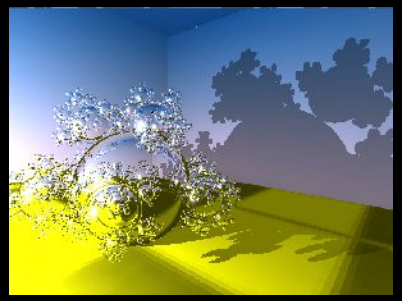
RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing

Sven Woop Jörg Schmittler Philipp Slusallek

Computer Graphics Lab
Saarland University, Germany



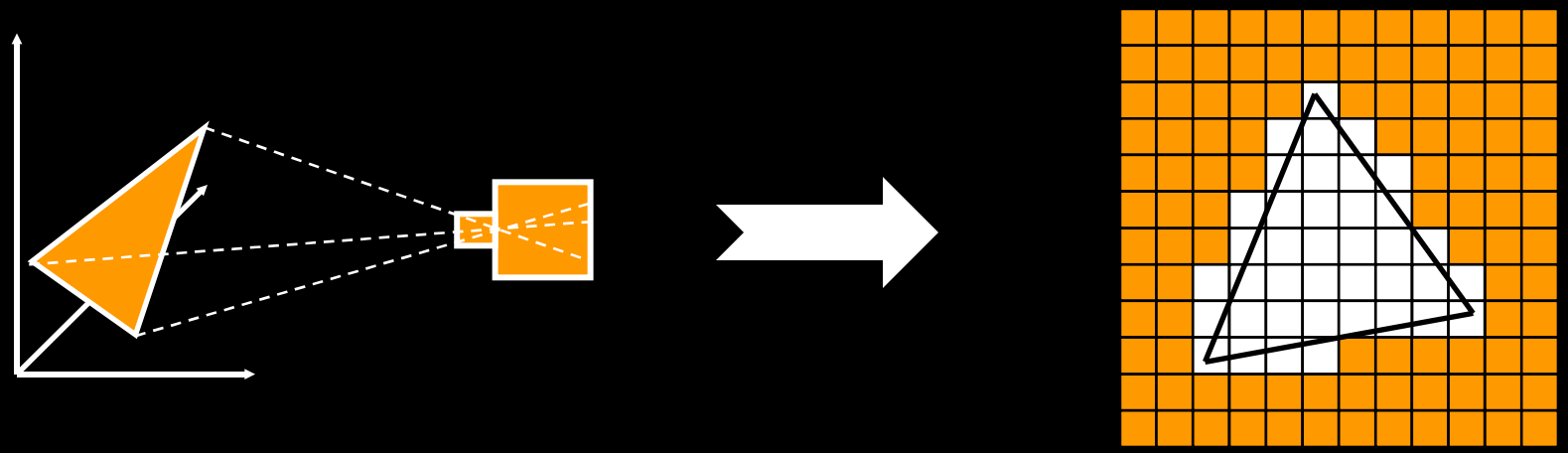

Motivation

„Some argue that in the very long term, rendering may best be solved by some variant of ray tracing, in which huge numbers of rays sample the environment for the eye's view of each frame.

And there will also be colonies on Mars, underwater cities, and personal jet packs.“

„Real-Time Rendering“, 1999, 1st edition, page 391

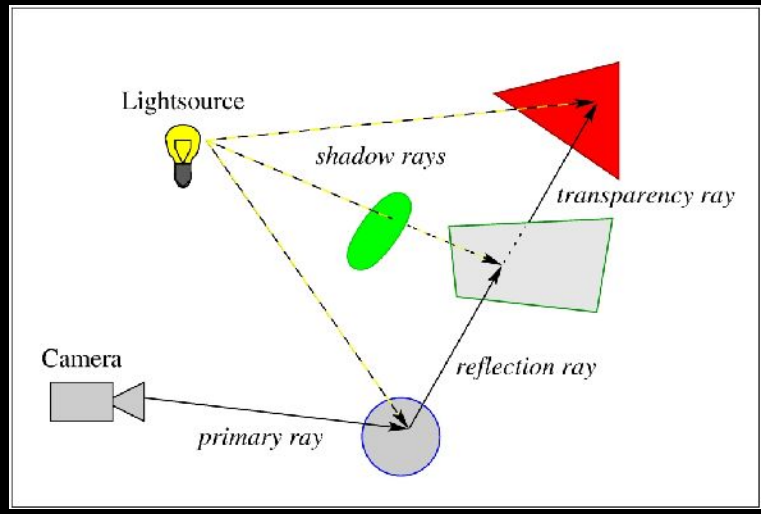
Rasterization



- Fundamental Operation: Project isolated triangles
  - No global access to the scene
- All interesting visual effects need 2+ triangles
  - Shadows, reflection, global illumination, …
  - Requires multiple passes & approximations, has many issues

Ray Tracing

- Fundamental Operation: Trace a Ray
  - Global scene access
  - Individual rays in O(log N)
  - Flexibility in space and time
  - Demand driven combination of visual effects
  - Physical light simulation
  - Embarrassingly parallel



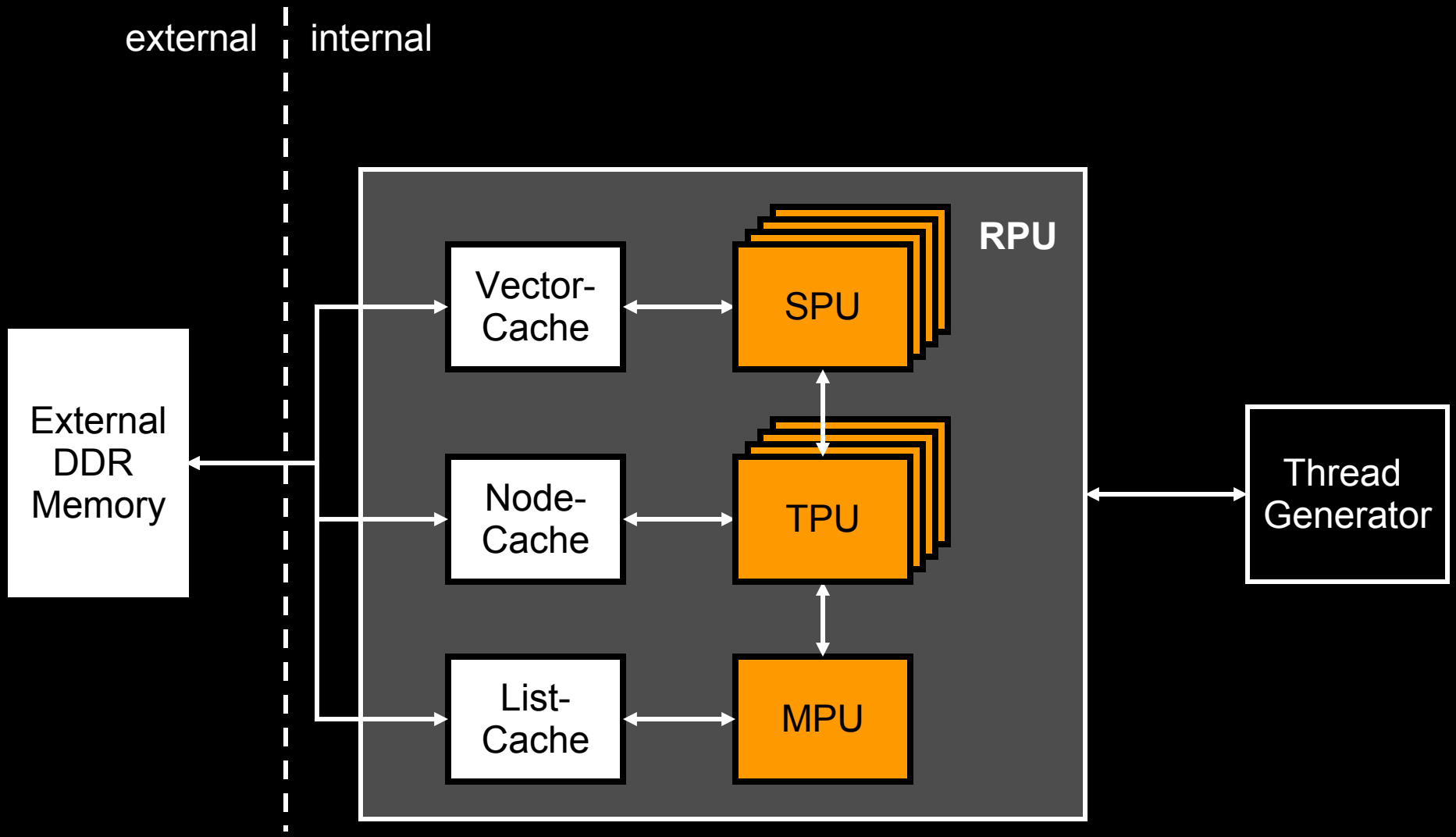
Previous Work

- CPUs [Parker'99, Wald'01]
  - Flexible, but needs massive number of CPUs
- GPUs [Purcell'02, Carr'02]
  - Stream programming model is still too limited
- Custom HW
  [Green'91, Hall'01, Kobayashi'02, Schmittler'04]
  - Mostly focused on parts of the rendering process
  - No programmability

RPU Approach

- **Shading processor**
  - Design similar to fragment processors on GPUs
  - Highly parallel, highly efficient
- **Improved programming model**
  - Add highly efficient recursion, conditional branching
  - Add flexible memory access (beyond textures)
- **Custom traversal hardware**
  - High-performance kd-tree traversal

RPU Architecture



RPU Design

- Shader Processing Units (SPU)
  - Intersection, Shading, Lighting, …
    - Significant vector processing
    - High instruction level parallelism
  - SPU approach: instruction set similar to GPUs
    - 4-vector SIMD
    - Dual issue & pairing
      - Arithmetic splitting (3+1 and 2+2 vector)
      - Arithmetic + load
      - Arithmetic + conditional jump, call, return
    - High code density

Instruction Level Parallelism +

Instruction Set of SPU

- Short vector instruction set
  - mov, add, mul, mad, frac
  - dph2, dp3, dph3, dp4
- Input modifiers
  - Swizzeling, negation, masking
  - Multiply with power of 2
- Special operations (modifiers)
  - rcp, rsq, sat

- Fast 2D texture addressing
  - texload, texload4x
- Random bulk memory access
  - load, load4x, store

- Conditional instructions (paired)
  - if <condition> jmp label
  - if <condition> call <fun>
  - If <condition> return

- Efficient recursion
  - HW-managed register stack
  - Single cycle function call

- Ray traversal function call
  - trace()

Instruction Level Parallelism +

Instruction Set of SPU

- Short vector instruction set
  - mov, add, mul, mad, frac
  - dph2, dp3, dph3, dp4
- Input modifiers
  - Swizzeling, negation, masking
  - Multiply with power of 2
- Special operations (modifiers)
  - rcp, rsq, sat

- Conditional instructions (paired)
  - if <condition> jmp label
  - if <condition> call <fun>
  - If <condition> return
- Efficient recursion
  - HW-managed register stack
  - Single cycle function call

- Fast 2D texture addressing
  - texload, texload4x
- Random bulk memory access
  - load, load4x, store

- Ray traversal function call
  - trace()

Instruction Level Parallelism +

Instruction Set of SPU

- Short vector instruction set
  - mov, add, mul, mad, frac
  - dph2, dp3, dph3, dp4
- Input modifiers
  - Swizzeling, negation, masking
  - Multiply with power of 2
- Special operations (modifiers)
  - rcp, rsq, sat

- Fast 2D texture addressing
  - texload, texload4x
- Random bulk memory access
  - load, load4x, store

- Conditional instructions (paired)
  - if <condition> jmp label
  - if <condition> call <fun>
  - If <condition> return

- Efficient recursion
  - HW-managed register stack
  - Single cycle function call

- Ray traversal function call
  - trace()

Instruction Level Parallelism +

Instruction Set of SPU

- Short vector instruction set
  - mov, add, mul, mad, frac
  - dph2, dp3, dph3, dp4
- Input modifiers
  - Swizzeling, negation, masking
  - Multiply with power of 2
- Special operations (modifiers)
  - rcp, rsq, sat

- Fast 2D texture addressing
  - texload, texload4x
- Random bulk memory access
  - load, load4x, store

- Conditional instructions (paired)
  - if <condition> jmp label
  - if <condition> call <fun>
  - If <condition> return
- Efficient recursion
  - HW-managed register stack
  - Single cycle function call

- Ray traversal function call
  - trace()

Instruction Level Parallelism +

Instruction Set of SPU

- Short vector instruction set
  - mov, add, mul, mad, frac
  - dph2, dp3, dph3, dp4
- Input modifiers
  - Swizzeling, negation, masking
  - Multiply with power of 2
- Special operations (modifiers)
  - rcp, rsq, sat

- Fast 2D texture addressing
  - texload, texload4x
- Random bulk memory access
  - load, load4x, store

- Conditional instructions (paired)
  - if <condition> jmp label
  - if <condition> call <fun>
  - If <condition> return

- Efficient recursion
  - HW-managed register stack
  - Single cycle function call

- Ray traversal function call
  - trace()

Instruction Level Parallelism +

Instruction Set of SPU

- Short vector instruction set
  - mov, add, mul, mad, frac
  - dph2, dp3, dph3, dp4
- Input modifiers
  - Swizzeling, negation, masking
  - Multiply with power of 2
- Special operations (modifiers)
  - rcp, rsq, sat

- Fast 2D texture addressing
  - texload, texload4x
- Random bulk memory access
  - load, load4x, store

- Conditional instructions (paired)
  - if <condition> jmp label
  - if <condition> call <fun>
  - If <condition> return

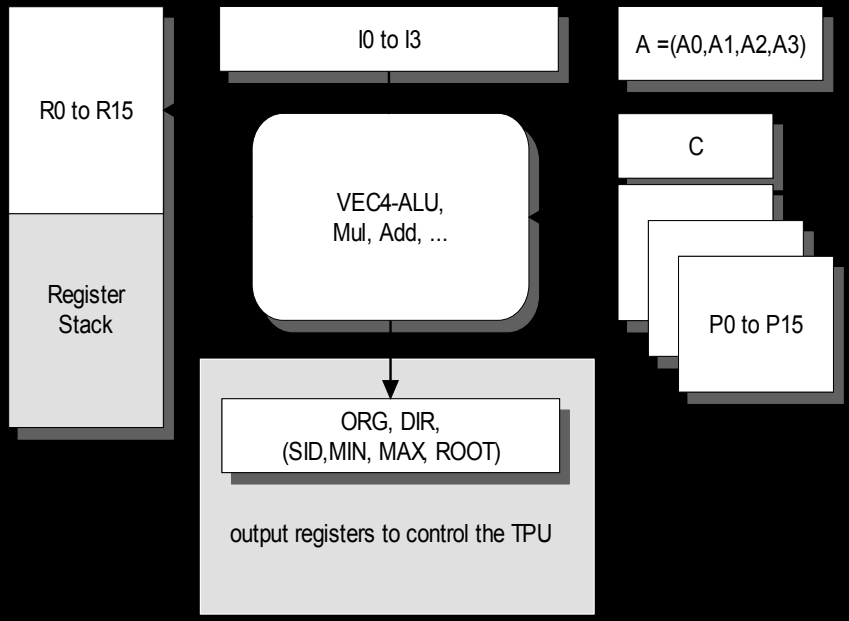
- Efficient recursion
  - HW-managed register stack
  - Single cycle function call

- Ray traversal function call
  - trace()

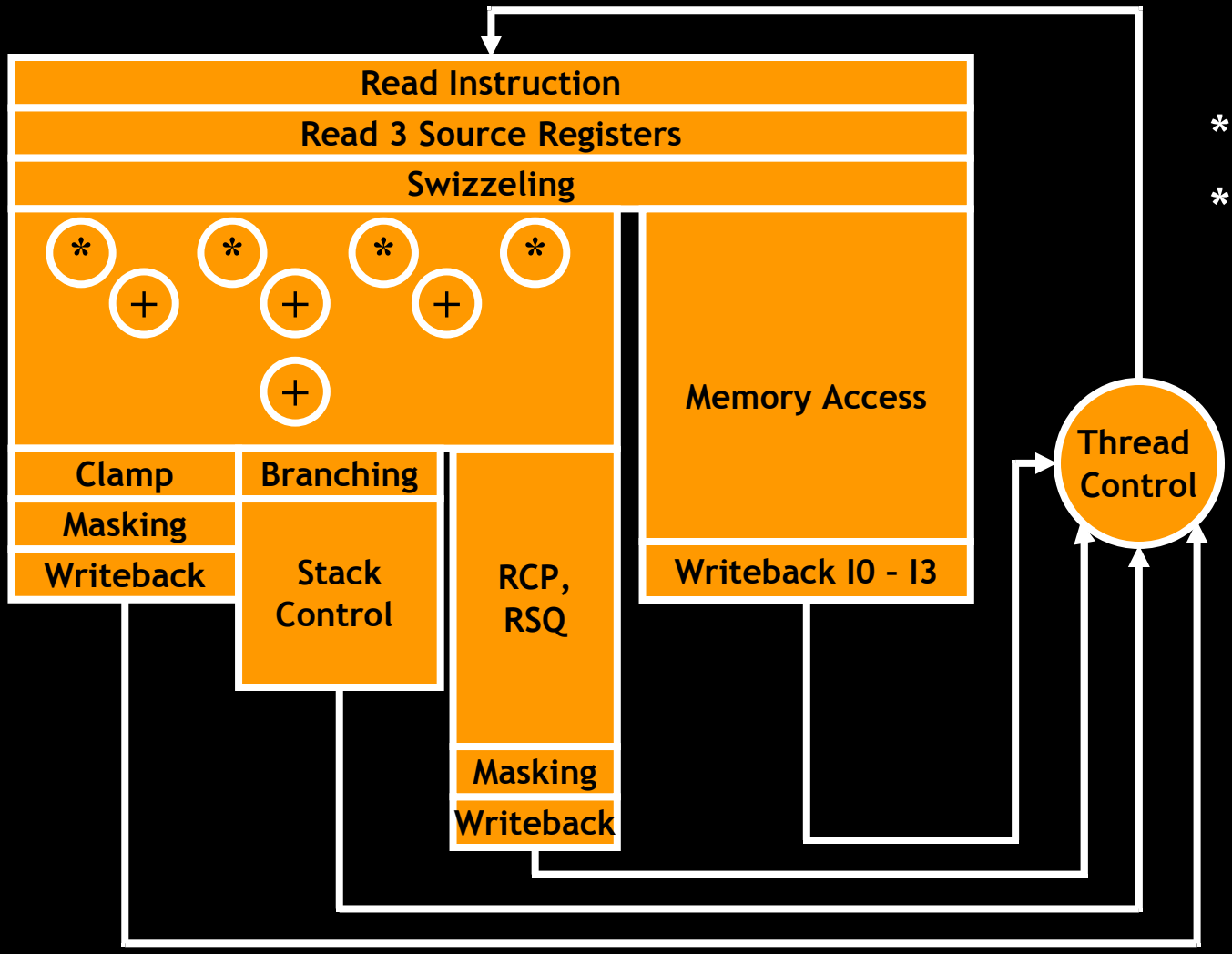
Instruction Level Parallelism +

SPU Vector Registers

- All registers have 4 components (float or integer)
- R0 to R15: General registers
  - Index into a HW managed register stack
- P0 to P15: shader parameters
- I0 to I3: data read from memory
- A = (A0,A1,A2,A3)
  - Memory addressing
- S: Special register
  - Result from rcp, rsq, …
- ORG, DIR, ...
  - TPU communication



Shader Processing Unit Pipelining



mov R0,R1
* mov R2,R3
* mov R0,R2

Ray Triangle Intersection

```
; load triangle transformation
  load4x A.y,0
```

```
; prepare intersection comp.
  dp3_rcp R7.z,I2,R3
  dp3 R7.y,I1,R3
  dp3 R7.x,I0,R3
  dph3 R6.x,I0,R2
  dph3 R6.y,I1,R2
  dph3 R6.z,I2,R2
```

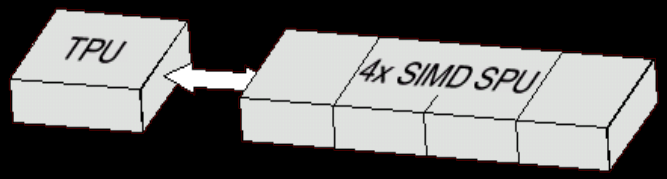
```
; compute hit distance
  mul R8.z,-R6.z,S.z
  + if z <0 return
```

- Input
- Arithmetic (dot products)
- Multi-issue (arith. & cond.)

```
; barycentric coordinates
  mad R8.xy,R8.z,R7,R6
    + if or xy (<0 or >=1)
        return

; hit if u + v < 1
  add R8.w,R8.x,R8.y
    + if w >=1 return

; hit distance closer than last one?
  add R8.w,R8.z,-R4.z
  + if w >=0 return

; save hit information
  mov SID,I3.x
  + mov MAX,R8.z
  mov R4.xyz,R8
  + return
```

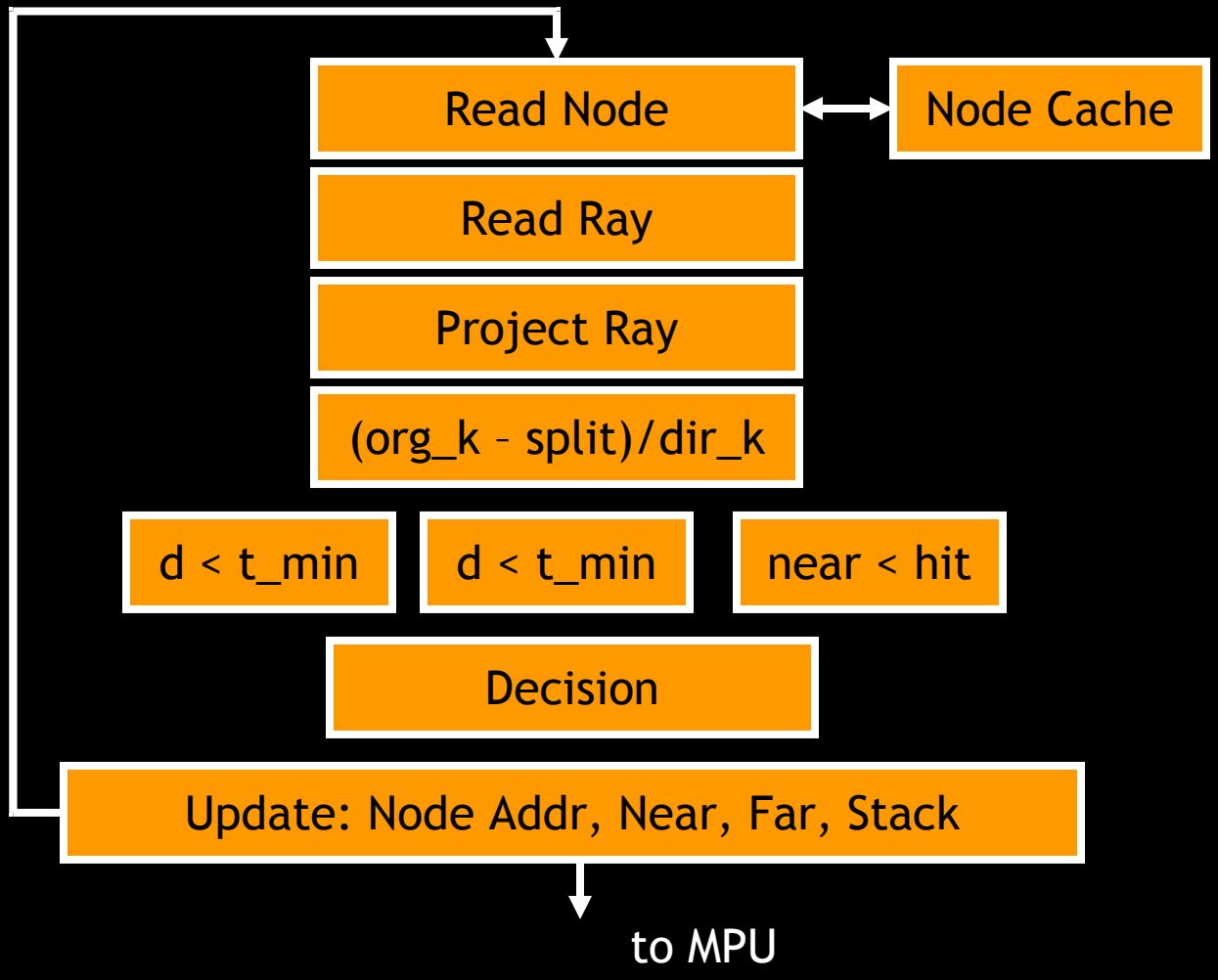
RPU Design

- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
  - Uses kd-trees: Flexible and adaptive
    - Handles general scenes well [Havran2000]
    - Simple traversal algorithm
  - But many instruction dependencies in inner loop
    - About 10 instructions
    - CPU: >100 cycles (un-optimized)
~15 cycles (optimized OpenRT)
  - TPU approach: Optimized pipeline
    - 1 cycle throughput



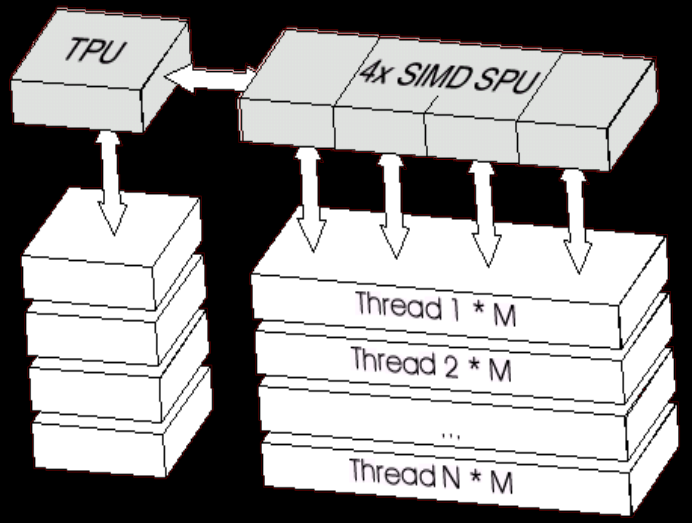
Instruction Level Parallelism + Optimized Pipelining +

TPU (Traversal Processing Unit)



RPU Design

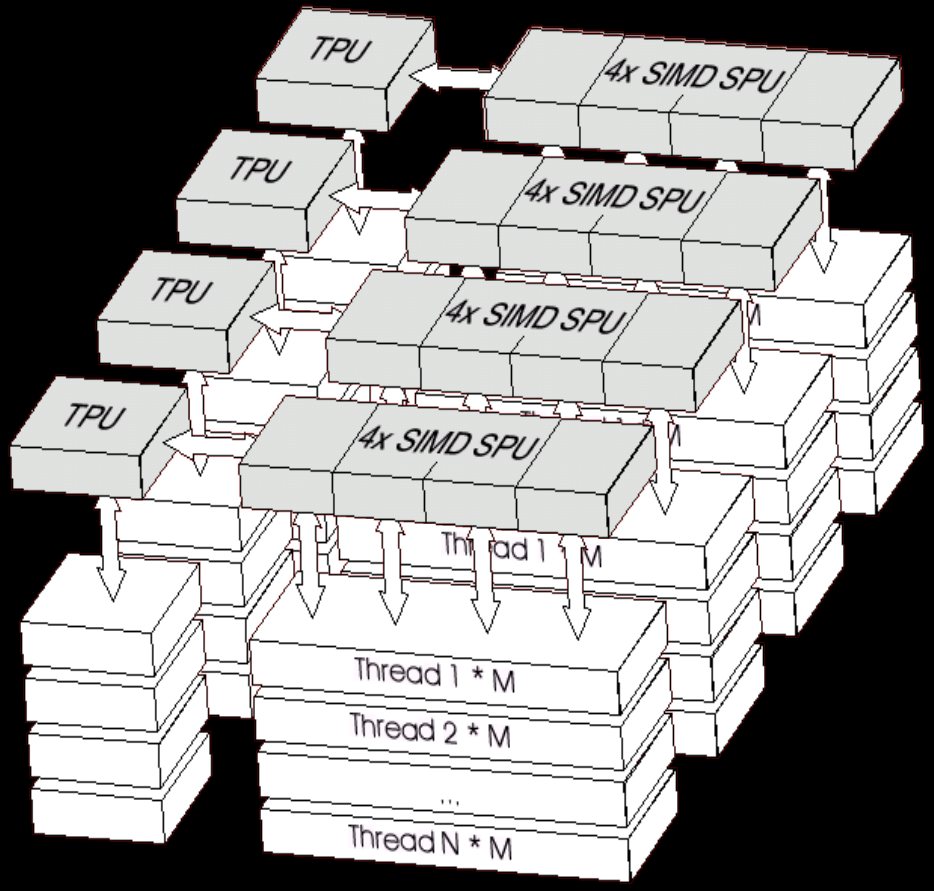
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
  - High computational & memory latency
  - Take advantage of thread level parallelism
  - Increased utilization of hardware units
  - No overhead for switching threads in HW



Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism +

RPU Design

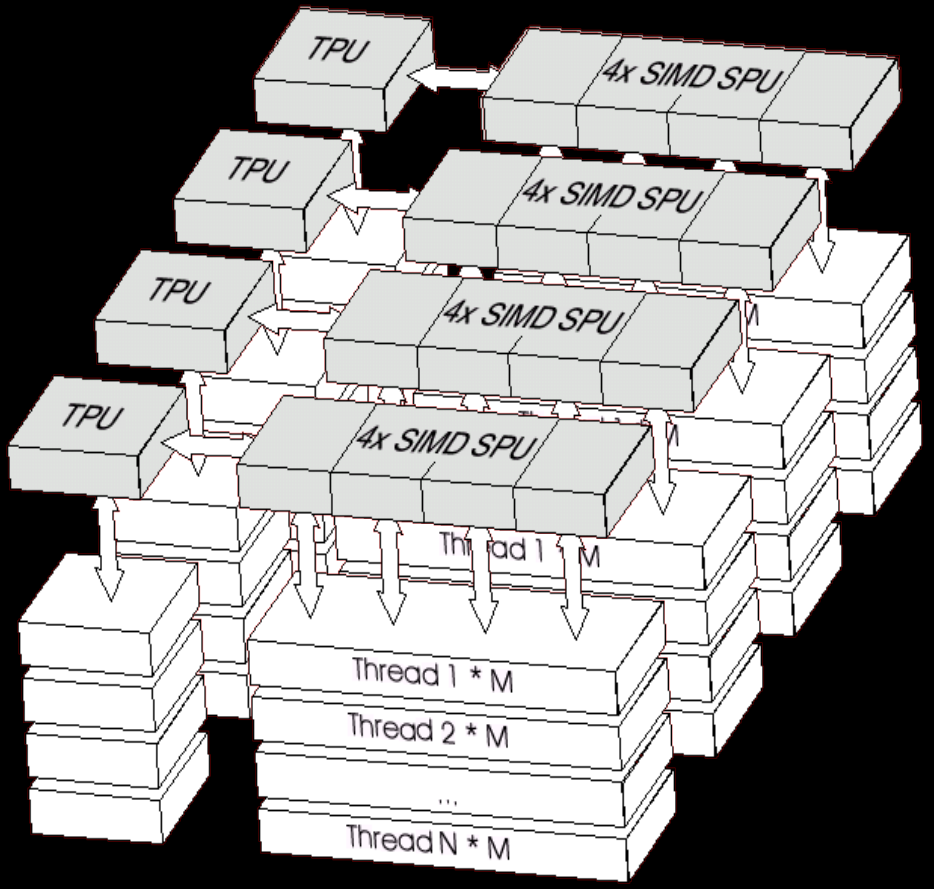
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
  - Takes advantage of ray coherence
    - SIMD execution of threads
  - Reduces hardware complexity
    - Shared processor infrastructure
  - Reduces external bandwidth
    - Combining memory requests
  - Automatic handling of incoherence
    - Splitting and masked execution



Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism + Control Flow Coherence +

RPU Design

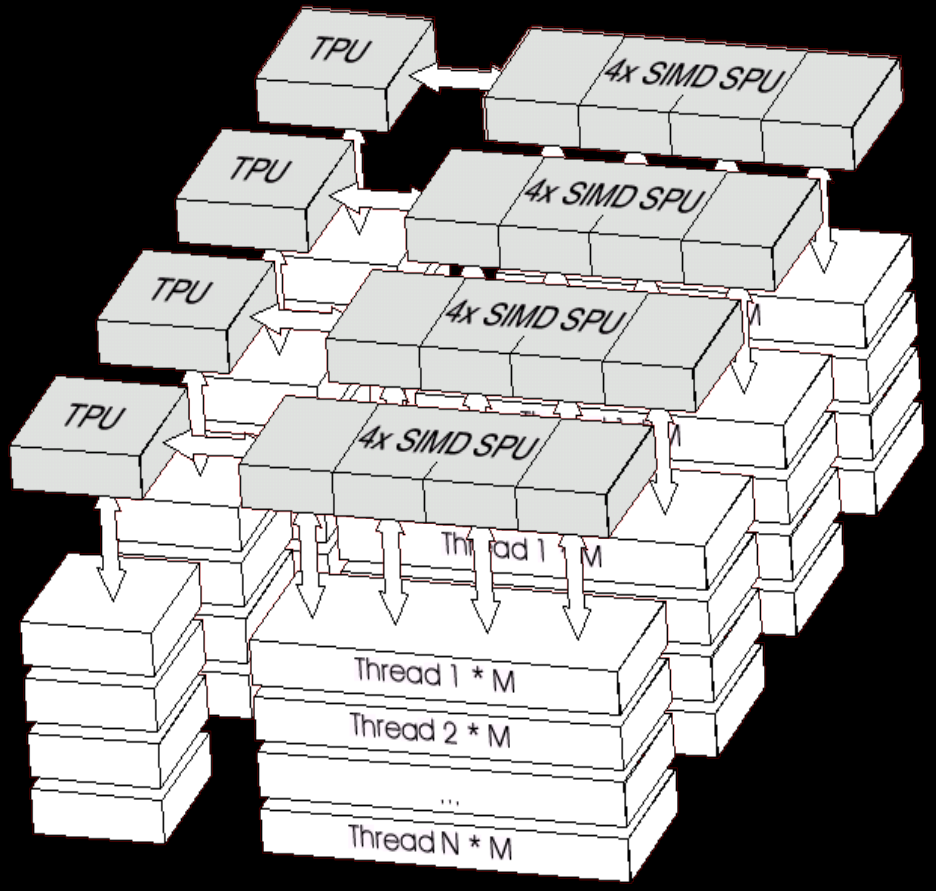
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
- Mailbox Processing Unit (MPU)
  - Objects often span many kd-tree cells
    - Redundant intersections
  - Mailboxing with a small per-chunk cache (e.g. 4 entries)
  - Up to 10x performance for some scenes



Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism + Control Flow Coherence + Avoiding Redundancy

RPU Design

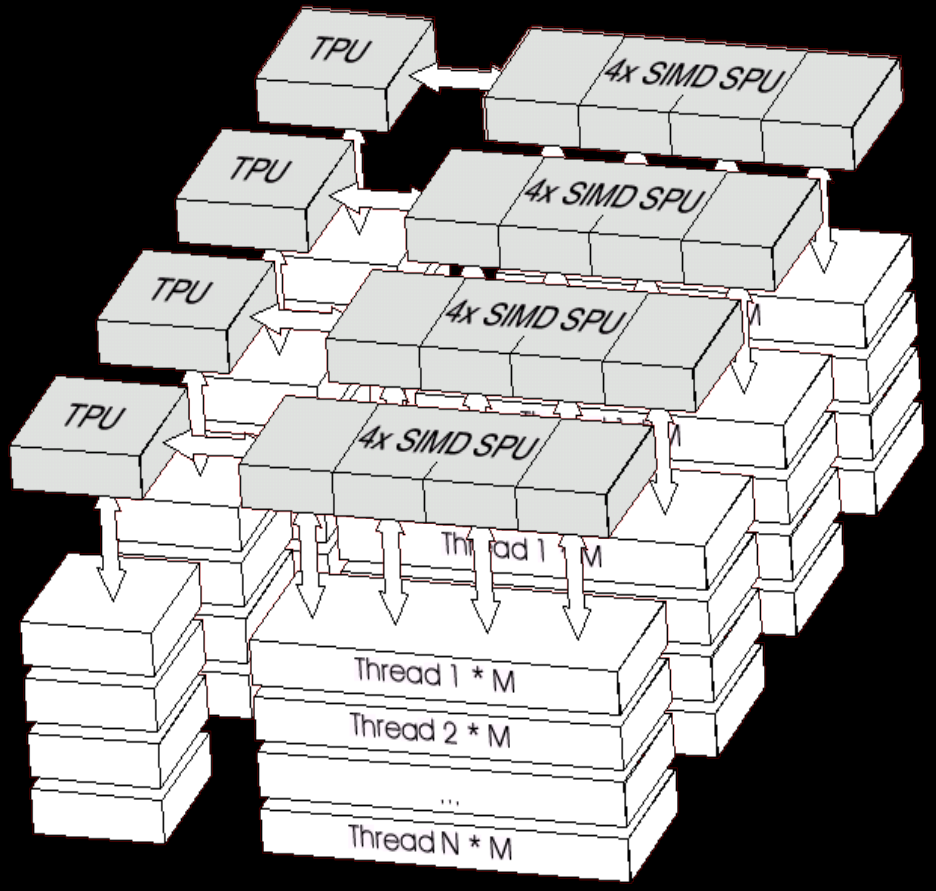
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
- Mailbox Processing Unit (MPU)



Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism + Control Flow Coherence + Avoiding Redundancy

RPU Design

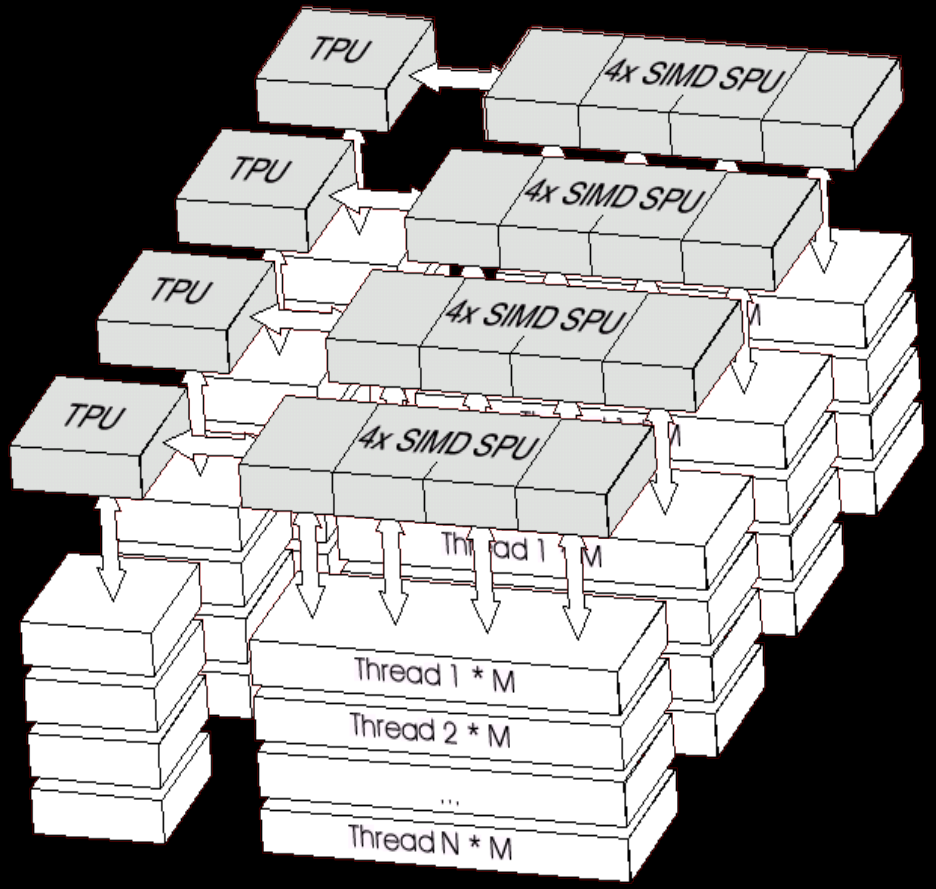
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
- Mailbox Processing Unit (MPU)



Instruction Level Parallelism + **Optimized Pipelining** + Thread Level Parallelism + Control Flow Coherence + Avoiding Redundancy

RPU Design

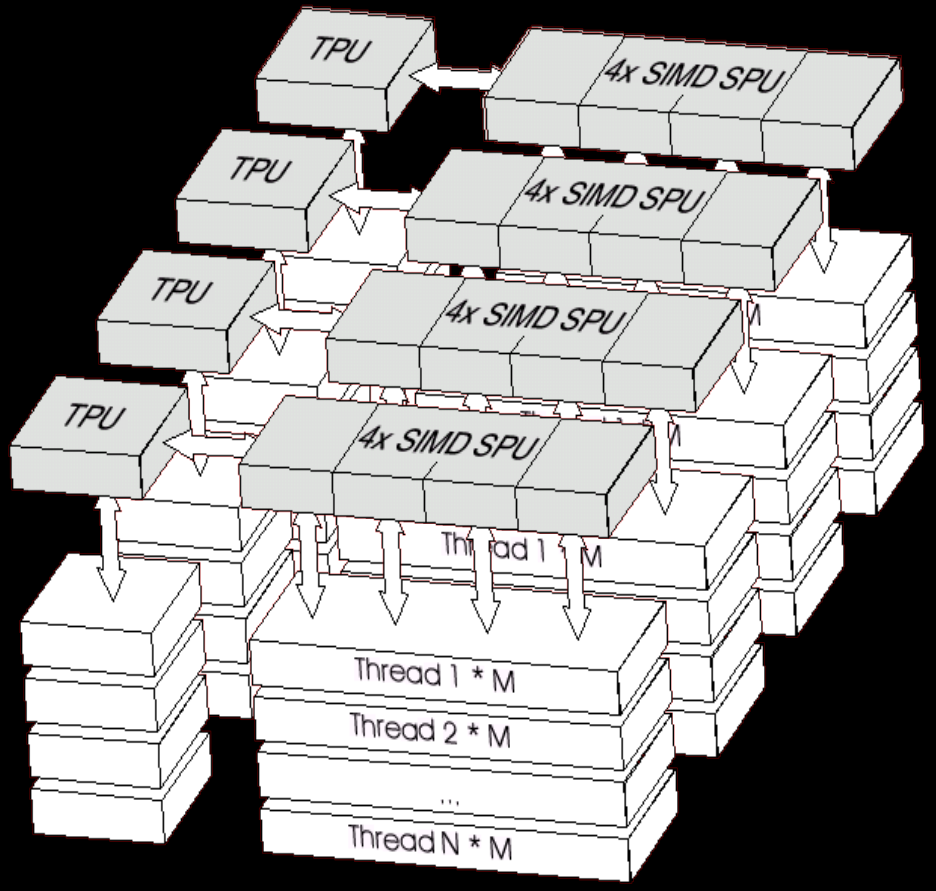
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
- Mailbox Processing Unit (MPU)



Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism + **Control Flow Coherence** + Avoiding Redundancy

RPU Design

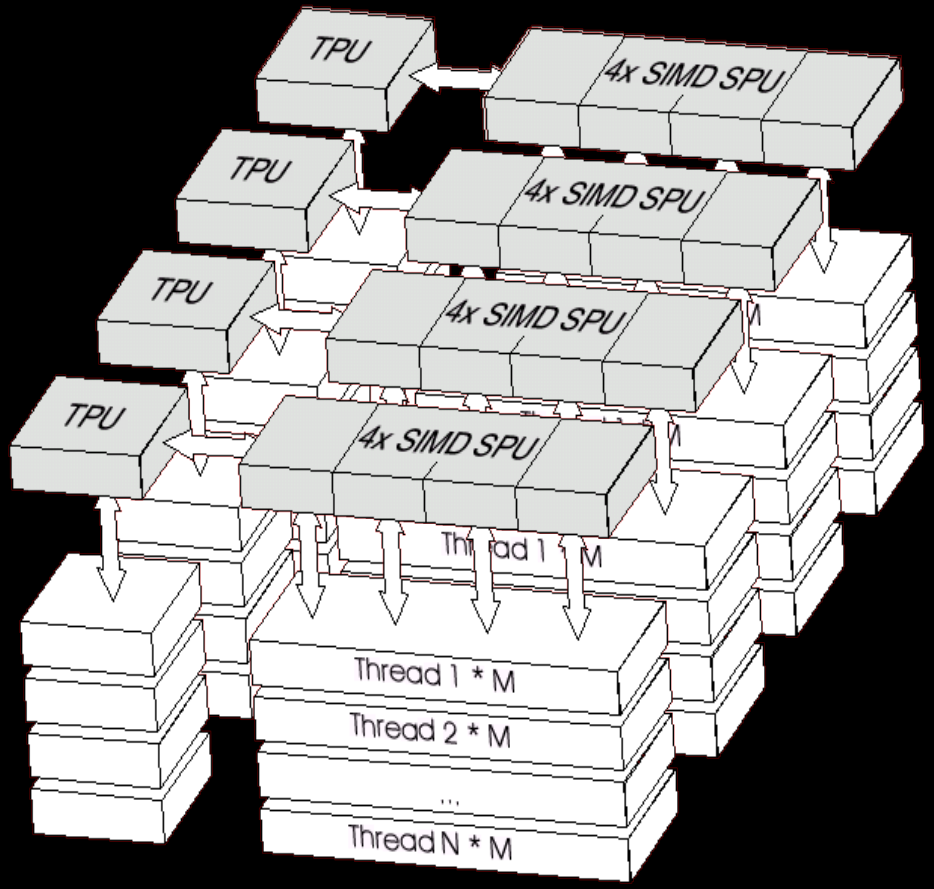
- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
- Mailbox Processing Unit (MPU)



Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism + Control Flow Coherence + **Avoiding Redundancy**

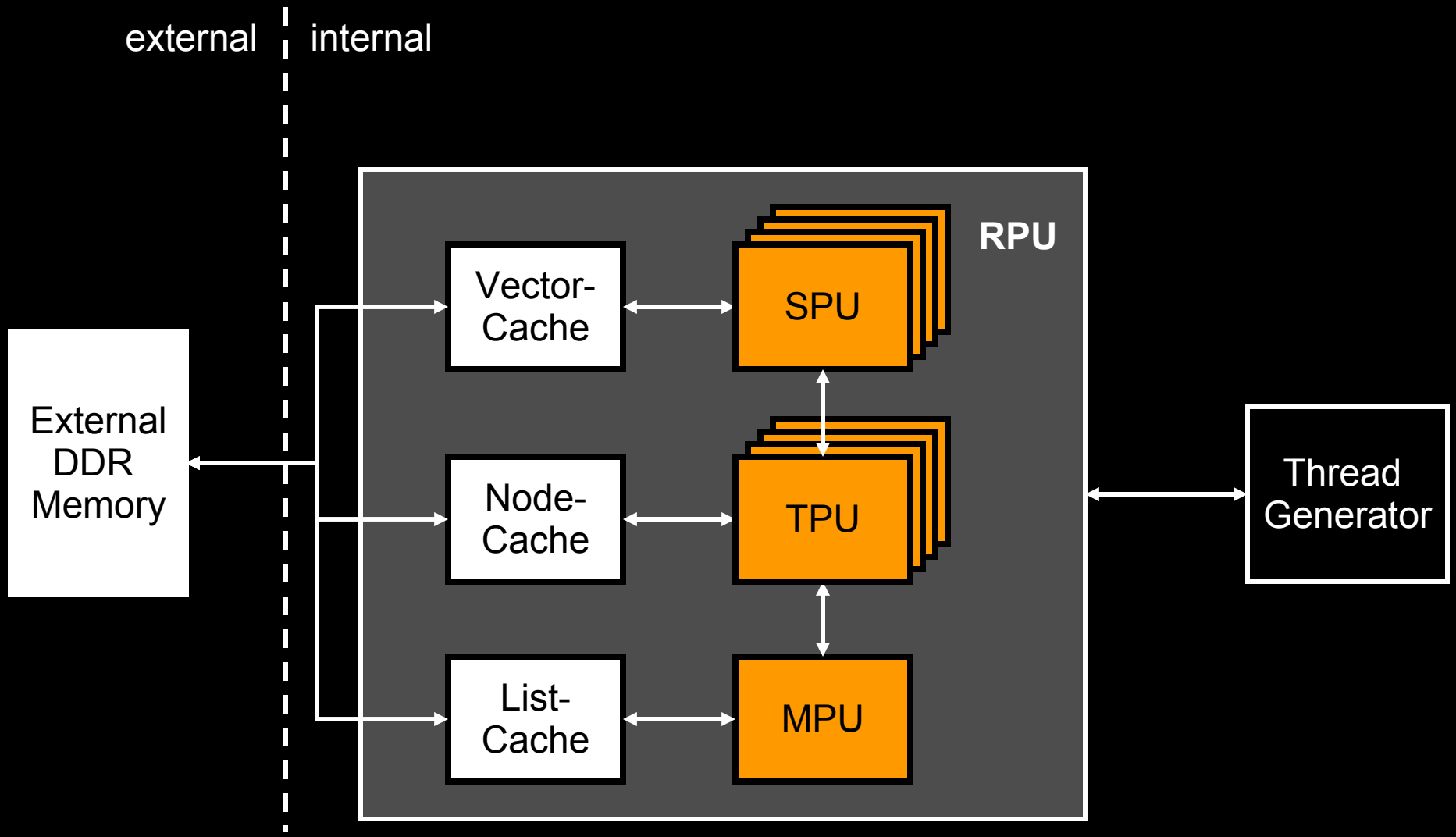
RPU Design

- Shader Processing Units (SPU)
- Custom Ray Traversal Unit (TPU)
- Multi-Threading
- Chunking
- Mailbox Processing Unit (MPU)

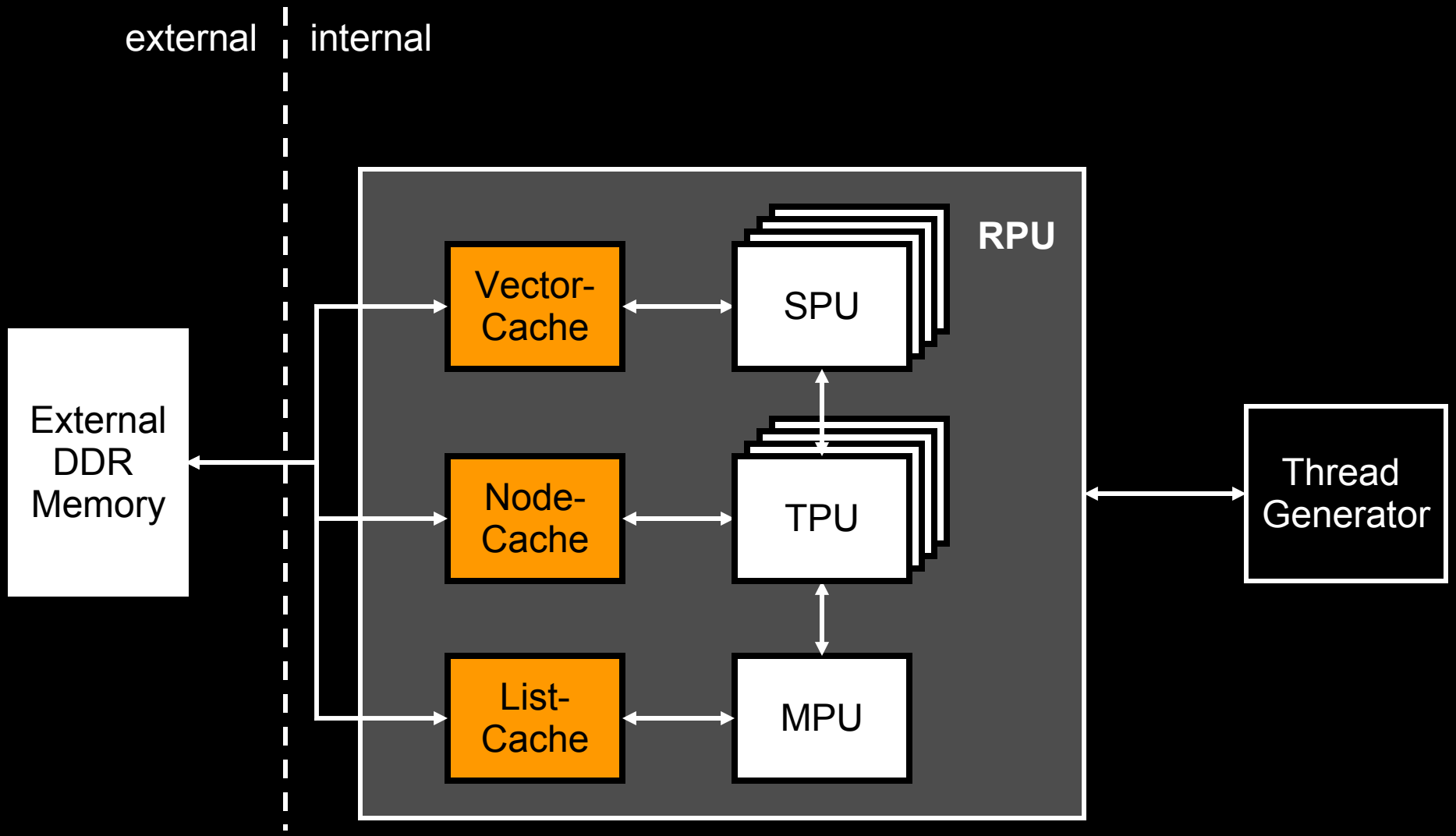


Instruction Level Parallelism + Optimized Pipelining + Thread Level Parallelism + Control Flow Coherence + Avoiding Redundancy

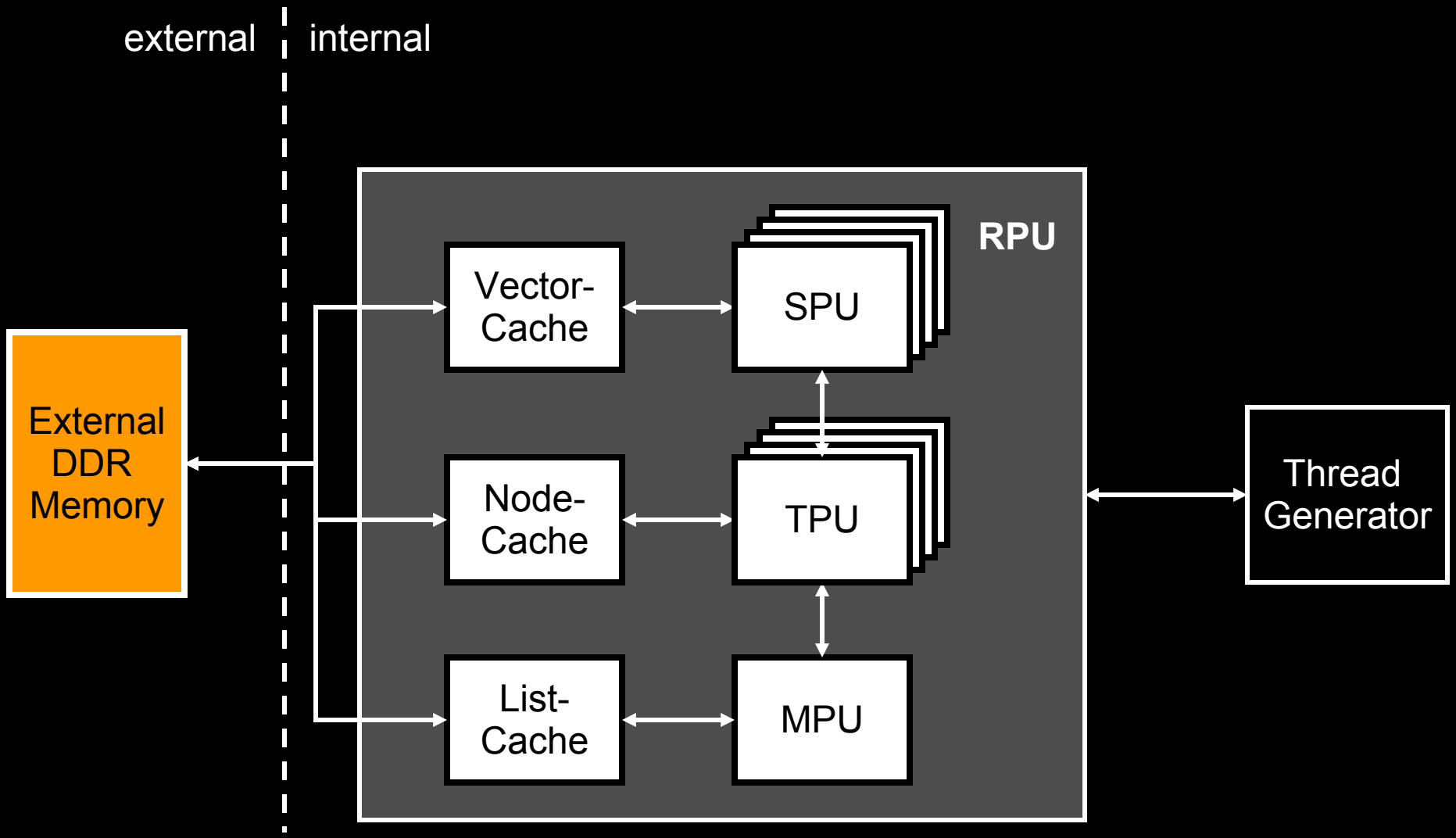
RPU Architecture



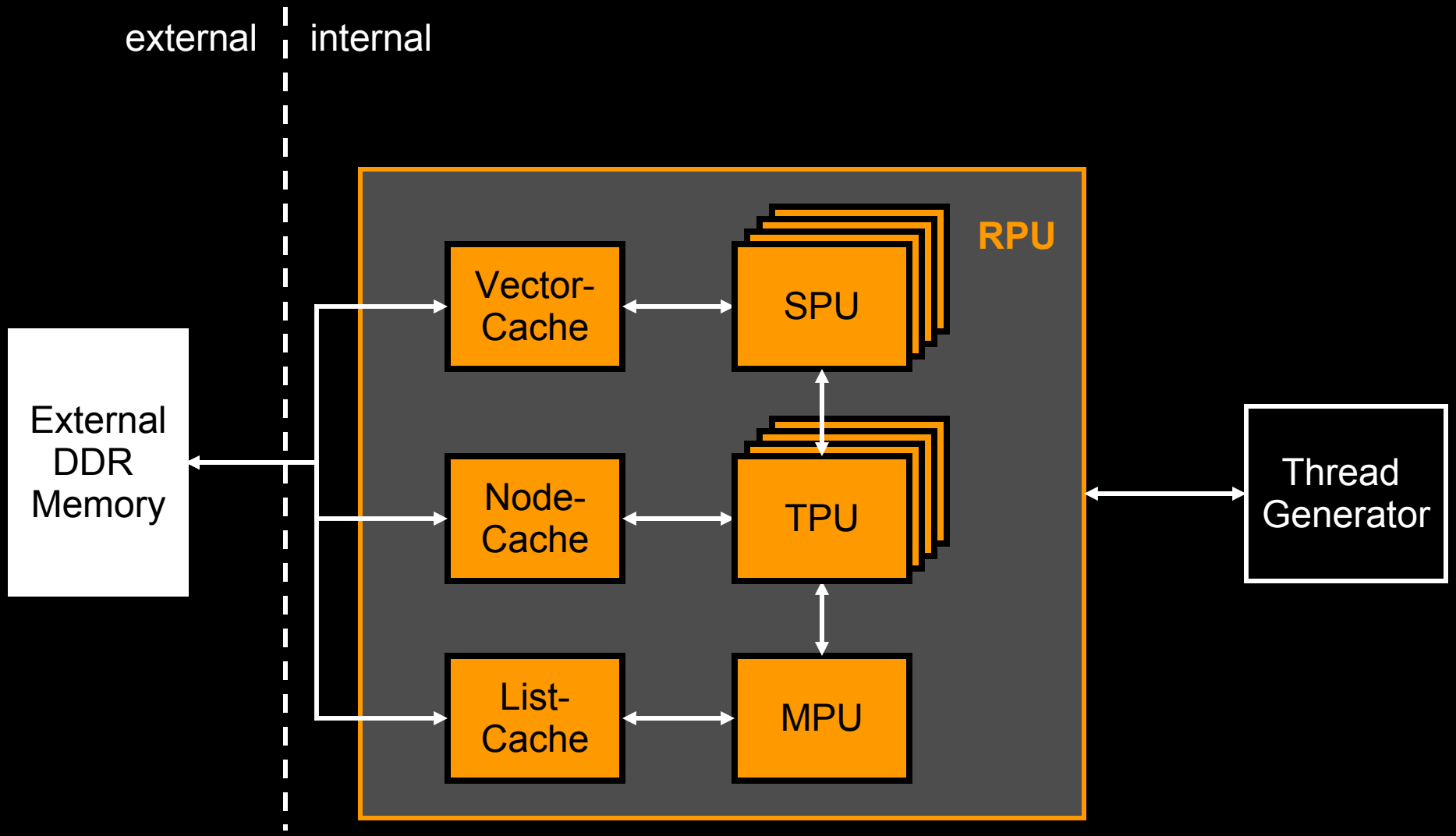
RPU Architecture



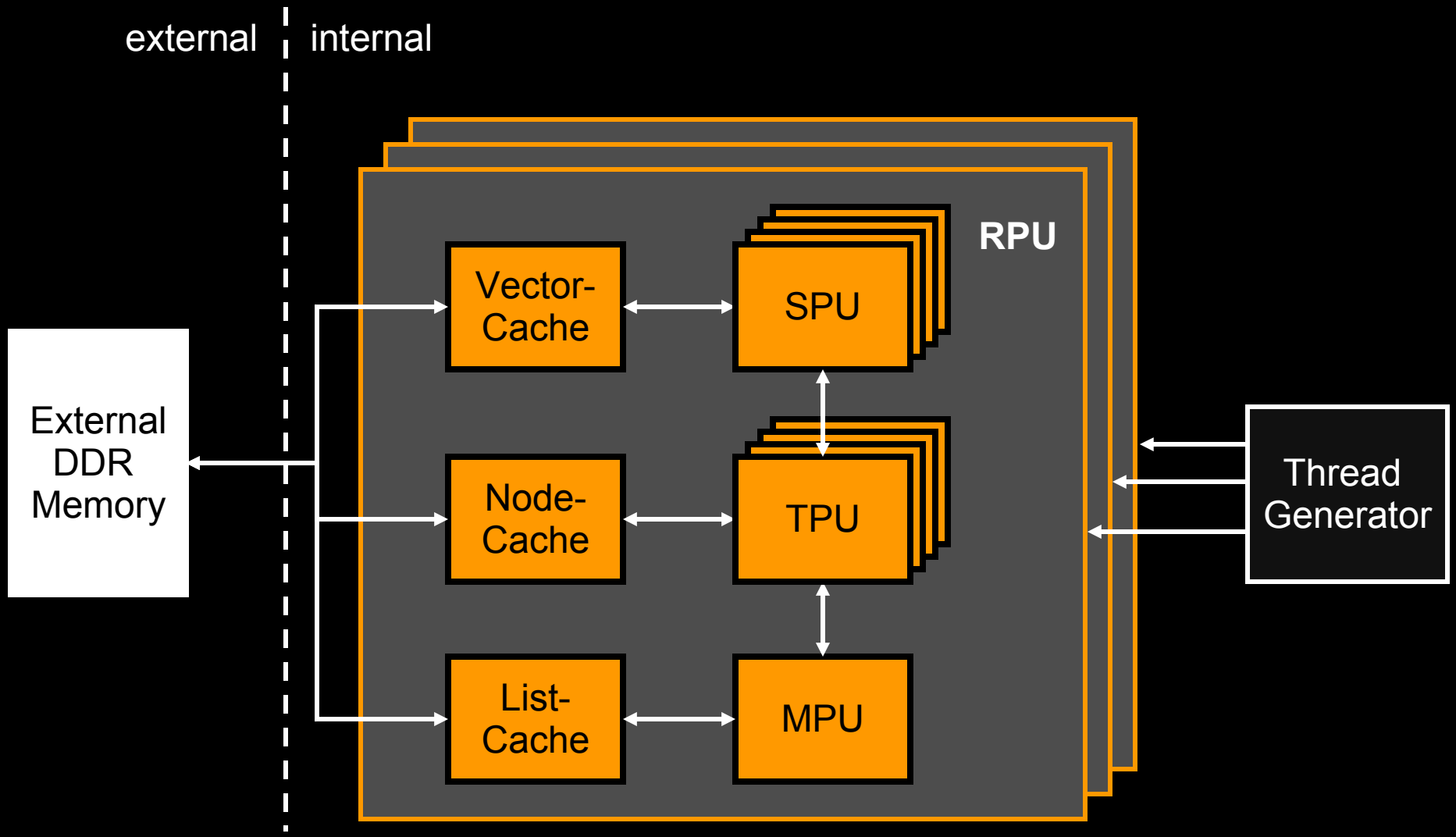
RPU Architecture



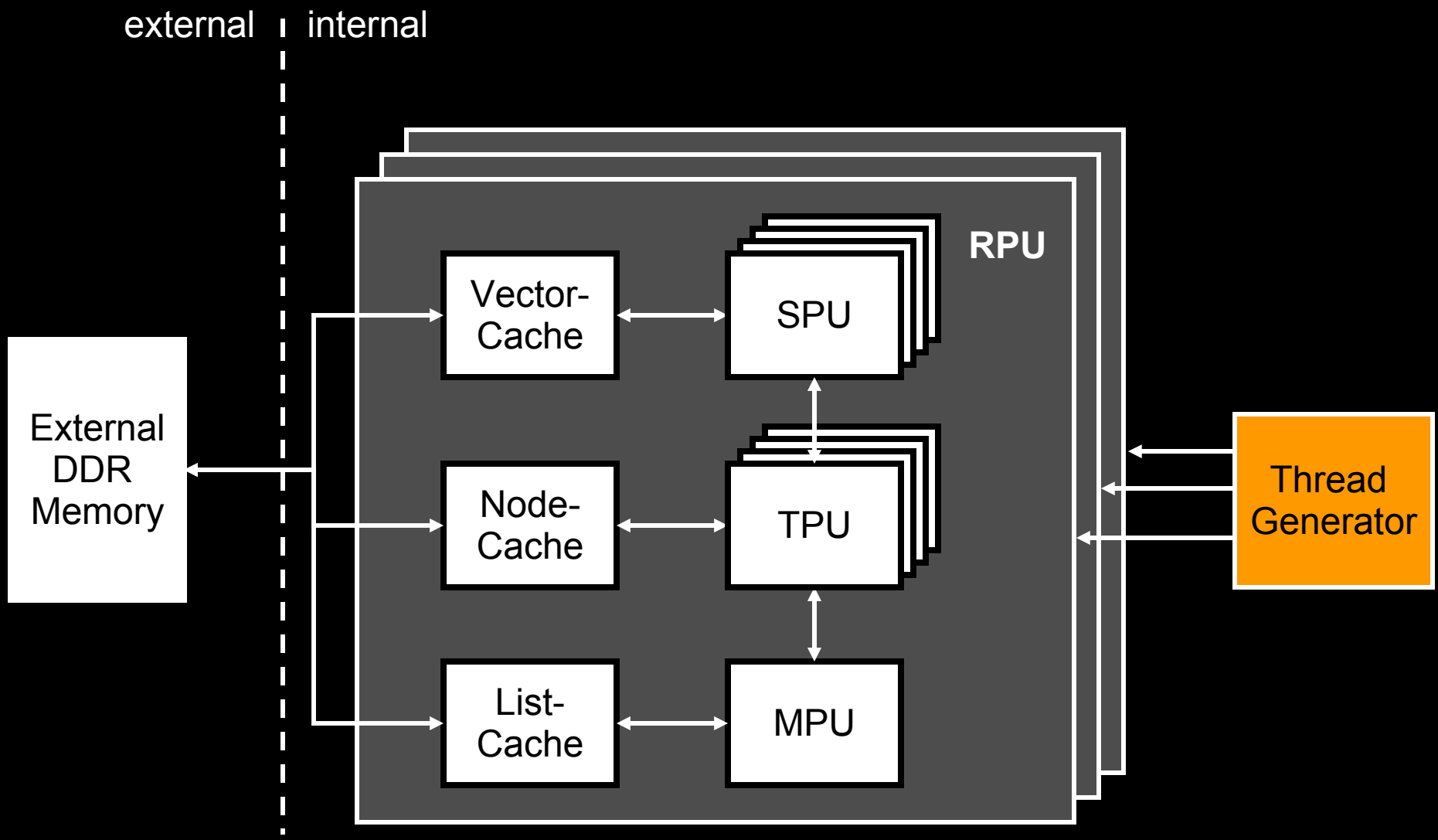
RPU Architecture



RPU Architecture

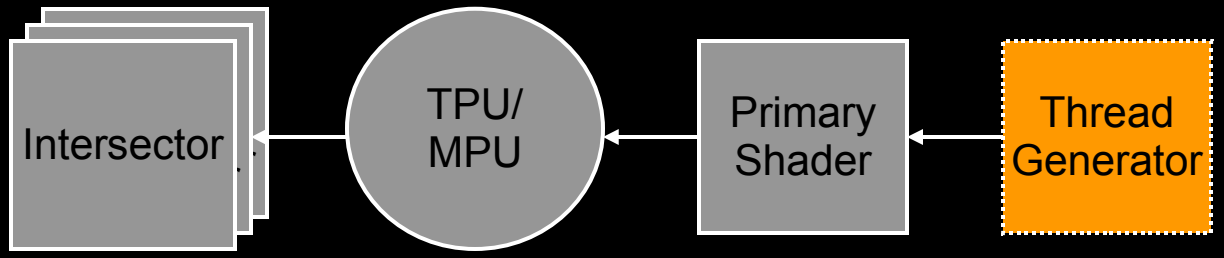


RPU Architecture



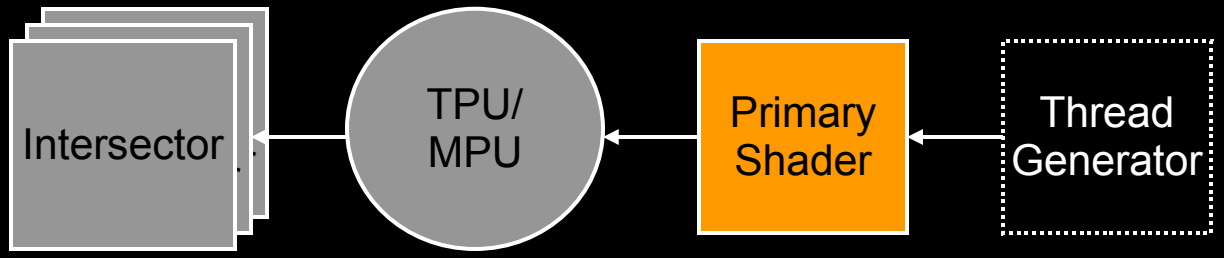
Ray Tracing on an RPU

- Thread generation: initialize SPU registers with pixel coordinates



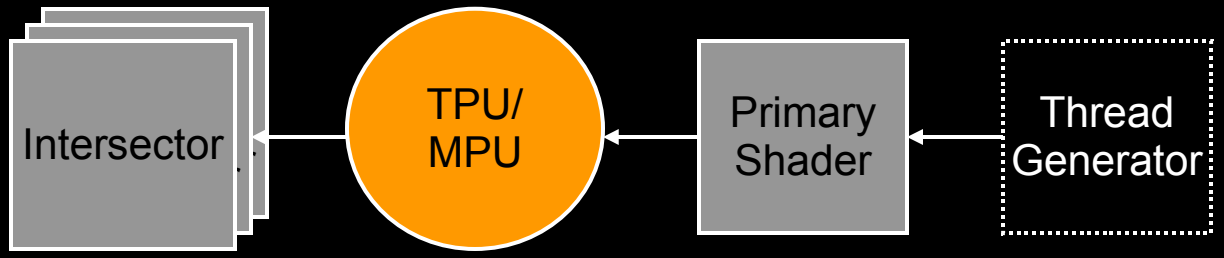
Ray Tracing on an RPU

- Thread generation: initialize SPU registers with pixel coordinates
- Primary shader generates camera ray and calls trace()



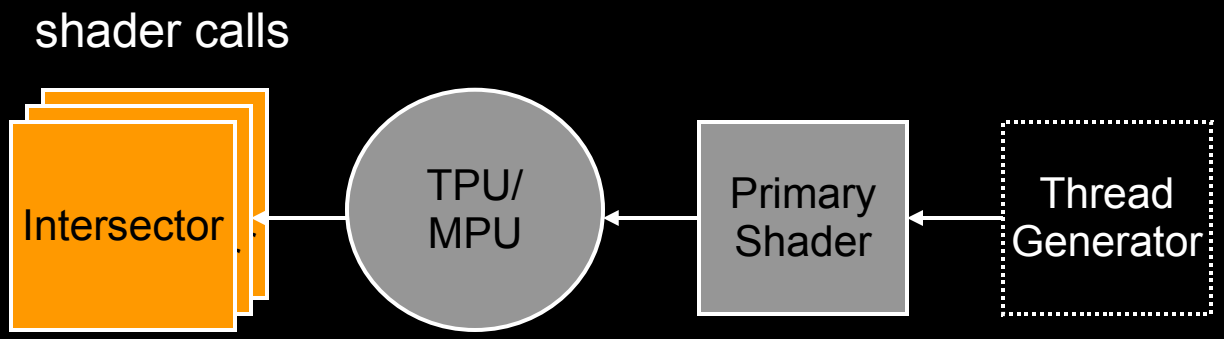
Ray Tracing on an RPU

- Thread generation: initialize SPU registers with pixel coordinates
- Primary shader generates camera ray and calls trace()
- Ray traversal performed on TPU with mailboxing on MPU



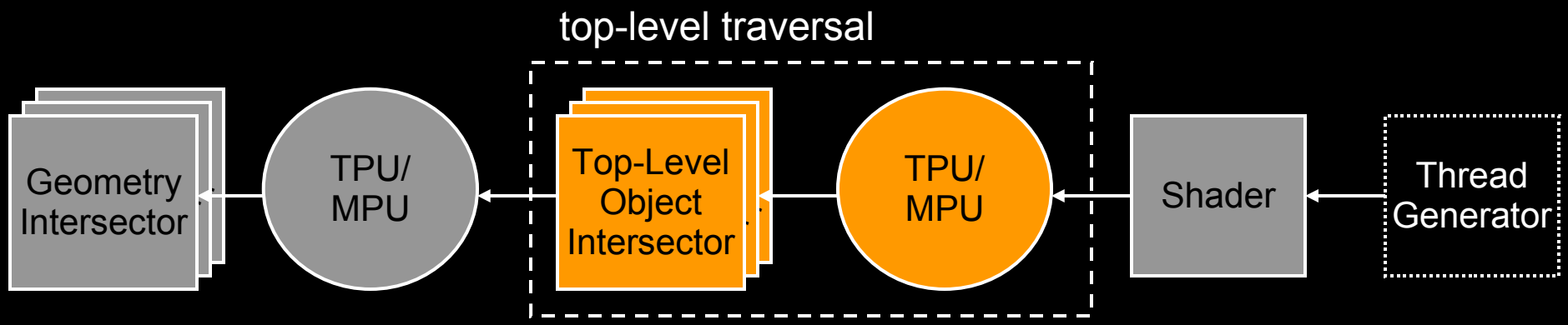
Ray Tracing on an RPU

- Thread generation: initialize SPU registers with pixel coordinates
- Primary shader generates camera ray and calls trace()
- Ray traversal performed on TPU with mailboxing on MPU
- Data dependent call to object/intersection shader on SPU
  - Programmable geometry (triangles, spheres, splines, object instances, …)



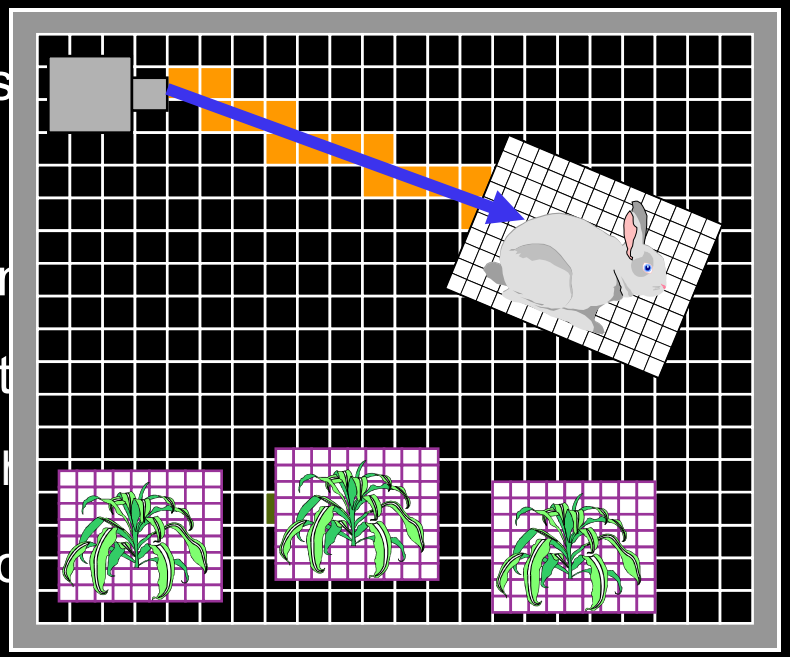
Ray Tracing on an RPU

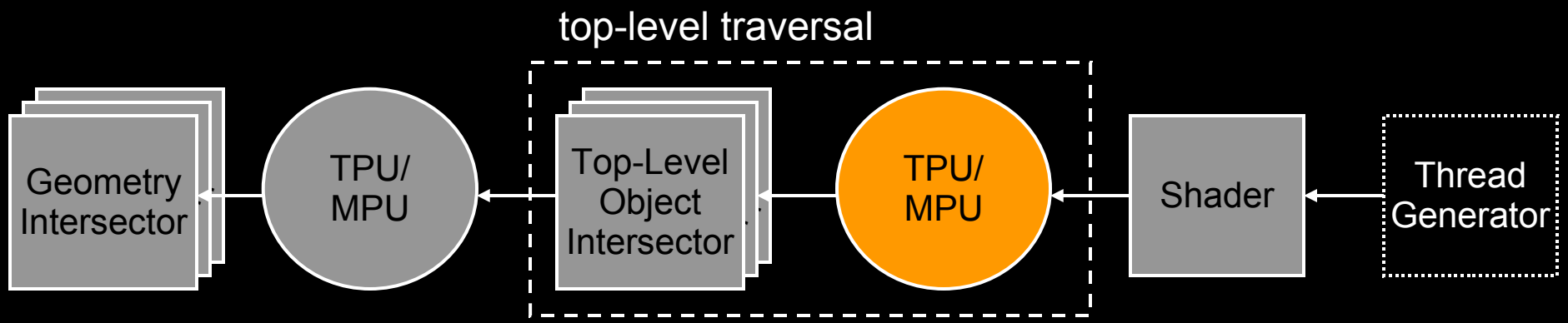
- Thread generation: initialize SPU registers with pixel coordinates
- Primary shader generates camera ray and calls trace()
- Ray traversal performed on TPU with mailboxing on MPU
- Data dependent call to object/intersection shader on SPU
  - Programmable geometry (triangles, spheres, splines, object instances, …)
- Nested ray traversal for object-based dynamic scenes



Ray Tracing on an RPU

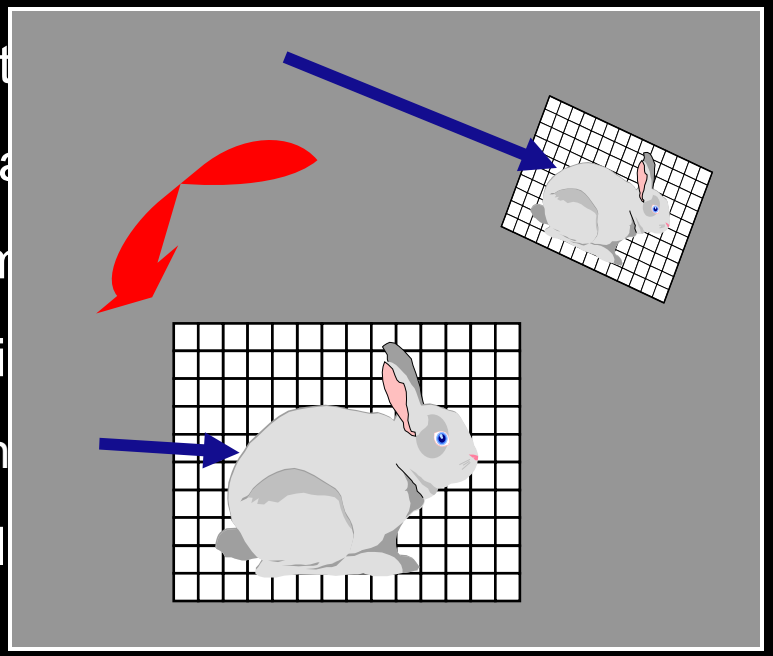
- Thread generation: initialize SPU regis
- Primary shader generates camera ray
- Ray traversal performed on TPU with r
- Data dependent call to object/intersect
  - Programmable geometry (triangles, sph
- Nested ray traversal for object-based c

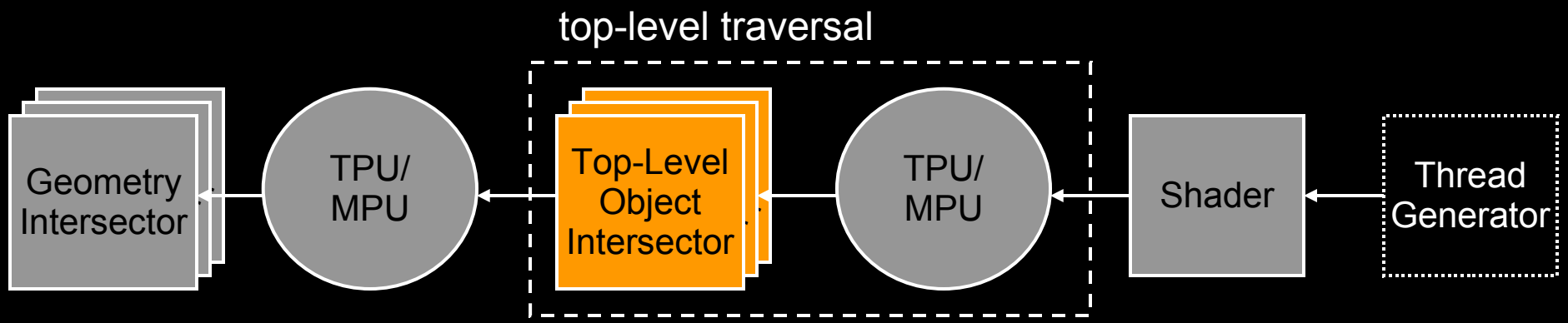




Ray Tracing on an RPU

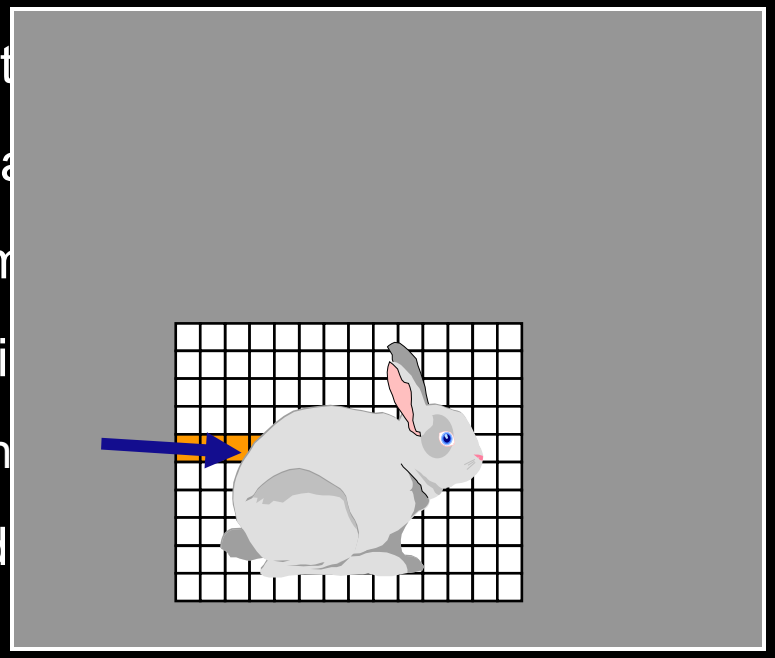
- Thread generation: initialize SPU regist
- Primary shader generates camera ray a
- Ray traversal performed on TPU with m
- Data dependent call to object/intersecti
  - Programmable geometry (triangles, sph
- Nested ray traversal for object-based d

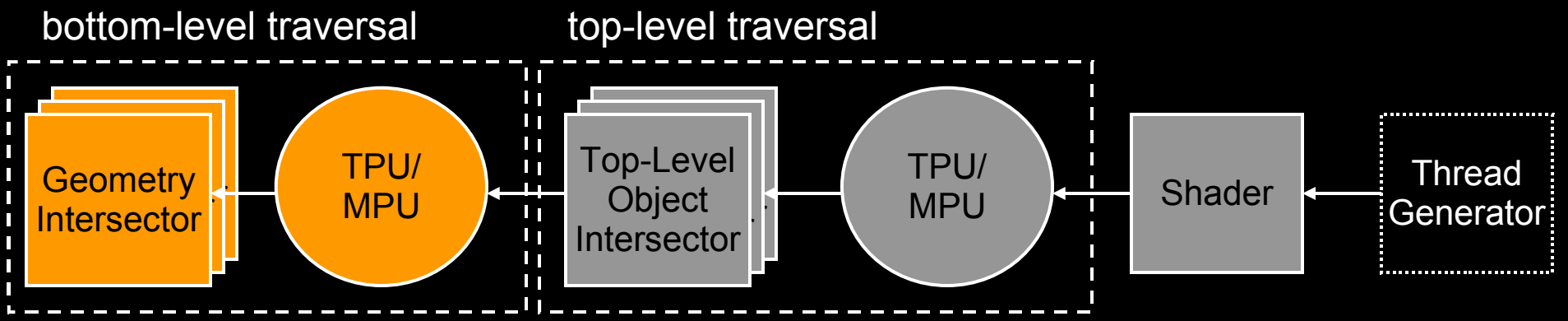




Ray Tracing on an RPU

- Thread generation: initialize SPU regist
- Primary shader generates camera ray a
- Ray traversal performed on TPU with m
- Data dependent call to object/intersecti
  - Programmable geometry (triangles, sph
- Nested ray traversal for object-based d





Hardware Description Library

- Problem
  - Languages like VHDL and Verilog are very coding intensive
    - Many lines of code for little functionality
  - Higher level behavioral languages are difficult to map to hardware
    - Handel-C, Mitrion-C, …
    - Inefficient circuits
- Solution: High-level structural language HWML
  - Implemented as an ML library
  - High level control to generated circuit

Hardware Meta Library (HWML)

- Components are ML functions
  - Recursive component descriptions (e.g. adders)
  - High level components (Components operating on component)
  - Polymorphic functions
  - Fully parameterized design
- Automatic pipelining of components
  - Memories in pipeline supported
  - Automatic multi-threading (high utilization)
- Only ~1000 LOC for entire SPU
  - About six person month for entire RPU design

Implementation Options

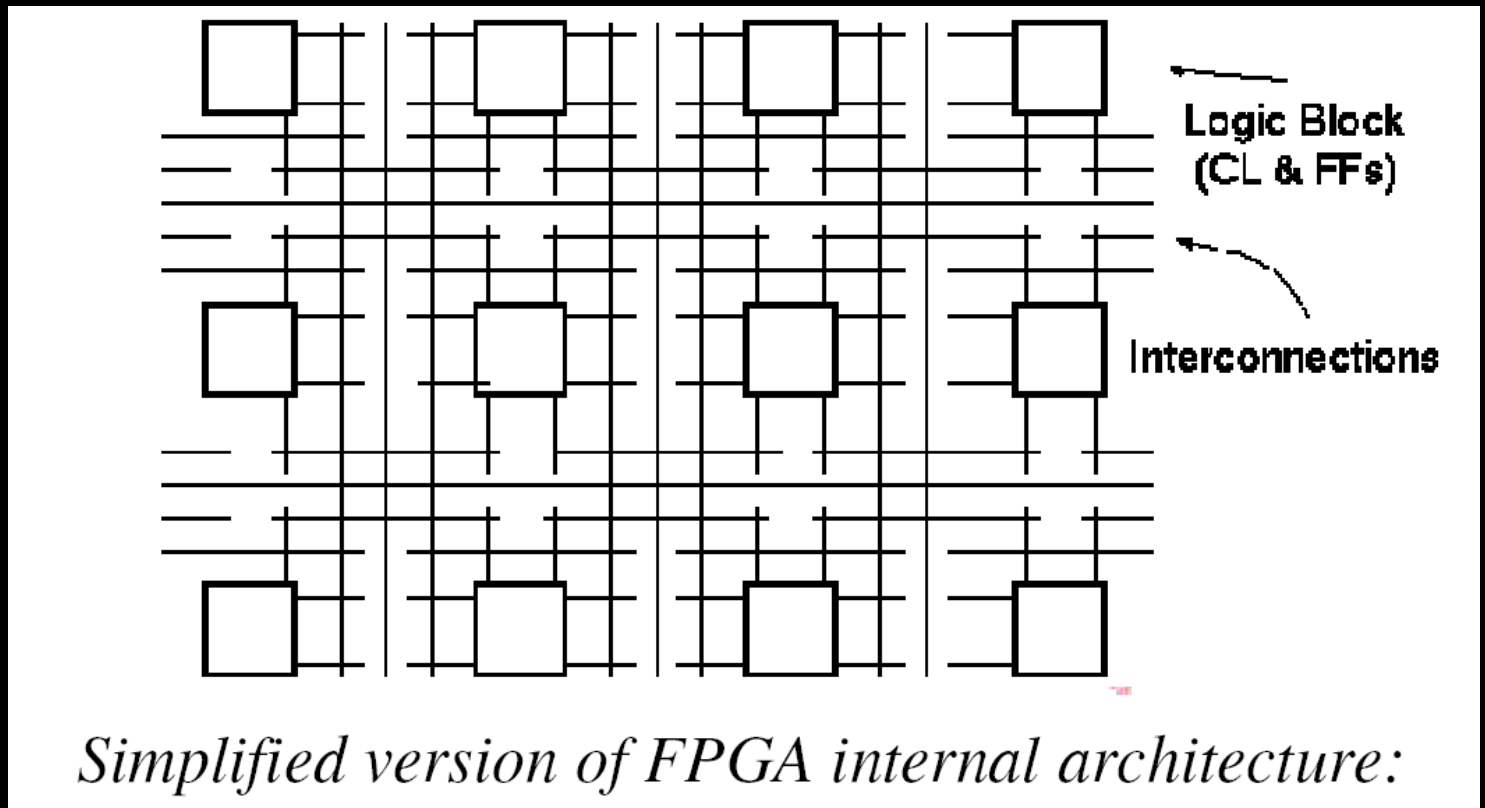
- FPGA: Field Programmable Gate Array
  - Mass-produced chip with user programmable logic and wiring
- ASIC: Application Specific Integrated Circuit
  - Special-purpose chip fabricated from synthesized hardware using standard cell libraries
- Full-Custom IC
  - Special-purpose chip where circuits have been carefully designed by hand

FPGA

- General programmable hardware
  - Programmable logic
  - Programmable wiring
  - Macroblocks like RAM, multipliers, etc.
- Amortize manufacturing costs
- Flexibility at the expense of performance
  - Great for prototyping
  - Or for small/moderate systems with modest performance requirements

FPGA

- Xilinx FPGAs sizes measured in CLBs
  - Configurable Logic Blocks



Simplified version of FPGA internal architecture:

FPGA

- Xilinx FPGAs sizes measured in CLBs

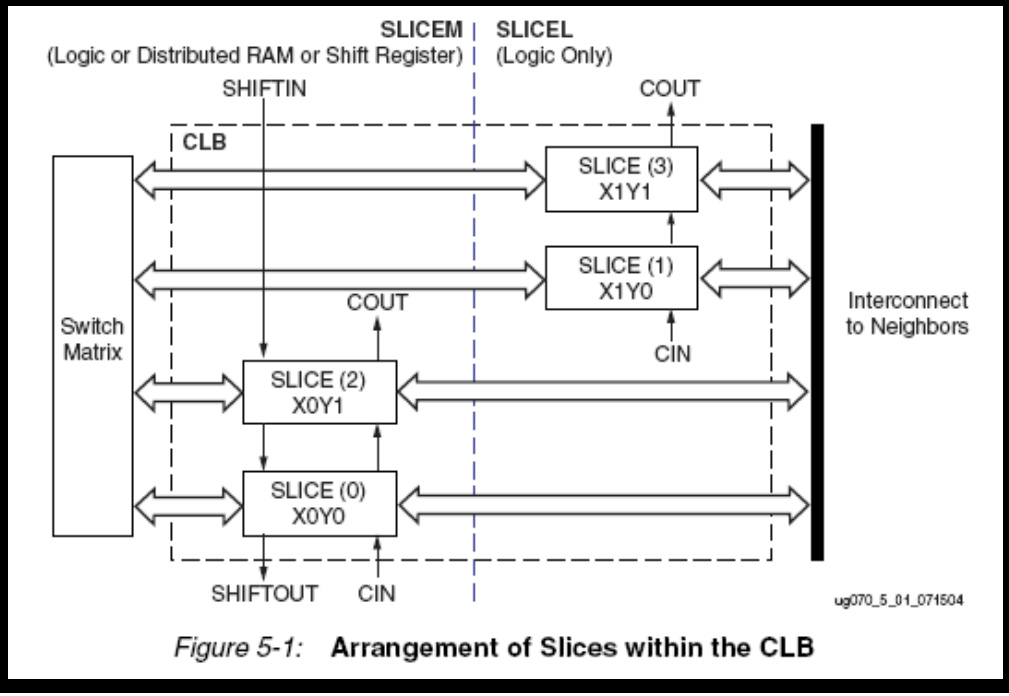


Figure 5-1: **Arrangement of Slices within the CLB**

Table 5-1: **Logic Resources in One CLB**

Slices	LUTs	Flip-Flops	MULT_ANDs	Arithmetic & Carry-Chains	Distributed RAM(1)	Shift Registers(1)
4	8	8	8	2	64 bits	64 bits

FPGA

- Xilinx FPGAs sizes measured in CLBs
  - Four of these “slices” in each CLB

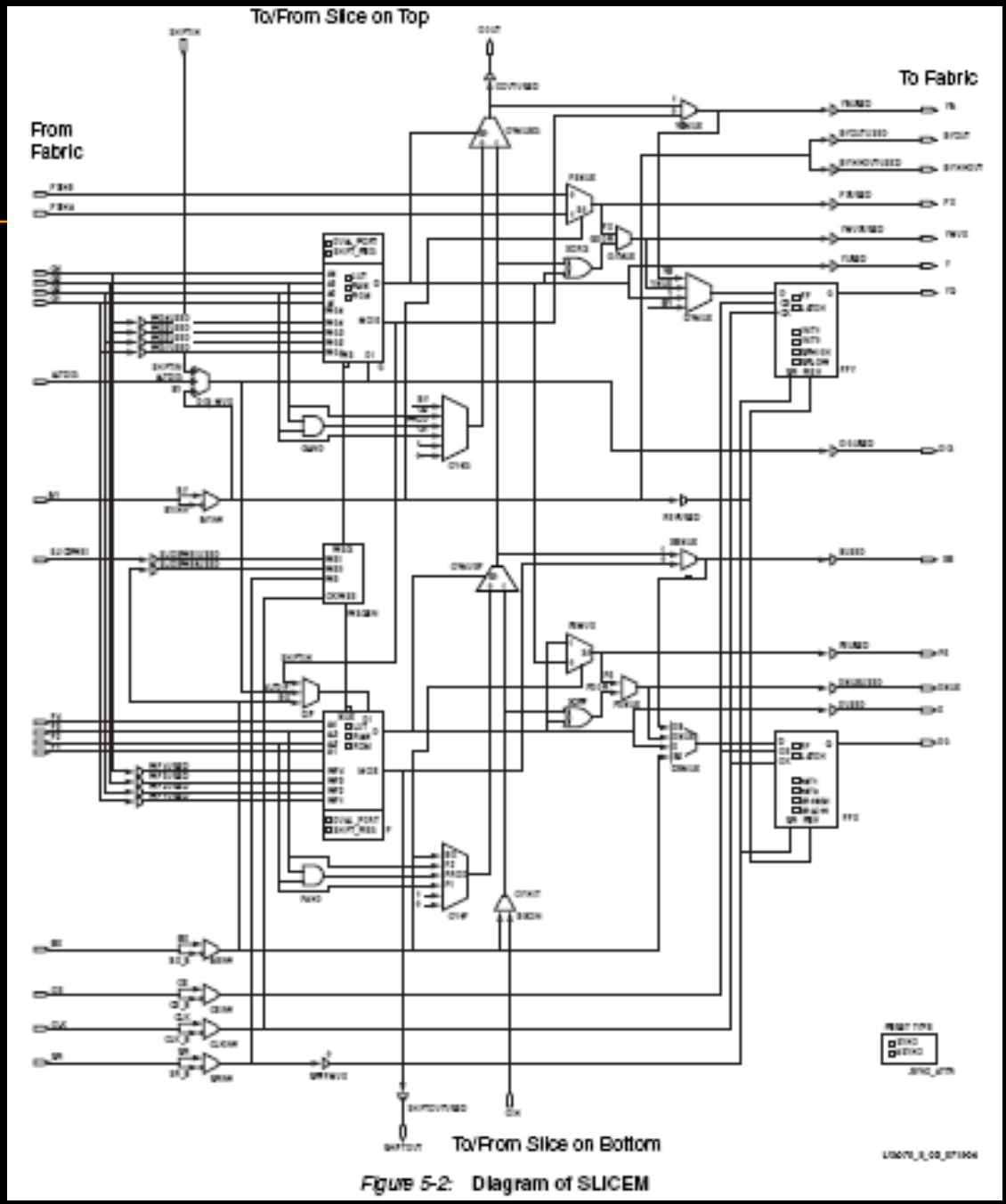
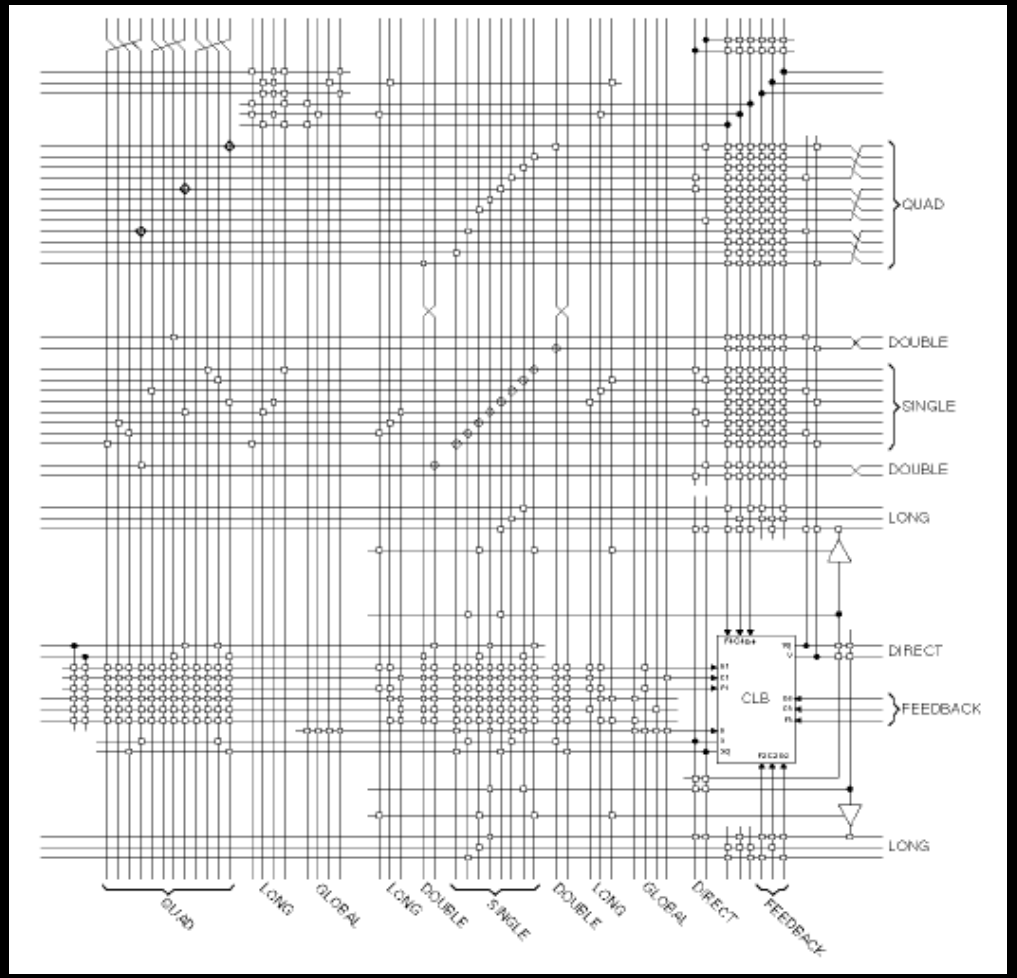


Figure 5-2: Diagram of SLICEM

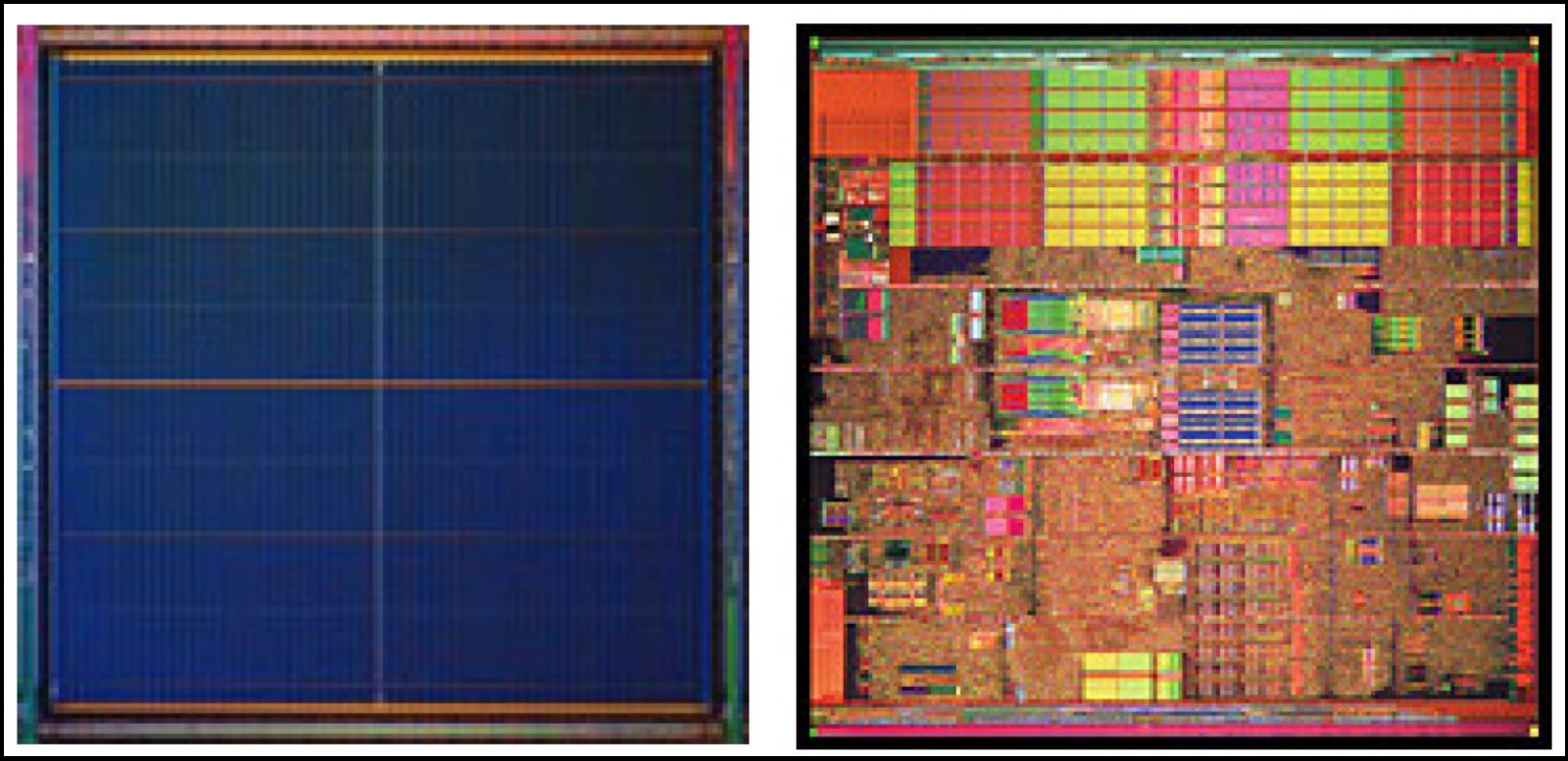
FPGA

- Routing resources typically dominate the chip…



FPGA

- Looks very regular compared to custom
  - Xilinx Virtex FPGA vs Intel Pentium 4



FPGA

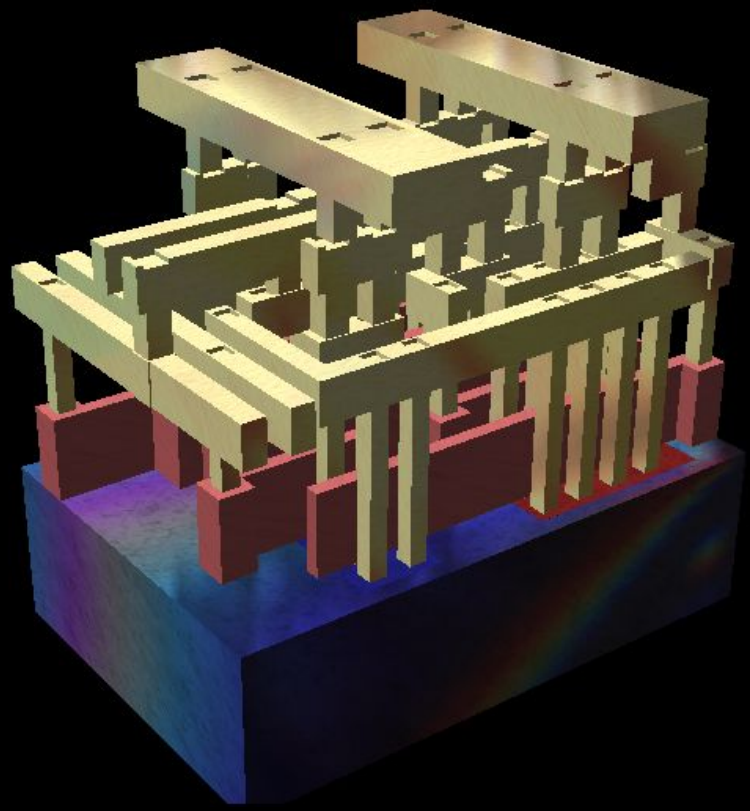
- Programmability means you can make the chip look like anything you want
- It also means it's slower than if it were fixed-function
  - Typical rule of thumb:
    An FPGA implementation will be roughly 5-7 times slower than an ASIC in the same process
  - Custom will be even faster, but more variation in speed difference

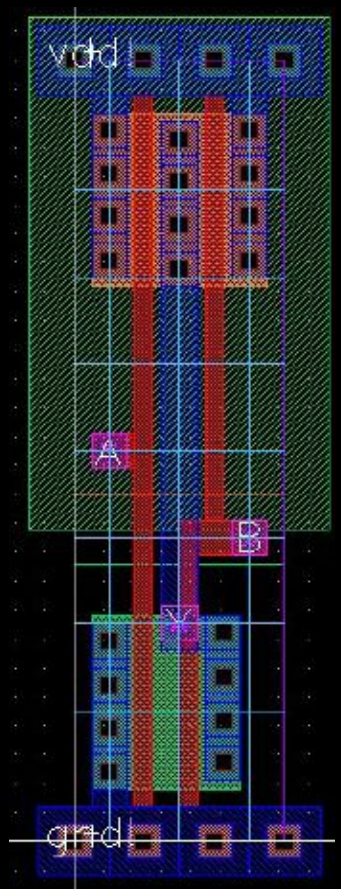
ASIC

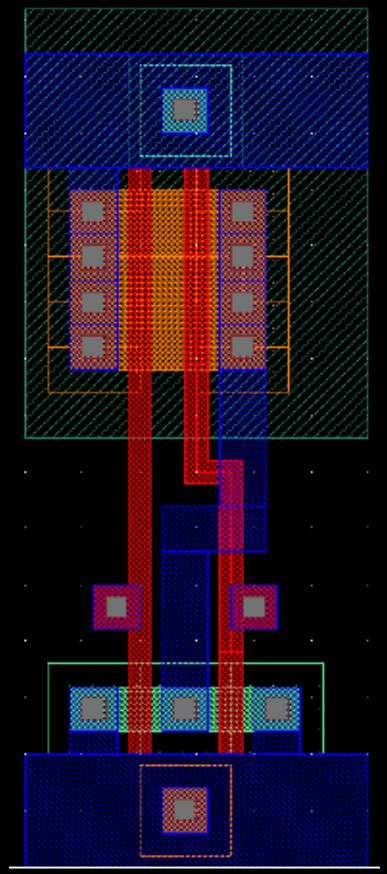
- Generally means automatically synthesized from a moderately high-level description
  - Verilog or VHDL
- Implemented with CMOS standard cells
  - Individual pieces of CMOS layout that implement small/moderate sized Boolean functions
  - Commercial library may have 200-400 cells
- Automatic place and route onto chip

ASIC Standard Cells

- Gate building blocks for ASICs

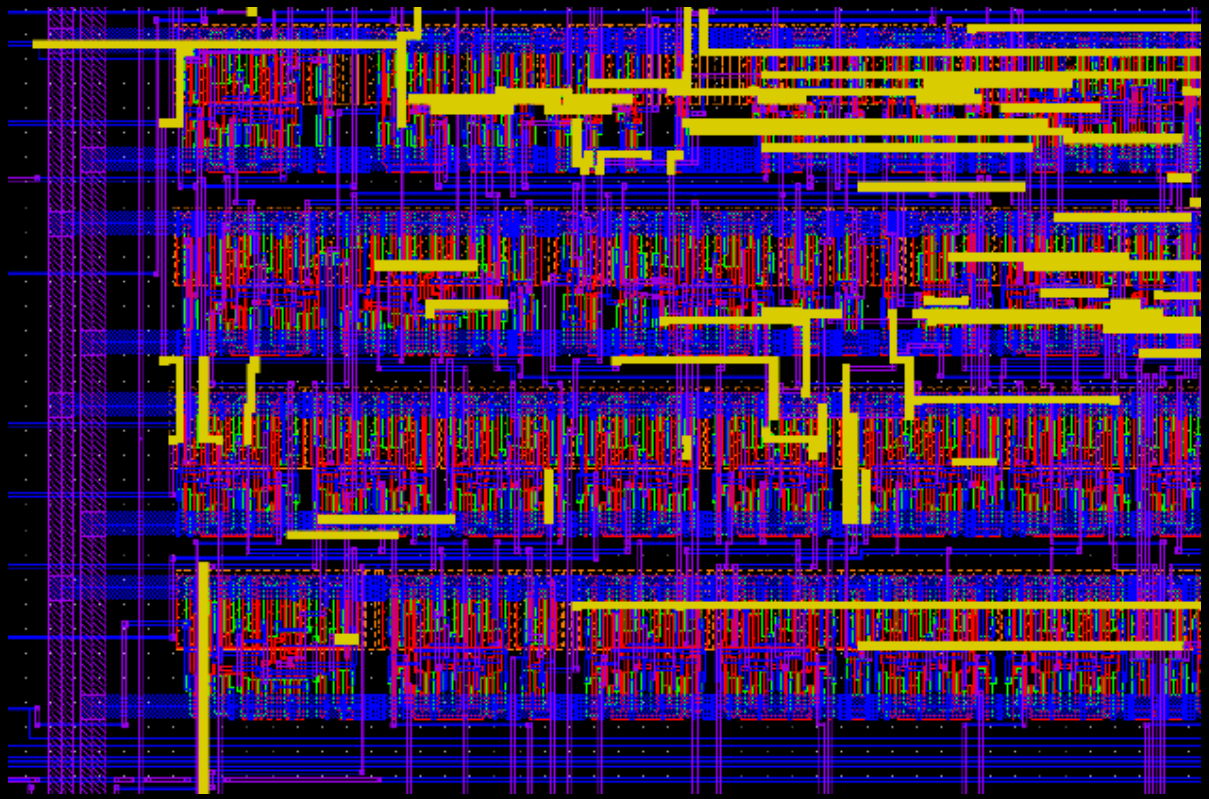






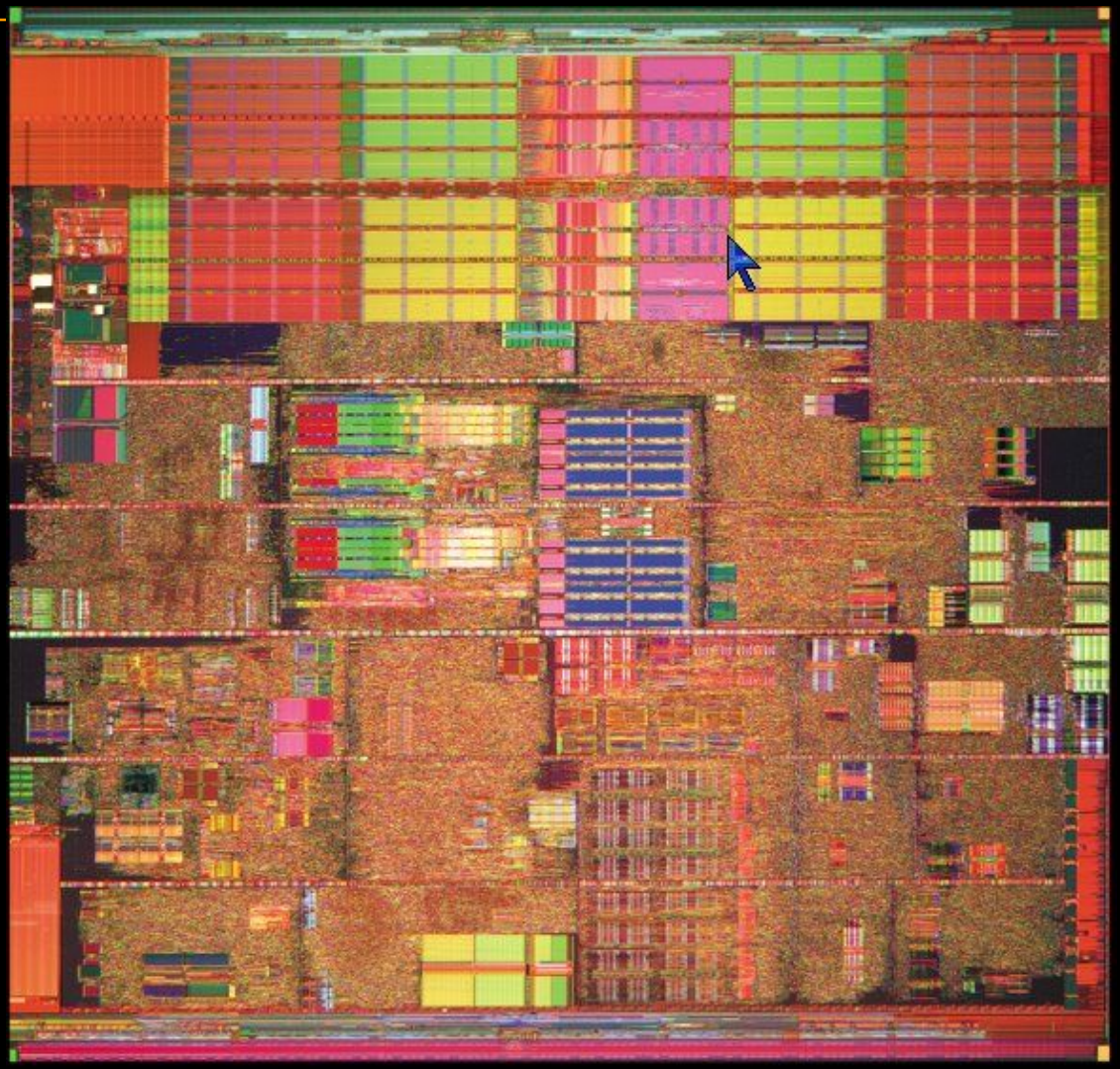
ASIC Standard Cells

- Gate building blocks for ASICs



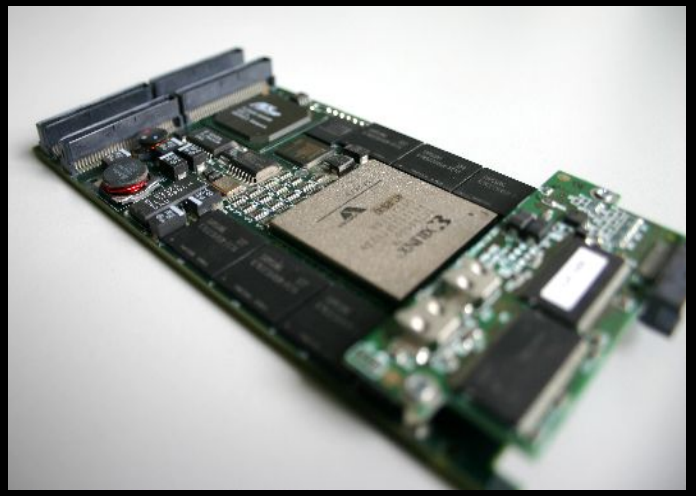
Full-Custom Design

- Hand design for the subsystems
  - Big speed/size advantages
  - Huge design cost



FPGA Implementation

- Hardware
  - Xilinx Virtex4 LX160
    - 24-bit Floating-Point
  - 66 MHz clock rate
  - 1.0 GB/s bandwidth
- Implementation
  - Packets with 4 rays (95% efficiency)
  - 32 Threads (60% usage)
  - 3x 8 KB Cache, direct mapped (90% hit rate)



Virtex4 Board

Performance

- Single FPGA at only 66 MHz
  - 4 million rays/s
    - 20 fps @ 512x384
  - Same performance as CPU
    - 40x clock rate (2.66 GHz)
    - Using highly optimized software (OpenRT with SSE)
- Linear scalability with HW resources
  - Tested: 4x FPGA ➔ 4x performance
  - Independent of scenes

Prototype Performance

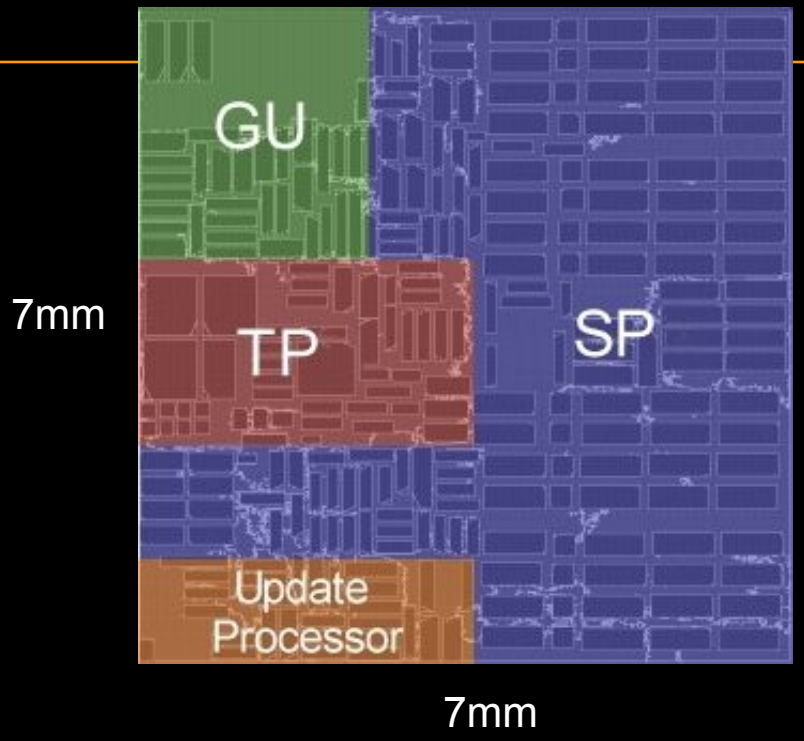
- Technology: FPGA versus ASIC (GPU)

RPU prototype (FPGA)	GPU (ASIC)	
4 Gflops	200 Gflops	50x
0.3 GB/s	30 GB/s	100x

  - Large headroom for scaling performance

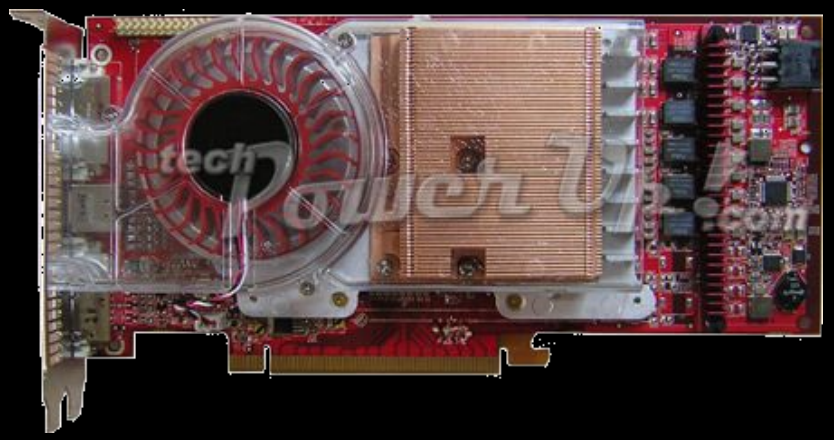
DRPU-ASIC

- Hardware
  - 130nm process
  - 49 mm^2 die area
  - 266 MHz clock rate
  - 2.1 GB/s bandwidth
- Differences
  - Larger caches (3x 16 KB, 4-way associative)
  - 32-bit Floating-Point



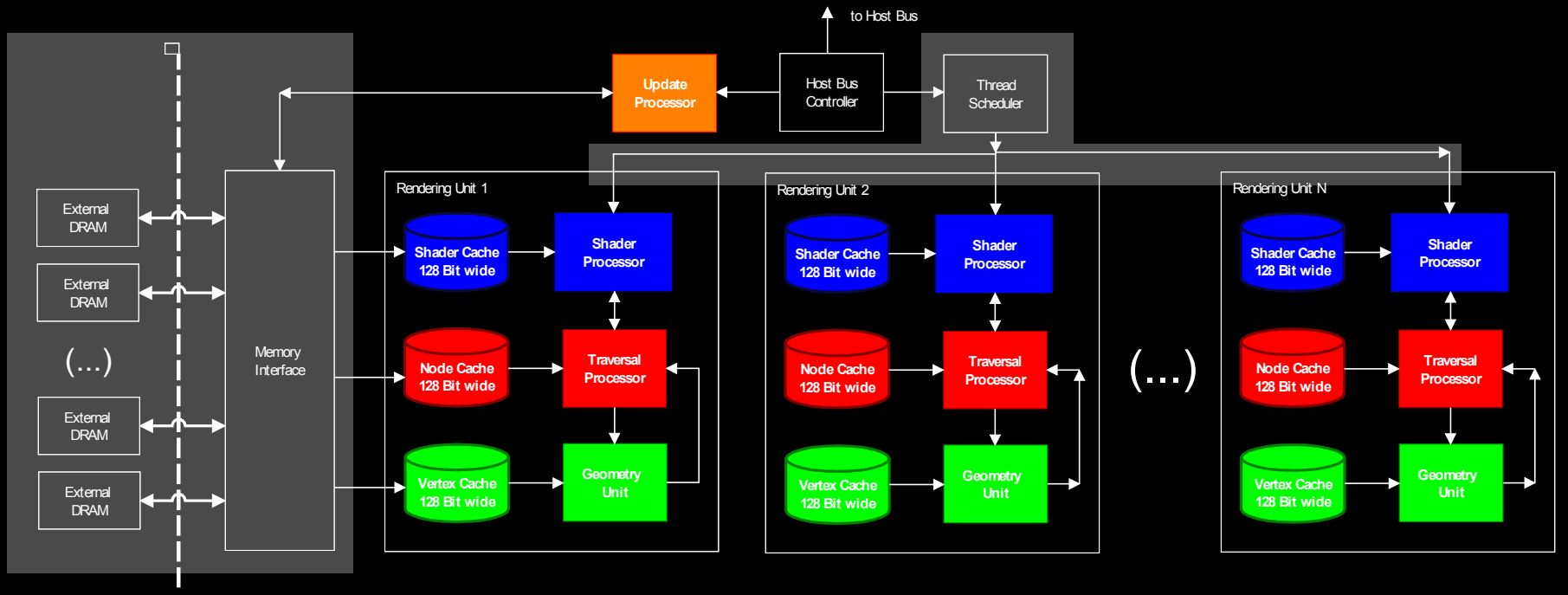
GPU Complexity

- ATI R520 (October, 2005)
  - 90nm process
  - 288 mm^2 die area
  - 600 MHz clock rate
  - 44.8 GB/s bandwidth



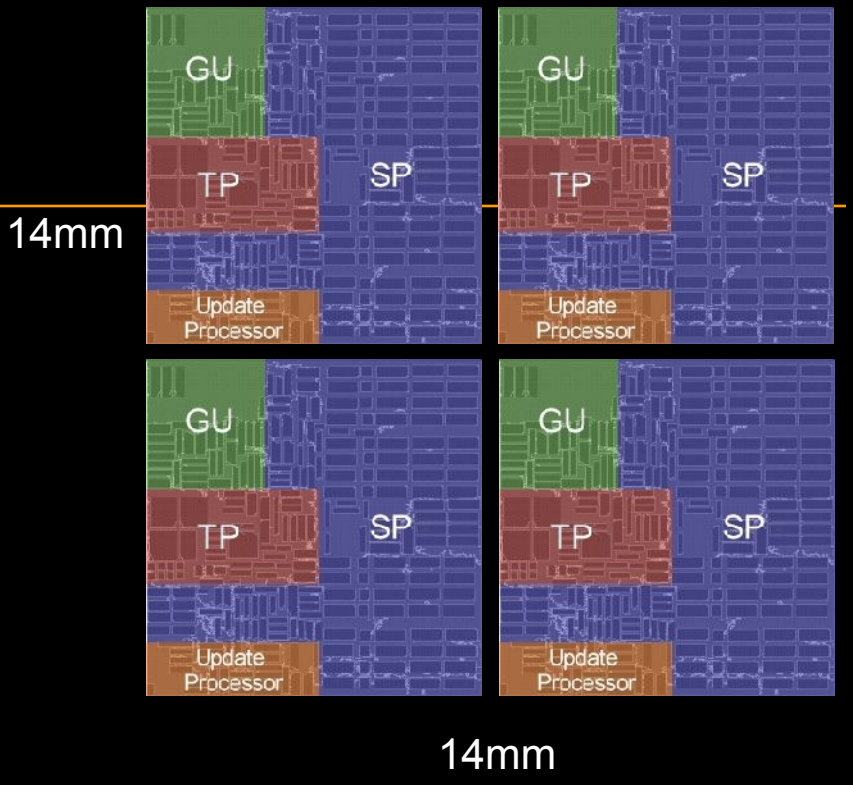
- Implementation
  - Packets with 4 Fragments each
  - 16 Fragment-Pipelines, 8 Vertex-Pipelines
  - 32-bit Floating-Point

On-Chip Parallelisation



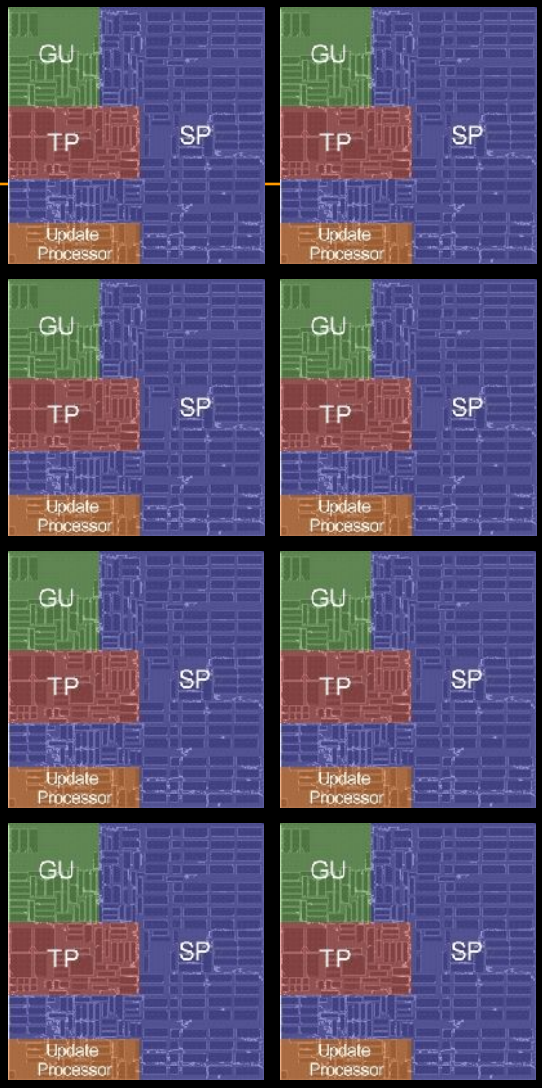
DRPU4 ASIC

- Hardware
  - 130nm process
  - 196 mm^2 die area
  - 266 MHz clock rate
  - 8.5 GB/s bandwidth
- Differences
  - 4x DRPU ASIC
  - No control logic yet



DRPU8-ASIC

- Hardware
  - 90nm process (extrapolated using Constant-Field-Scaling)
  - 186 mm^2 die area
  - 400 MHz clock rate
  - 25.6 GB/s bandwidth
- Implementation Differences
  - 8x DRPU-ASIC

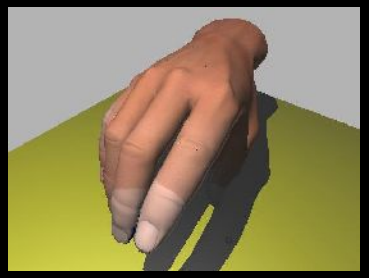
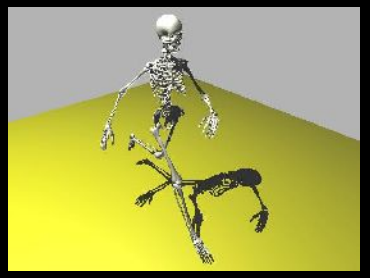
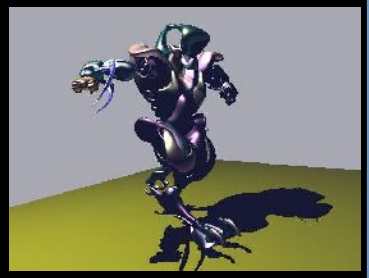
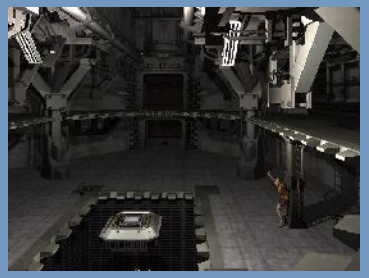


Results 1024x768, shadows





Scene	triangles	objects	#rays	DRPU FPGA	DRPU ASIC	DRPU4 ASIC	DRPU8 ASIC
Shirley6	0.5k	1	1.5M	4.7 fps	18.8 fps	75.2 fps	225.6 fps
Conference	282k	52	1.5M	1.7 fps	6.7 fps	27.0 fps	81.2 fps
Office	34k	1	1.5M	3.6 fps	14.4 fps	57.6 fps	172.8 fps
Mafia Room	15k	1	1.5M	2.8 fps	11.2 fps	44.8 fps	134.4 fps
Mafia Spheres	20k	6	1.6M	1.8 fps	7.2 fps	28.8 fps	86.4 fps
Hand	17k	2	1.3M	5.0 fps	20.0 fps	80.0 fps	240.0 fps
Skeleton	16k	2	1.3M	5.9 fps	23.6 fps	94.4 fps	283.2 fps
Helix	78k	2	1.5M	3.5 fps	14.0 fps	56.0 fps	168.0 fps
Gael	52k	1	1.5M	1.9 fps	7.6 fps	30.4 fps	91.2 fps
DynGael	85k	4	1.5M	2.0 fps	8.0 fps	32.0 fps	96.0 fps

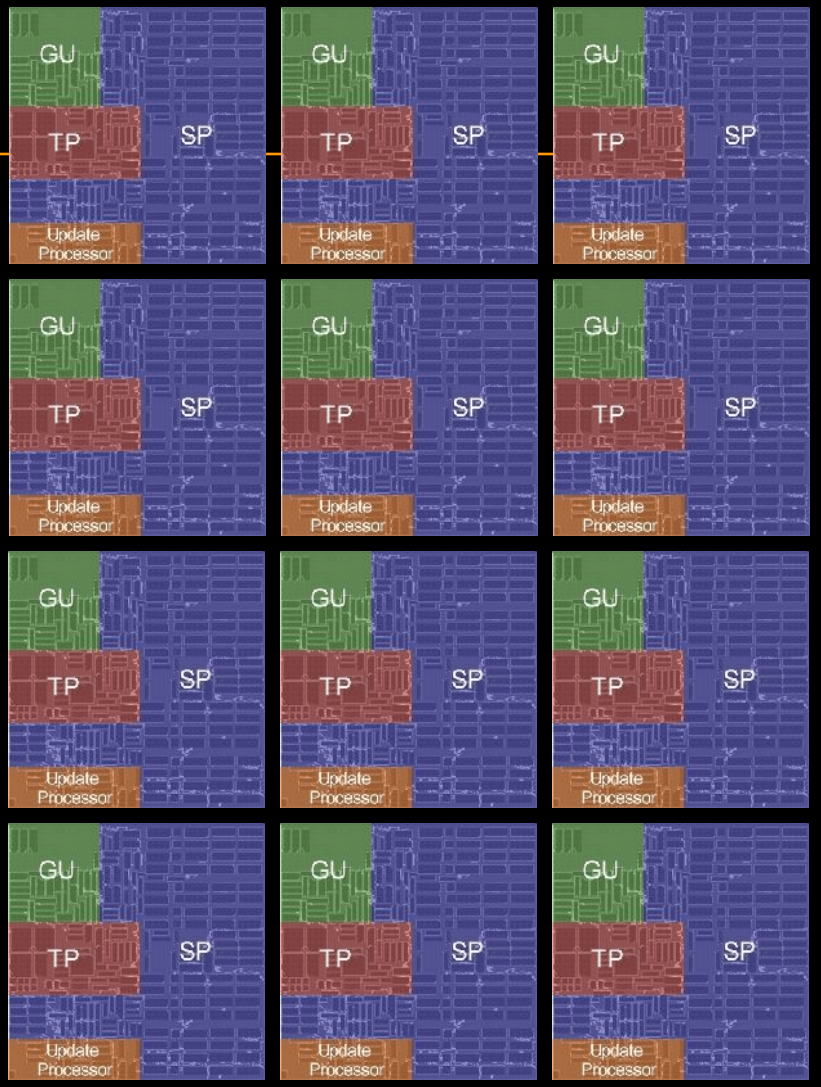






DRPU12-ASIC

19.3 mm

- Hardware
  - 90nm process
  - 278 mm^2 die area
  - 400 MHz clock rate
  - ~40 GB/s bandwidth

- Remember ATI R520?
  - 288mm^2, 600MHz, 44GB/s

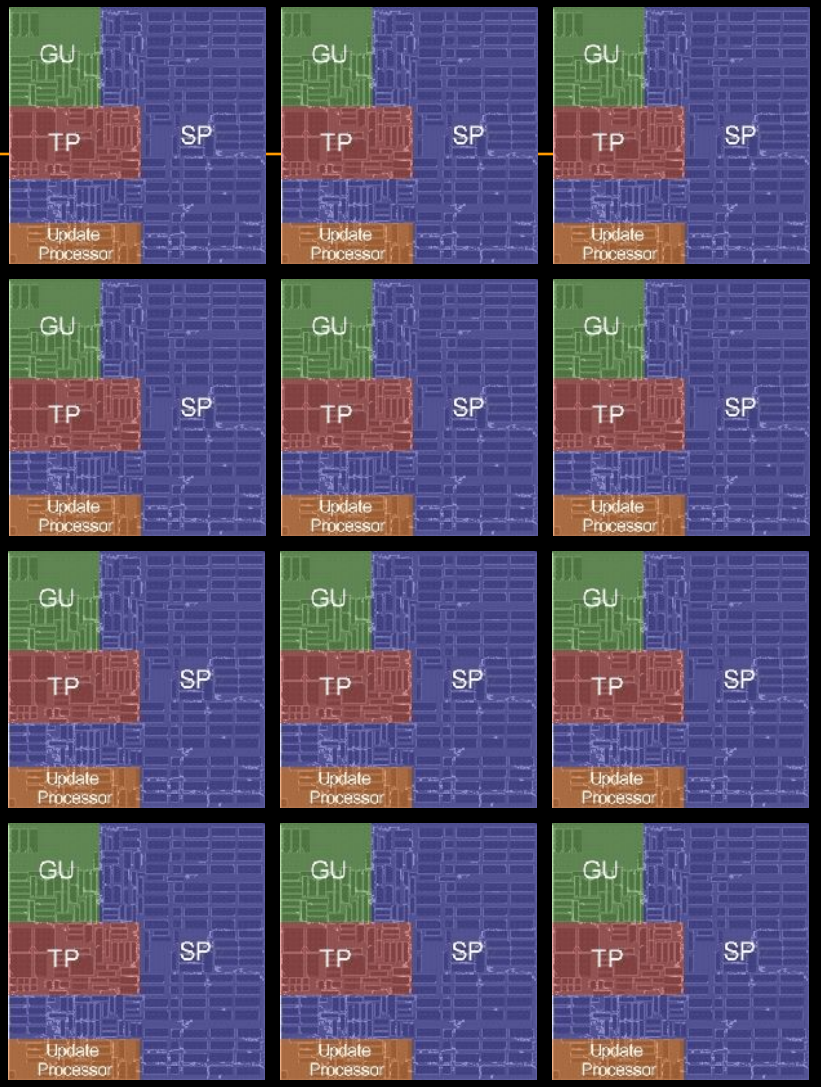


14.4 mm

DRPU12-ASIC

19.3 mm

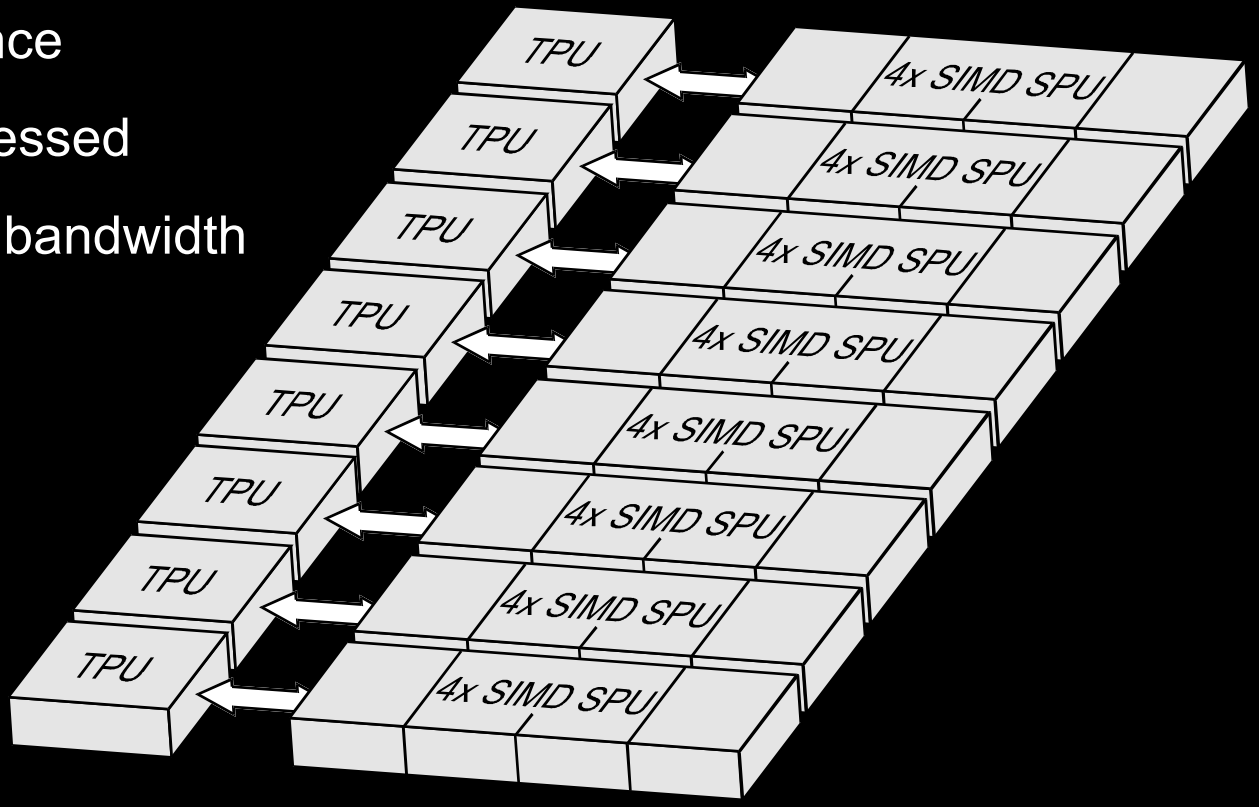
- Compare to FPGA
  - Same 90nm Process
  - Larger chip = more RPUs
  - 400/66 MHz = 6x speedup
- ► 72x Performance
- News from Nvidia (G80)
  - 2,5x die area
  - 65nm process (2x)



14.4 mm

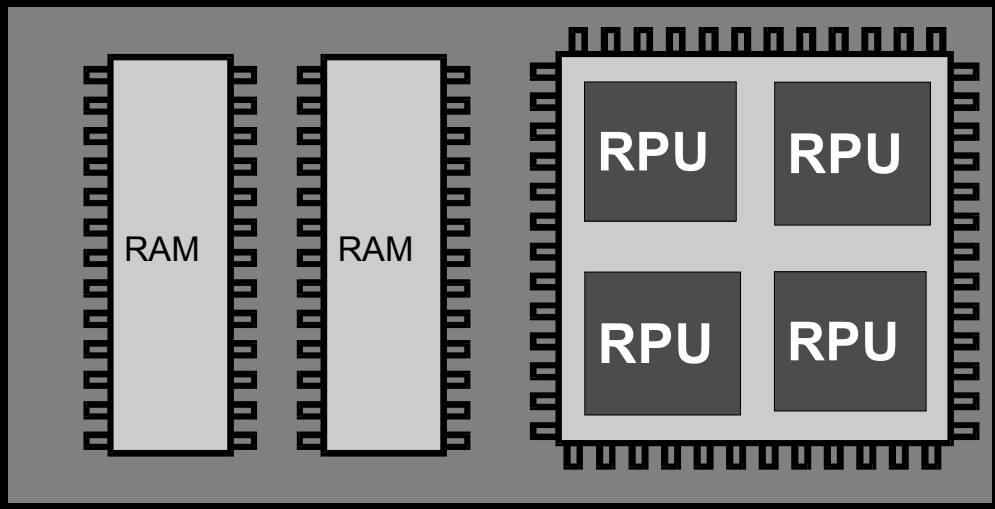
Scalability

- Larger Chunk Size
  - Less ray coherence
  - More data is accessed
  - Increased cache bandwidth
  - Larger caches



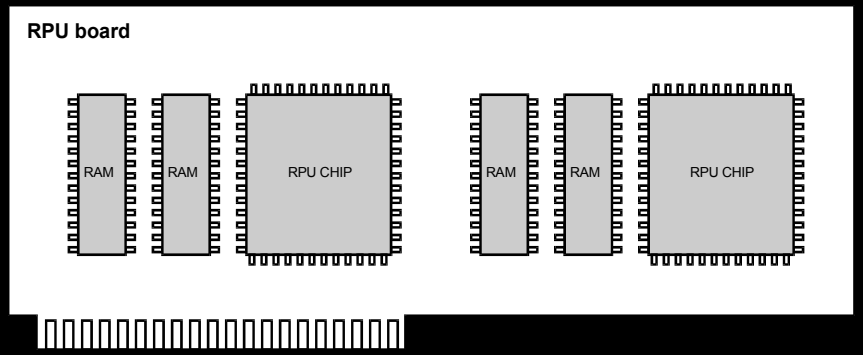
Scalability

- Larger Chunk Size
- Multiple RPUs on a Chip
  - Limited by
    - VLSI technology
    - Memory bandwidth



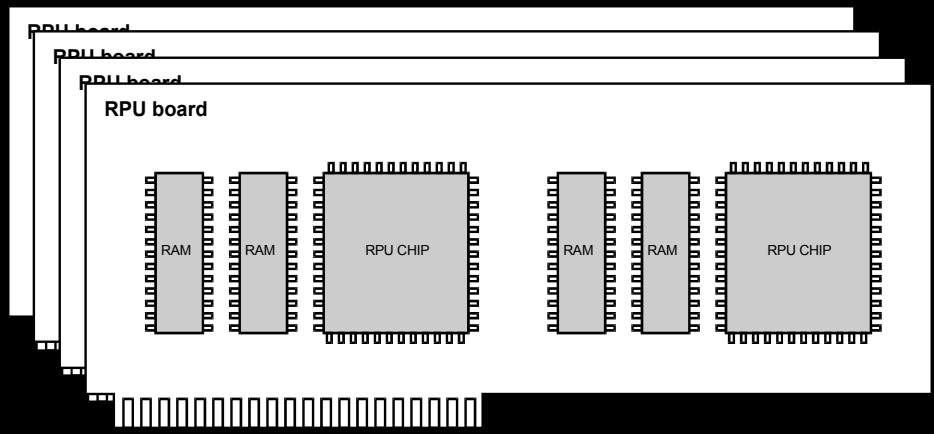
Scalability

- Larger Chunk Size
- Multiple RPUs on a Chip
- Multiple chips on a board
  - Fast interconnect for data exchange
  - Cache sizes accumulate
  - Managed through virtual memory [Schmittler'2003]
  - Limited through external bandwidth due to scene changes



Scalability

- Larger Chunk Size
- Multiple RPUs on a Chip
- Multiple chips on a board
- Multiple boards in a PC
  - Similar to today's PC clusters in a much smaller form factor



Conclusions

- Programmable Architecture for Ray Tracing
  - GPU-like shading processor
  - More flexible programming model
    - Flexible control flow and random memory access
    - Efficient recursion with HW-managed stack
  - Custom hardware for fast kd-tree traversal

➔ Realtime performance on FPGA prototype

Outlook

- ASIC design
  - Need industrial partner to manufacture chip
- RPU as a general purpose processor
  - Explore non-rendering use of RPU

New fundamental operation: Tracing a ray

- Basis for next generation interactive 3D graphics
- What could you do with >1 GRays/s on a PC?