would remove the overhead of explicit network word-by-word copy instructions and allow efficient manipulation of large block transfers independent of the primary register and data path size.

The obvious extreme of such a design would move all the way to a pure shared memory model. Such an architecture would allow not just the communications, but the communication models/structures—essentially the logical pipeline layout and interconnections, currently limited to simple queues, implemented in hardware in the network—to become dynamic and move entirely to software. (Explicit software-controlled network transactions could still be approximated via prefetch instructions.) This is potentially very powerful, as arbitrary communication structures could be dynamically created, with different ordering and other semantics, independent of the underlying hardware. However, it requires that performance and scalability can be maintained in the face of the coherence overhead which has limited traditional (multi-chip) shared memory multiprocessors.

4.2 SMIPSv2 Extensions

We extend our base SMIPSv2 core with a variety of new standard and coprocessor instructions and other features to support a multicore implementation, as well as essential graphics operations and better software debugging.

Multiple Programs Given multiple tiles communicating on the network, we need to be able to load distinct code (`vmh` files) on the different processors, depending on their roles.

For simplicity, we do not support shared memory between processor tiles, but instead implement a pure message-passing architecture where the tile network is used explicitly for all communication. In practice, this means that each tile still has its own, independent cache and variable-latency "memory" (implemented on top of a register file in the test bench, as in the BSV SMIPSv2 reference implementation). We implement multiple images by passing a `String` filename argument to each `mkSmipsTile` module. The filename is propagated down to the memory system, which gets loads the appropriate `vmh` during initialization.

Multiplication We extend the baseline SMIPSv2 processor with a `MUL` instruction. This allows us to use the `*` operator in C and execute the compiled code. We trivially implement the ALU portion of MUL in hardware using Bluespec's single-cycle `*` operator. In addition, since SMIPSv3 specifies that MUL returns a 64-bit result in separate `HI` and `LO` registers, we also add these registers to the processor, along with the `MFHI` and `MFLO` instructions to access them. Finally, we modify the stall function and writeback rule to handle the hazards introduced by the new instructions.

We chose a single-cycle multiplier due to its simplicity: it is a combinational circuit and does not require any changes to the existing ALU. However, after synthesis, it turned out to be extremely inefficient. Due to the multiplier's long combinational path, the multiplier became the critical path in our design, with a clock period of over 35 ns after place and route (compared to the 12ns result from Lab 3). The critical path went from the writeback queue, through the register file, through the ALU's multiplier, and back to the writeback queue (the writeback queue is a bypass FIFO). The multiplier's occupies about 147,815 um^2, which is extremely larged compared to the 2,242 um^2 occupied by a 32-bit adder. We suspect that this is due to Bluespec's interpretation of the $*$ operator. SMIPS specifies that $*$ takes in two 32-bit operands and returns a 64-bit product. However, Bluespec's $*$ only returns a 32-bit product on 32-bit inputs. Our implementation uses `signExtend()` to implement 64-bit multiplication, which probably synthesizes into a 64-bit multiplier. A more carefully written multiplier would only occupy half the area.

We believe a real SMIPS design that supports MUL and DIV should include separate, fully pipelined, multi-cycle ALUs for complex operations. Since both MUL and DIV can be trivially pipelined, a multi-cycle ALU would significantly decrease the critical path while still achieving good throughput.

Mini Standard I/O Since our design contains a number of tiles, debugging using the simulator's `$display` system call is impractical. To facilitate debugging, we augment SMIPS with additional I/O instructions, where each processor writes to a separate file.

We use Bluespec's C foreign function interface to import `openFile()`, `writeInt()`, `writeChar()`, and `writeFixed()` as Action functions. When each processor resets, an initialization rule calls `openFile()`, and passes the processor's address as the argument. `openFile()` uses `fopen()` to open a file stream for the processor, the file handle is returned to Bluespec as a `Bit#(32)`, which we store in a register. We expose the other functions as SMIPS instructions by register mapping them to `MTC0`, where the coprocessor destination register selects the function. To expose the mini standard I/O to software, we wrap them as C functions using inline assembly.

Finally, we also provide an `exit( exitcode )` function, that maps directly to `$finish( exitcode )`. We found it useful to allow one processor to unilaterally terminate the simulation with a condition code.

Fast Image Scanout To generate the final image, we have to retrieve the data from the simulator. Initially, we implemented the final scanout completely in software using our debug I/O interface. Once

ROP determined there were no more fragments, it iterates over all the pixels in the frame buffer and calls `writeFixed()` to write it to a file. However, even at a resolution of 200×200, the frame buffer occupies a fair amount of memory (640 kB – enough for anyone!) and simulation was prohibitively expensive. Instead, we extend the SMIPS memory system with a fast scanout mechanism.

We hijack 3 additional coprocessor registers in `MTC0`. To initiate a scanout, the user first writes the image width and height into coprocessor registers 7 and 8. He then writes the framebuffer start address into coprocessor register 9, which pushes a token through the caches to the memory main memory. The main memory has a small FSM that iterates over each pixel and writes it out to the processor's output stream. Since scanout is the final step in our simulation, it simply calls `$finish()` once scanout is complete.

4.3 Rasterizer

Our rasterizer directly implements the `ITile` interface, taking triangles as input and producing fragment interpolants as output. It computes visibility and attribute interpolation using the *homogeneous rasterization* algorithm [Olano and Greer 1997]. Homogeneous rasterization consists of three logical stages: primitive assembly, triangle setup, and interpolation. Our implementation begins as a corresponding three-stage multicycle implementation, which we refine into a three-stage pipeline connected by queues, with each stage iteratively subdivided further into multiple cycles for speed (Fig. 4).

Primitive Assembly Primitive assembly (Fig. 4, left) reads the ordered vertex stream from the Rasterizer input queue and assembles triples of vertices (triangles) for setup. It is implemented as a simple three-stage state machine which reads from the network and pushes a new triangle down the pipeline every third step.

Triangle Setup Setup (Fig. 4, center) begins by pulling off one triangle from the triangle queue and constructing a 3×3 *vertex matrix* consisting of the three vertex positions. The vertex matrix is then *inverted* using fixed point arithmetic. (A singular matrix indicates that the triangle is invisible, and our rasterizer discards triangles whose matrices are within a small epsilon of singular.)

The entries in the vertex matrix inverse are used to compute a set of *interpolators*: one for each scalar attribute of the incoming vertex. An interpolator is a 3D vector, and it is constructed by a single matrix-vector product. There are four required interpolators, which need special handling: the three corresponding to the edges of the triangle and one corresponding to depth. All other interpolators are computed identically.

Triangle setup also independently computes the screen space bounding box of the triangle directly from the vertex positions, and feeds it forward to the interpolation stage.

Interpolation The interpolator (Fig. 4, right) computes values for each interpolant at each pixel inside the triangle's bounding box. It first computes visibility (the inside/outside-triangle test) by using the z-interpolator to compute z, and using z to interpolate the three edge equations. If all edge equations evaluate to greater than 1, then the pixel is inside the triangle. If the pixel is inside, it uses z to interpolate each attribute. Finally, it pushes the fragment onto the network.

In our implementation, "computing z" involves a single three-component dot product and a reciprocal operation. "Interpolating" is computed as a three-component dot product between the current pixel coordinate and the corresponding interpolator, scaled by z.

Interpolation is trivially data-parallel: every pixel is independent and we can instantiate a hardware unit for every pixel. Additionally, after the z and edge testing, each attribute can be interpolated independently.

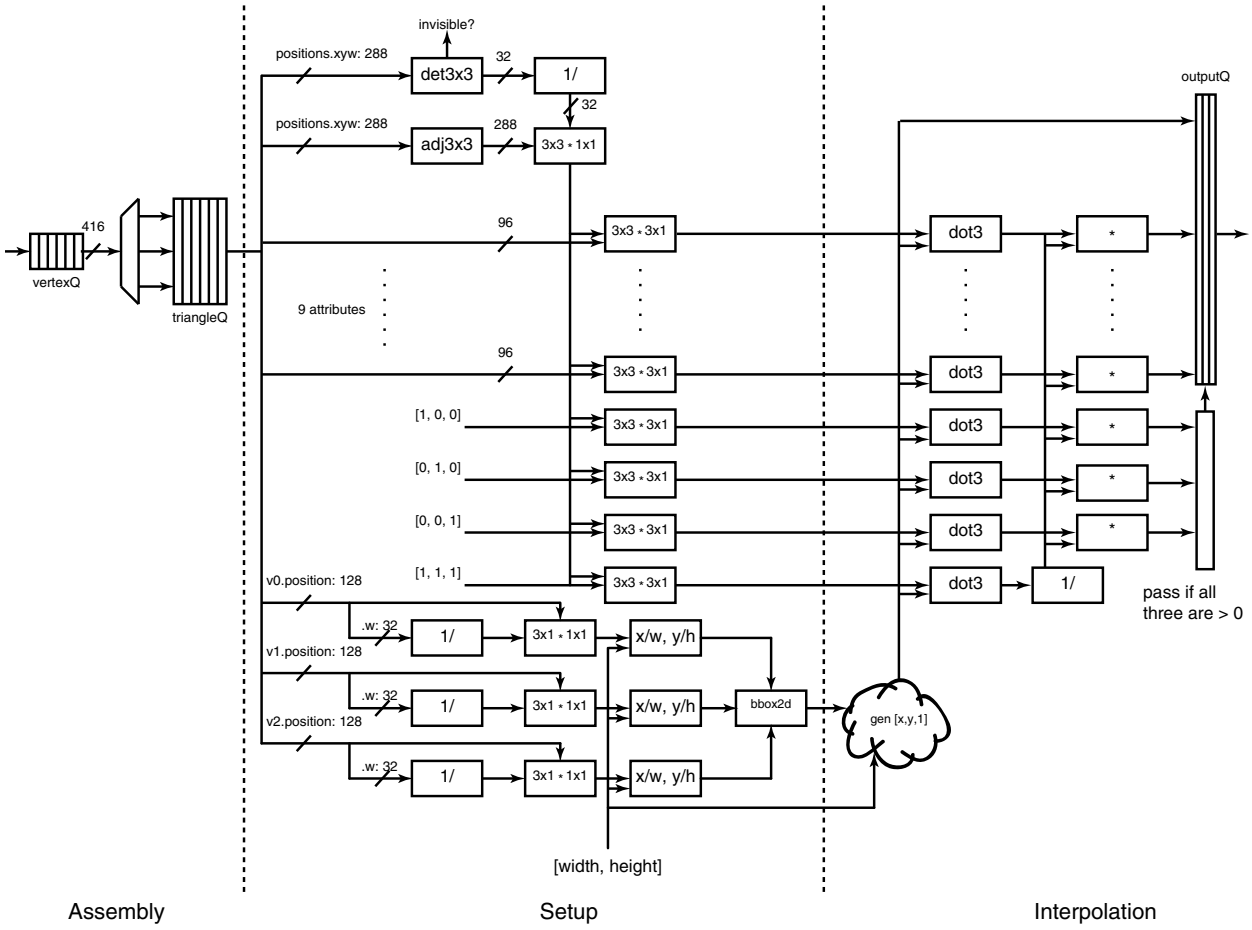

Figure 4: *Our rasterizer consists of three logical stages: assembly, setup, and interpolation. Assembly groups individual vertices into whole triangles ready for setup. Setup performs a sequence of 3D homogeneous linear algebra operations to construct a* triangle matrix*, along with* interpolator *vectors for each vertex attribute and each of the three edge equations. This includes a reciprocal operation per-edge. Finally, interpolation computes attribute values and edge equations at every pixel as the dot product of each attribute interpolator with the current pixel coordinate, scaled by the perspective. (This requires an additional division per-pixel to compute the* $1/w$ *perspective projection term for perspective-correct interpolation.)*

4.3.1 Physical implementation

After achieving correctness, we iteratively synthesized and refined our rasterizer block (Fig. 8). From the start, the rasterizer tile is extremely wide and logic-heavy, generating a large synthesized design. Throughout refinement, our rasterizer tile remained 16.75 mm^2 and 1.78 M cells, of which 1.8 mm^2 is the tile network, and the rest is entirely logic and internal pipeline registers.

Three-stage pipeline We first pipelined our rasterizer according to the three-stage logical pipeline described above. Each stage communicates via queues, and the lack of data hazards inherent in the strict feed-forward pipeline allows full rule concurrency. This design exploits the substantial arithmetic parallelism inherent in the 3D linear algebra of setup and interpolation, as well as across the many parallel interpolants.

It does not exploit parallelism across pixels during interpolation, but nevertheless creates a massively arithmetically intense block capable of easily saturating the rest of our SMIPS-based pipeline.

This design, however, still includes extremely long combinational paths through the many single-cycle multipliers and adders synthesized by Bluespec, yielding a slow design only about to sustain a 50 ns clock after place-and-route, with a critical path through the setup stage.

Pipelined setup and interpolation We then further pipelined the setup stage after the determinant and adjoint calculations. Next, based on the predicted critical path after synthesis, we pipelined the interpolation stage after the initial dot product. This design sustained a post-place-and-route clock of 25 ns while marginally increasing area by 0.1%.

Clearly, this can be reduced by pipelining the design much further. The pure feed-forward pipeline has no inter-element dependence, allowing almost arbitrarily deep pipelining with no stall penalty or other throughput overhead. As it stands, however, our design is already fairly complex code and logic, quickly becomes much more complex as stages are added and math gets more deeply pipelined. Furthermore, this is indicative of a general problem we face: to achieve reasonable clock speeds, we need an efficient, pipelined multiplier rather than the single-cycle combinational unit synthesized by Bluespec/Verilog. Without an efficient, pipelined integer multiplier, our rasterizer cannot improve much further, as evidenced by the 35 ns cycle time through the SMIPS MUL stage.

Either way, the rasterizer could also be optimized by tuning the internal fixed-point representation to remove bits wherever possible. Our design uses the same 32-bit (17 bit signed magnitude, 15 bit fraction) as our C libraries and the rest of our pipeline. However, since most of rasterization occurs in normalized clip coordinates, most magnitude bits go completely unused. Significantly reducing the number of bits would dramatically reduce the size and depth of the ALU trees (especially multiply), as well as the size of the pipeline registers and FIFOs which must carry very wide data elements through the pipeline. This could dramatically reduce both area and cycle time.

Hardware Divider Homogeneous rasterization requires five dividers: 4 per triangle during setup, and 1 per pixel during interpolation (see Figure 4). We implement a simple unsigned 64-bit multi-cycle hardware divider that implements the "radix two" division algorithm [6.004 Lecture Notes, 2002].

The hardware divider is initialized by registering its inputs with `initDivide()`. The second stage shifts a trial divisor left until it is greater than or equal to the dividend. Finally, it determines each bit of the quotient by successively testing shifted versions of the trial divisor and determining whether the trial divisor can be subtracted from the dividend without producing a negative remainder. The result is registered and can be read by a `result()` method. Each stage is implemented as a finite state machine within the module. To implement signed division, we also register the sign of the input operands. We perform unsigned division and restore the sign in `result()`.

4.4 Software Stages

We implement the vertex shader, fragment shader, and raster operations stages of the graphics pipeline in software on SMIPS. In this section, we detail the functionality and our implementation of each stage.

4.4.1 Vertex Shader

As mentioned in section 2, the vertex shader operates in a one-in-one-out fashion over *vertices*. Data is input to the vertex shader at two frequencies. *Uniforms* variables are specified at most a few times per frame; for example, camera position, light position, character bone orientations. *Varying* vertices are the actually data over which the shader executes. A vertex is defined by a number of *attributes*, such as position, normal, color, texture coordinate, and anything else the user supplies. A vertex shader program is a user-supplied program that processes the input vertex into a format appropriate for rasterization. For example, input vertices are typically stored in *object space*, when the user moves the camera, the vertex shader is updated with the new camera matrix and transforms each vertex into *clip space*.

For our project, the vertex shader is also our input unit. Normally, vertices are streamed in from the CPU over a bus. In our design, the vertex shader directly reads its input vertices by iterating over an array in memory. Our input data consists of a list of vertices with positions and normals in object space and a camera matrix. For each vertex, the vertex shader multiplies its position by a 4×4 matrix to perform perspective projection into the current view, producing a clip space position.

We implemented two vertex shaders: a debug shader, which requires that every input vertex has a color attribute, and a *per-vertex lighting* shader that computes the color of a mesh using the Lambertian material model. The Lambertian material model states that the color of the vertex is determined by cosine of the angle between the normal vector and the direction to the light source. To compute the angle between the normal and the light direction, we first find the *unit* vector between the light source and the vertex by subtracting and *normalizing* the difference. Then we also normalize the vertex's normal vector and find the cosine of the angle by computing a dot product. Finally, after computing the color, the vertex shader performs a network send to push the vertex to the rasterizer.

4.4.2 Fragment Shader

A fragment shader operates in a one-in-one-out fashion similar to the vertex shader. Like the vertex shader, data enters the fragment shader at two frequencies: *uniforms* and *varying* fragments. Uniforms are specified at most a few times per frame and specify shading parameters such as the position and strength of lights. Fragments are the results of rasterization and like vertices, carry a number of *interpolants*. Interpolants are the linearly interpolated values of vertex attributes across a triangle. For example, given a triangle where each vertex's color and texture coordinate have been specified by the vertex shader, each fragment would have a color and texture coordinate that is a linear blend of the per-vertex values. Fragment shaders output *post-shading fragments*, which contain only a 3D position (screen location and depth), and a 4D color (red, green, blue, and opacity).

We implemented two fragment shaders: a passthrough, which simply sets the output color to be the rasterizer interpolated color, and a *per-pixel lighting* shader that evaluates a Lambertian lighting model at each pixel. Note that per-pixel lighting is significantly more expensive, and produces much higher quality images than per-vertex lighting. A per-vertex lighting shader computes the color at each vertex and linearly interpolates the resulting color values. Per-pixel lighting uses the rasterizer to interpolate the per-vertex lighting *parameters*, and evaluates the lighting model at each pixel.

For our simulation results, we were unable to evaluate the per-pixel lighting model. Per-pixel lighting performs the same operations as per-vertex lighting (vector subtraction and a reciprocal square root), but at a significantly higher frequency (each triangle typically occupies dozens of pixels). It would have taken at least a day to simulate on our 200×200 images. All of our results use the passthrough shader.

4.4.3 Raster Operations

For our simple graphics pipeline, raster operations consists of only two operations: depth testing and compositing, which we implement completely in software. The ROP unit begins by pulling a shaded fragment off its input queue. The fragment's screen space position determines its address in memory. First, we perform depth testing: which issues a memory request into the Z-buffer for the depth of the previous fragment at that position, compare the returned value, and if the new fragment is closer, update the Z-buffer (otherwise, discard). If the new fragment passes the depth test, issue another memory request to retrieve the previous fragment's color. Blend the incoming and existing color together using linear interpolation, and update the color buffer. For all our results, we used a framebuffer size of 200×200 pixels due to the cost of simulation.

Software Cache Flush Section 4.2, describes how we extend the SMIPS processor's memory system to do a fast image scanout. However, due to the fact that the data cache is not write-through, parts of the frame buffer may still be in the cache and not flushed to main memory. We provide a small software cache flush instruction wrapped in inline assembly to allow the ROP to flush its cache during frame scanout. Since we know both the size of the cache, as well as the memory address where the *instruction* segment begins, our

cache flush function simply reads memory starting at the beginning of the code segment. Since the SMIPS data cache is direct mapped, a linear memory scan is sufficient to evict every cache line. To prevent the compiler from optimizing away the cache flush entirely, we perform a sum and return it to the caller.

5 Support Software

5.1 Software Libraries

We implement high-level support libraries to make programming easier and to facilitate debugging. The libraries allow most high-level operations such as shaders and network communications to be written in C without touching low level assembly.

High-level Network Interface We provide a simple interface for communicating over the on-chip network. We expose blocking `send()` and `receive()` functions and non-blocking `sendQueueIsFull()` and `receiveQueueIsEmpty()`, which let the program query whether a `send()` or `receive()` will block. `send()` and `receive()` come in four variants: control vs. data network, and to/from SMIPS vs. hardware. All variants of `send()` take as parameters a target address, a message encoded as a `void *`, and the number of 32-bit words in the message. `receive()` takes as parameters a `void *` buffer, and the size of the buffer. The only difference between the SMIPS and hardware variants of the functions is the byte ordering. Our SMIPS waystation register is (at first unintentionally) little-endian, while Bluespec structs are encoded as big-endian. In debugging the issue, we quickly patched this in our C library: in order for structs written in C to map directly onto structs in Bluespec, we swizzle the words when communicating between software and hardware modules.

The functions themselves are a written in inline assembly that directly wrap the SMIPSv2 extensions described in Section 4.2. For simplicity, we always send and receive a full packet. For `send()`, we take the input message, copy it into a buffer of maximum packet size (swizzling the words if necessary), and call the corresponding coprocessor instructions. `receive()` works analogously: we allocate a buffer of the maximum packet size, call the coprocessor instructions to retrieve the full packet, and then copy into the user-supplied buffer as many words as the buffer has.

Fixed-Point Math Library Since SMIPSv2 does not have a floating point unit, we implement fixed-point arithmetic in a C library. Our library is compatible with Bluespec's FixedPoint typeclass, which makes it easy to communicate between software and hardware modules. For example, the Vertex Shader (running in SMIPS) sends shaded vertices, which are structs of structs of fixed-point quantities, to the Rasterizer (a pure hardware module). In the Rasterizer code, we can declare the shaded vertex struct the exact same way as in C and the two will be bit compatible.

On top of the basic fixed point operators, we implement a number of functions used in graphics, including:

1. Division: We did not include our multi-cycle hardware divider into our SMIPS core, as that would make the pipeline more complicated. Instead, we implement division in software.
2. Square Root: Many graphics operations require a square root (e.g., normalizing a vector to unit length). Using fixed-point division, we implement an iterative square root approximation algorithm.
3. Vector and Matrix Data Types: `fixed2, fixed3, fixed4, fixed2x2, fixed3x3, fixed4x4`
4. Vector and Matrix Arithmetic: dot product, matrix-vector multiplication, matrix-matrix multiplication, matrix inverse.

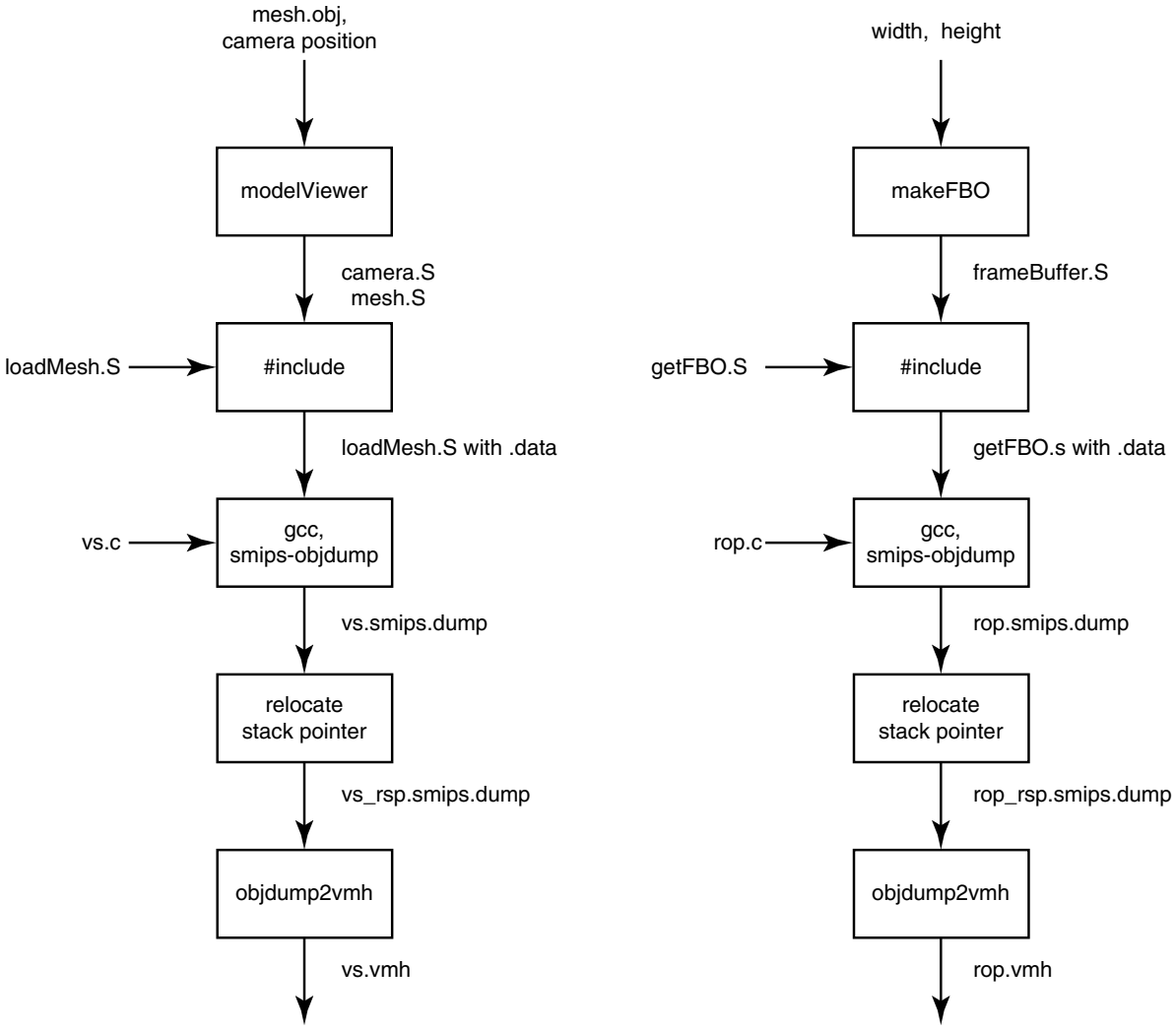

Figure 5: *Extended SMIPS toolchain. Left: vertex shader toolchain. Right: raster operations toolchain.*

5.2 Toolchain

We render 3D meshes using our processor (through BlueSim) and a reference OpenGL implementation to verify the correctness of our design. In order to vary the input data and parameters, we supplement the existing toolchain with a number of utilities. The tool flow is as follows (also see Figure 5):

1. Select an input mesh. Use `ModelViewer` to render the mesh in OpenGL and interactively choose camera parameters. ModelViewer outputs a vertex array for the model, and the camera view matrix and position vectors as raw data in MIPS assembly.

2. Write a vertex shader `vs.c` and compile it with the output from ModelViewer to produce `vs.vmh`, which can be directly loaded into the simulator.

3. Select an output resolution. Run `makeFBO` to allocate and initialize the corresponding framebuffer and compile it with `rop.c` to produce `rop.vmh`.

4. Write a fragment shader `fs.c` and compile it into `vs.vmh`.

5. Run `relocateSP` to repair the object dumps.
6. Execute the sim.
7. Use `txt2ppm` to produce the file image.

We detail each of the tools below.

5.2.1 ModelViewer

We use ModelViewer to load a mesh in the standard .obj file format and interactively select a camera viewpoint. Once the input mesh and camera parameters are determined, ModelViewer dumps out the data into `mesh.S` and `camera.S` as `.word` statements in the `.data` section. `mesh.S` and `camera.S` are included inside a `loadMesh.S` file, which has functions that return pointers to the data and parameters to the main vertex shader. When we load a vertex shader, we link it against `loadMesh.S` to produce `vs.vmh`.

5.2.2 makeFBO

Our toolchain lets the user specify an output image resolution. Since the Z-buffer needs to be initialized to a large number before rendering begins, and simulating the initialization process is prohibitively time consuming, we add a `makeFBO` utility to the toolchain. When the user changes the image resolution, he runs `makeFBO` which returns a `frameBuffer.S` assembly file with an appropriately initialized `.data` segment. `frameBuffer.S` is included in `getFBO.S`, which has functions that return pointers to the data. `getFBO.S` is then linked with the main ROP program to produce the final `rop.vmh` file.

5.2.3 relocateSP

Due to the large data sections produced by ModelViewer and `makeFBO`, and the fact that gcc-smips initializes the stack pointer to a low address, the stack tends to clobber our model data and framebuffer. `relocateSP` is a small script that takes the disassembled object dump, searches for the largest used address (i.e., the last line in the file), and relocates the stack pointer just beyond the

5.2.4 txt2ppm

As described earlier, the raster operations unit's memory system performs a scanout to a file once it receives a flush message. `txt2ppm` parses the file and produces a final PPM image, viewable with standard software (e.g. `display`) and which can be converted to PNG and other common formats (via `convert`).

6 End-to-End Results

In addition to unit tests of the various blocks, tiles, and software components, we rendered three main test scenes, described and shown in Figure 6, through the entire pipeline.

Performance It is difficult to analyze highly dynamic load balance over hundreds of millions of simulated cycles to determine efficiency in a complex machine like this. To capture a first-order approximation of utilization and bottlenecks, we added additional rules to our SMIPS and rasterizer tiles to track whenever a rule should have fired, but blocked on either the input or the output network queue. Because these rules block on the network based on implicit Bluespec guards, we wish to detect whenever they should have fired but were blocked on the network. We do so by inserting an additional profiling rule with an identical guard, but which is marked never to fire when the original rule actually fires, using Bluespec `preempts` declarations. Measuring the fraction of the time each tile is blocked on its input or output queues provides a reasonable picture of bottlenecks and load balance in the pipeline as a whole.

Figure 6: *Three images rendered entirely through our pipeline in simulation. The rasterizer interpolation test (left) contains two triangles with colors assigned to each vertex. The teapot scene (center) consists of 633 triangles, lit by a single light with per-vertex lambertian shading. The head scene (right) consists of 1,214 triangles, also lit by a single light with per-vertex lambertian shading All scenes load the vertex array and perform the view transformation in the vertex shader, and all are rendered at* 200×200 *resolution.*

scene	total cycles	Vert. shader blocked cycles	Rasterizer blocked cycles	Frag. shader blocked cycles	Raster Op blocked cycles
Teapot	328.9 M	0	10.9 M	20.8 M	23.5 M
Head	633.6 M	0	16.0 M	30.8 M	38.4 M

Figure 7: *Simulated performance in cycles for the benchmark scenes in Figure 6. Total cycle time includes a full software cache flush and image scanout from memory, roughly 1 M cycles.*

Performance and cumulative blocked cycles for the teapot and head benchmark scenes are given in Figure 7. For these scenes, we find that each stage is blocked on input far more than output, and each stage blocks more than the last. The vertex shader stage never blocks because it performs significantly more math (reciprocal SQRT) than either of the other SMIPS stages. Nevertheless, total time blocked is less than 10% of execution time, suggesting surprisingly balanced execution. First, although it performs much more arithmetic than the later software stages, the vertex shader performs this computation on significantly lower-frequency data elements (vertices vs. pixels). Second, the vertex shader draws its input vertex array directly from local memory, rather than over the network interface, while the "passthrough" fragment shader must perform a full read of a 30 word wide packet at a cost of at least 100 cycles to receive its input. Deeper queues could likely improve this remaining 10% by reducing instantaneous imbalance. Moderate balance is likely achieved for a number of reasons.

block	gates	area	cycle time
Extended SMIPS tile	1.45 M	13.68 mm^2	35 ns
3-stage Rasterizer	1.78 M	16.77 mm^2	50 ns
Fully-pipelined Rasterizer	1.78 M	16.79 mm^2	25 ns

Figure 8: *Post-place-and-route area and cycle time for our two major tiles, SMIPS and rasterizer, including multiple, increasingly-pipelined variants of the rasterizer tile. Note that the channel interface between tiles can easily allow separate clock domains for the different tiles, depending on both achievable cycle time and pipeline load balance, using Bluespec sync FIFOs. In our implementation, increasing the relative clock of the rasterizer would rarely increase total throughput, as the pipeline is bottlenecked in the SMIPS shader cores. All results are achieved using zero "effort" and other relaxed constraints to reduce memory usage due to difficulty synthesizing these monolithic tiles (particularly the rasterizer) on 32-bit workstations.*

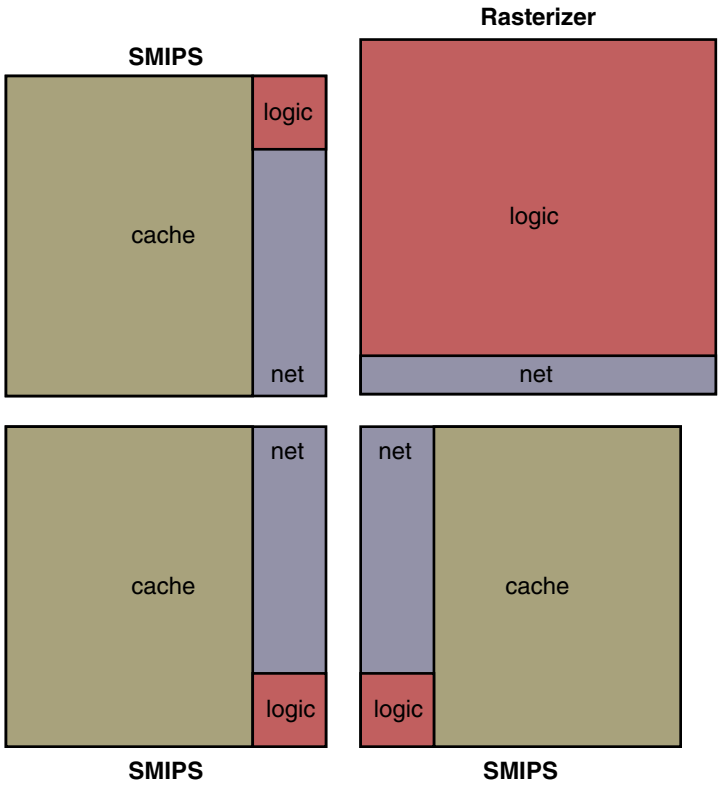

Figure 9: *A rough symbolic floorplan of our 4-tile heterogeneous processor. Individual tiles were synthesized separately, without intra-tile floorplanning, but all block and sub-block areas are precisely to scale. Not that while the rasterizer and SMIPS tiles are of comparable area, the ratio of logic to memory is radically different, due to the arithmetic density of the rasterizer.*

Synthesis Figure 8 shows the results of synthesizing our two primary tiles, the SMIPS tile and the rasterizer, including the large tile network interface with three-element queues in each direction. As we saw earlier, cycle times are ultimately limited by the use of automatically generated single-cycle ALUs (particularly for multiplies).

Finally, Figure 9 shows a scale rendering of the major components of our 4-tile heterogeneous processor. Note that the tiles are naturally roughly evenly sized, supporting effective 2D packing, but that they have extremely disproportionate ratios of logic to memory. Given that the immense rasterizer logic block consists largely of fixed point ALUs, while the SMIPS logic block includes the entire main processor pipeline and only a tiny handful of ALUs, it is easy to see how rasterization can cripple a naïve, software-only graphics pipeline.

7 Lessons and Future Work

Many of the challenges and interesting experiences in this project were with system-level issues.

Software is Powerful During implementation we quickly found that true C programmability, with full-featured support libraries (for network i/o, math, debug printing, etc.) makes this style of pipeline design amazingly flexible and easy-to-manipulate. We frequently found ourselves modifying and patching behavior in C or in our libraries, rather than in hardware (like with the network byte order issue, where a quick

software fix stuck for the duration of the project). Even in the relatively anemic environment provided by SMIPSv2 with no OS or memory management, this is a testament to the power of using general-purpose (CPU) tiles vs. hardware pipeline stages in this sort of design.

Addressing Static/hard-coded network addresses make design changes and exploration painful. This is even true on the software side, but is particularly problematic in the hardware blocks. Even with organized preprocessor definitions to easily change the layout in a single location, rebuilding the full hardware design takes at least ten minutes. Adding a layer of indirection via a programmable simple routing table or other mechanism would be useful for reconfigurability during initial exploration, for reuse between designs, and for dynamic configuration of a single design under changing workloads.

Flushing the Pipeline Flushing the pipeline is essential before reading out final framebuffer values. While test cases seemingly produce correct results, more complex scenes will mysteriously lose data without a proper flush. We implement flush with a tagged "end-of-stream" barrier packet which moves through the data stream of each pipeline stage, only advancing to the next in-order after all prior elements have completed. This is relatively straightforward in the context of hardware stages like the rasterizer. However, in an SMIPS tile stage, "flushing" further requires draining the cache to main memory. This becomes significantly more complex in a parallel pipeline implementation, with multiple independent units at each stage.

Parallelism We put significant additional low-level design and implementation work into a parallel, unified shader design with significantly more tiles and better balance. We chose to focus our time on correctness, completeness, and some hardware exploration in the rasterizer, but we still found this work interesting. For general interest and to inspire future work, we hope to post some of our notes online in the next few days at http://people.csail.mit.edu/jrk/6.375.

Citations

Jiawen Chen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli, Frédo Durand. A Reconfigurable Architecture for Load-Balanced Rendering. In *Graphics Hardware 2005*.

Marc Olano and Trey Greer. Triangle scan conversion using 2D homogeneous coordinates. In *HWWS '97*.

Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen Matt Frank, Saman Amarasinghe and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, Mar/Apr 2002.

6.004 Lecture Notes, Spring 2002.