Throughput Computing

Harry Guo - Minh Truong

Larrabee

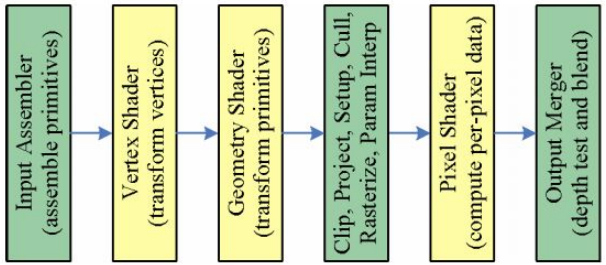
Outline

1. Motivation
2. Architectural Overview
    a. Scalar Unit
    b. VPU
    c. Inter-processor Ring Network
    d. Fixed function logic
3. Software Rendering
4. Evaluation
    a. Merits
    b. Pitfalls

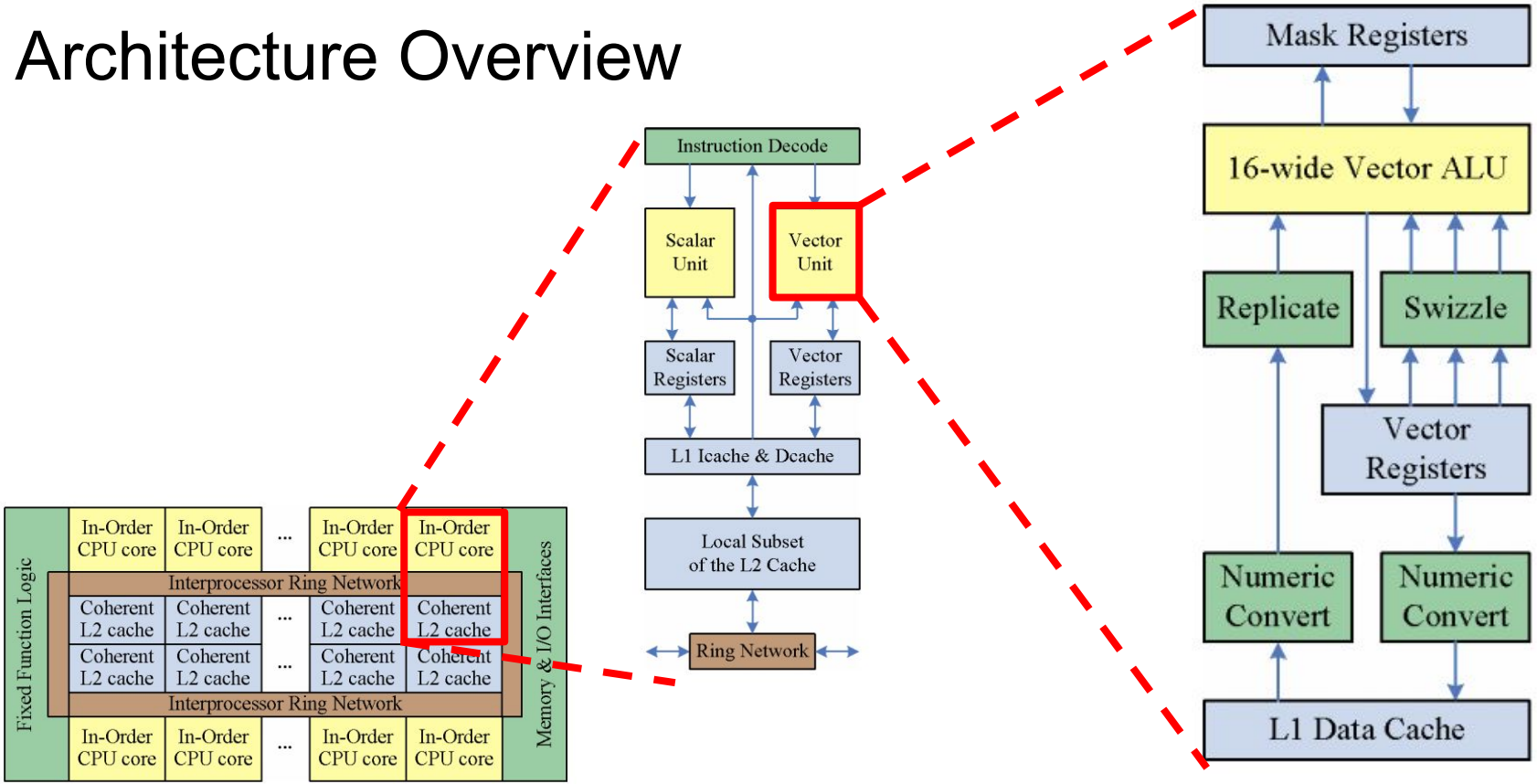
Motivation

At that time:

- GPGPU is an emerging field.
- Arithmetically intensive workloads perform well on GPU-like arch.
- CUDA/OpenCL have limited support for GPU.
- x86 is an established ISA - why not add extension to support GPGPU?
- Intel is exploring discrete GPU.
- Graphic Pipelines are largely realized through HW => not entirely update-able



Architecture Overview



Scalar Unit

- Based on in-order Pentium dual issue (use x86 ISA).
  - For ease of programmability and legacy code support.
  - Create a foundation to push the Rasterization Pipeline to Software level.
  - They argue that many “weaker” cores are better than 1 “beefy” core.
  - What type of workload benefit from many “weaker” cores?
  - Why is Area and Power a good scaling metric?

# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

Scalar Unit

- 1 Primary Pipeline + 1 Secondary Pipeline
  - Ease the difficulty of finding instructions that can execute together.
  - The penalty is low since the secondary pipeline cost is low.
- 64-bits extension + complex prefetcher
  - To allow higher cache use efficiency (sweep data in & out easily, early eviction)

Scalar Unit

- New scalar instructions.
- New Cache instructions for explicit prefetch and priority management.
  - This is not unique to Larrabee, although they might benefit more from such mechanism
- 4 Multi-threads per core.
  - Unsure why they choose 4 threads per core. May be it is because of area constraints?
- 32 KB for I/D L1 Cache.
  - 8 KB per thread, similar to Pentium
- 256 KB of subset of L2 Cache.
  - They did not describe anything about the Cache Coherency Policy :(

Vector Processing Unit (VPU)

- 16-wide VPU
  - Trade-off between increased computational density and difficulty of obtaining high utilization.
  - Analogous to 32 threads per warp of NVIDIA GPU
- Allowed up to 3 source operands, L1 cache works as extended register file
  - Convert 8-bit Unorm and uint, 16-bit sint and float data into 32-bit floats or integers easily.
  - Reduces the need for separate data conversion instruction results in speedup.
  - Workload with irregular data access might inhibit this feature.
- Wide variety of instruction on integer and floating point data types
  - Standard arithmetic operations
  - Fused multiply-add
  - Standard logical operations

Vector Processing Unit (VPU)

- Gather and scatter support
  - Load or store elements to up to 16 different addresses - similar to GPU shared mem banks
  - Will there be potential bank conflicts?
  - allow 16 shader instances run in parallel
  - Handling irregular data structure with more ease
- Predicate mask register
  - One bit per vector lane.
  - Reduce branch misprediction penalties for small clauses.
  - Give compiler instruction scheduler greater freedom over predication.

Inter-Processor Ring network

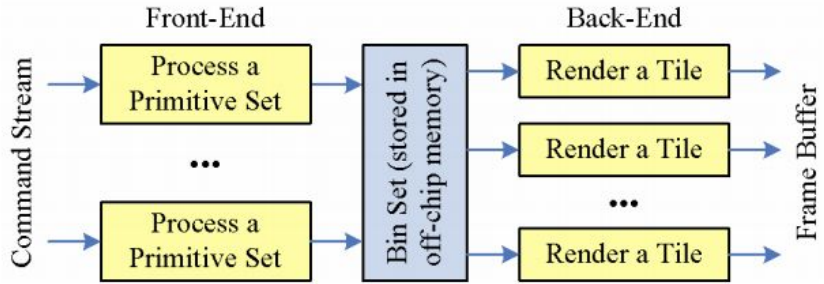
- Bi-directional ring network
  - Easy communication between cores, caches and other logic block
  - Use multiple short rings if scaling more than 16 cores.
  - Cutting communication distance by half with clocking.
- 512-bits wide per direction
  - This make sense since 1 cache line is 64 Bytes = 512 bits.
  - Routing decisions are made before injecting to network.
  - Clock parity controls communication direction.

Fixed function Logic

- Use software in place of several fixed function unit
  - No function units for Rasterization, interpolation or post-shader alpha blending
  - Greater programmability to fit different applications
  - Allow for more freedom of where these processes take place in rendering pipeline
- Does include function unit for texture filter logic
  - This operation cannot be efficiently performed in software on the cores (12x to 40x longer)
  - Texture filtering commonly use 8-bit color component, which can be filtered more efficiently in dedicated logic
  - Selecting unaligned 2x2 quads to filter requires specialized logic
  - Loading texture data in to VPU require impractical amount of register file BW
  - On-the-fly texture decompression is more efficient in dedicated HW

Software Rendering

- a set of rendering commands + their targets = RTset
- RTset produces batches of triangles
- Batch of triagle + RTset tags = PrimSet
- Sub area of a frame = Tile
- Each Tile has 1 bin filled with triangles from Primsets that are in the Tile.

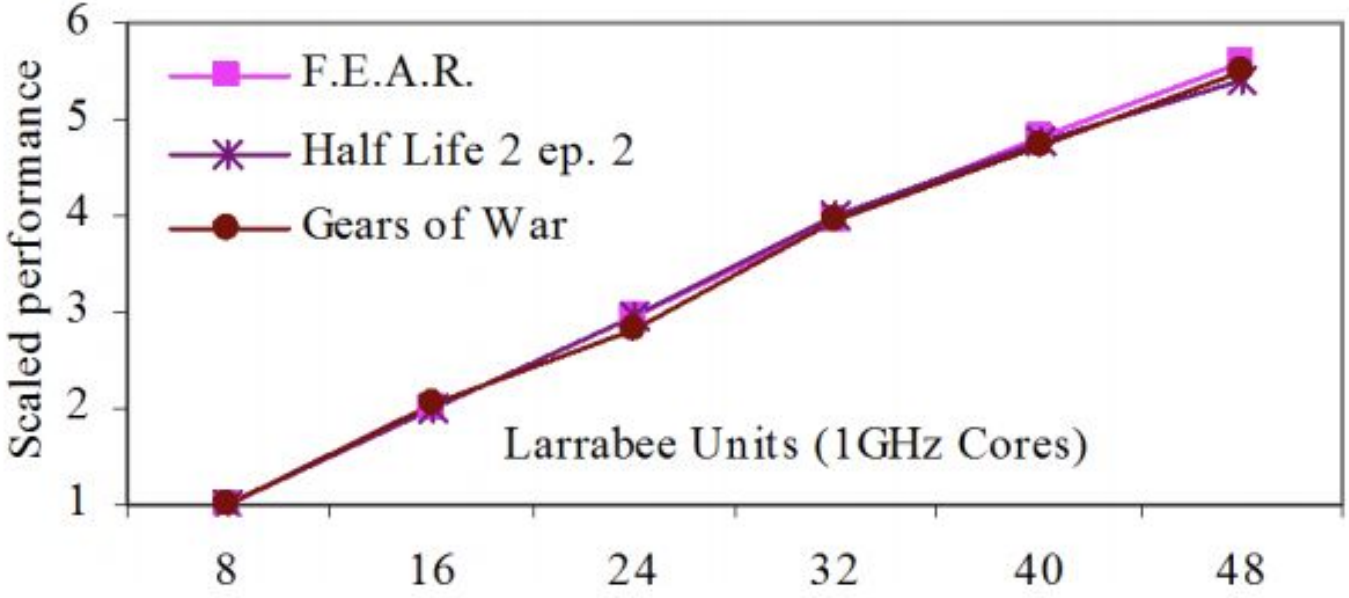


Software Rendering

- Tile size is chosen so that RTset for the tile fits in a core L2 cache
  - This is scalable, since the tile size is programmable according to increasing L2 size
- PrimSet has a sequence ID for correct ordering.
- Each Tile (RTset/binset) maps to 1 core.
  - This suggest that RTsets should ideally be independent.
- Not all cores need to process PrimSets from the RTsets at the same time.
  - This is good for load balancing, any idle core can just take up the next PrimSet
  - PrimSets are stored in an active list.
  - PrimSets ID are used for reordering (what is the overhead?)

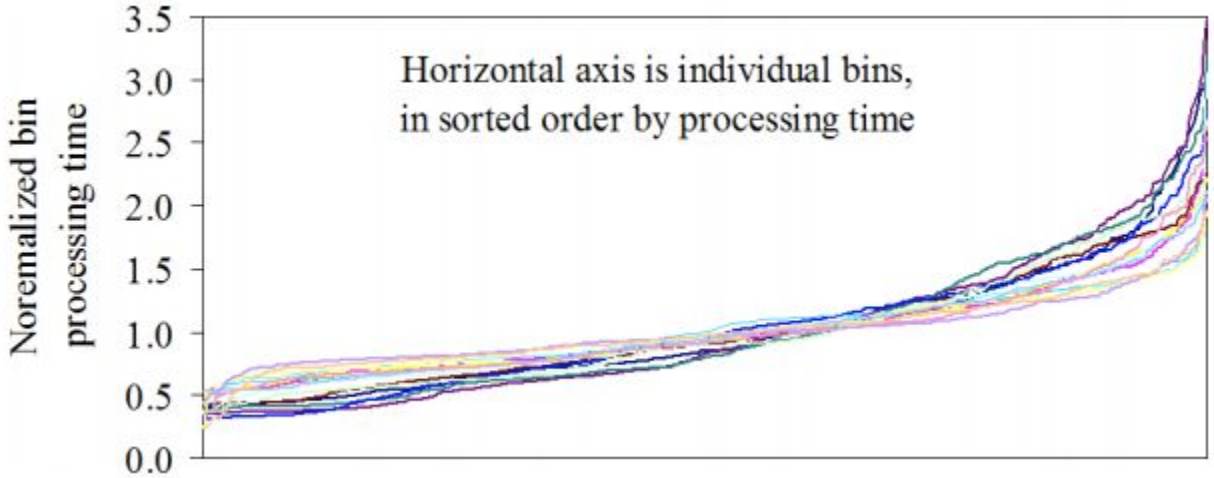
Merits

- Achieve near linear scalability projection



Merits

- Update-able Processing Pipeline
  - They benchmark with DirectX10, but ideally Larrabee GPU requires only a software update to get DirectX11 pipeline.
- Potentially Resolving Load-Balancing issues through Software.
- Suitable for GPGPU tasks.



Pitfalls

- No standardized benchmark (SPECs)
  - They are pushing a GPGPU architecture, shouldn't they benchmark on different apps?
- No mention of the Cache Coherency Policy
- No detailed analysis on Ring communication protocol and overheads.

Debunking 100x GPU vs. CPU myth

Outline

1. Motivation
2. Workload Summary
3. Architecture Differences
4. Performance Analysis
5. Optimization

Workload Summary

Kernel	Application	SIMD	TLP	Characteristics
SGEMM (**SGEMM**) [48]	Linear algebra	Regular	Across 2D Tiles	Compute bound after tiling
Monte Carlo (**MC**) [34, 9]	Computational Finance	Regular	Across paths	Compute bound
Convolution (**Conv**) [16, 19]	Image Analysis	Regular	Across pixels	Compute bound; BW bound for small filters
FFT (**FFT**) [17, 21]	Signal Processing	Regular	Across smaller FFTs	Compute/BW bound depending on size
SAXPY (**SAXPY**) [46]	Dot Product	Regular	Across vector	BW bound for large vectors
LBM (**LBM**) [32, 45]	Time Migration	Regular	Across cells	BW bound
Constraint Solver (**Solv**) [14]	Rigid body physics	Gather/Scatter	Across constraints	Synchronization bound
SpMV (**SpMV**) [50, 8, 47]	Sparse Solver	Gather	Across non-zero	BW bound for typical large matrices
GJK (**GJK**) [38]	Collision Detection	Gather/Scatter	Across objects	Compute Bound
Sort (**Sort**) [15, 39, 40]	Database	Gather/Scatter	Across elements	Compute bound
Ray Casting (**RC**) [43]	Volume Rendering	Gather	Across rays	4-8MB first level working set, over 500MB last level working set
Search (**Search**) [27]	Database	Gather/Scatter	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree
Histogram (**Hist**) [53]	Image Analysis	Requires conflict detection	Across pixels	Reduction/synchronization bound
Bilateral (**Bilat**) [52]	Image Analysis	Regular	Across pixels	Compute Bound

Apps.	SGEMM	MC	Conv	FFT	SAXPY	LBM	Solv	SpMV	GJK	Sort	RC	Search	Hist	Bilat
Core i7-960	94	0.8	1250	71.4	16.8	85	103	4.9	67	250	5	50	1517	83
GTX280	364	1.4	3500	213	88.8	426	52	9.1	1020	198	8.1	90	2583	475

Architecture Differences

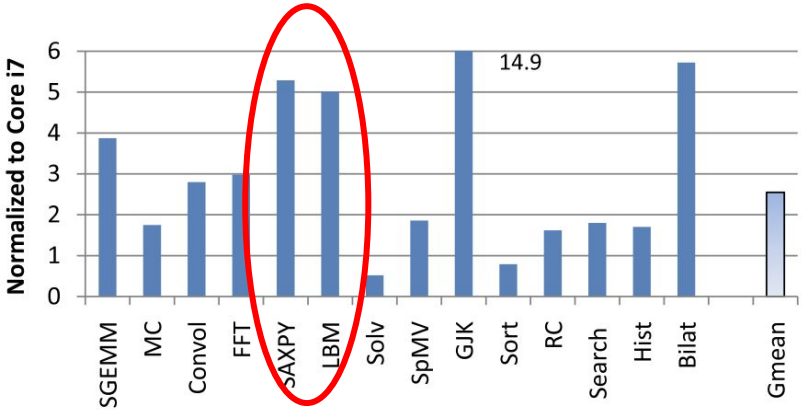
- Design purpose:
  - CPU: design for a wide variety of application; provide fast responding times to a single task
  - GPU: design for for rendering and application with high degree of data parallelism
- Latency hiding:
  - CPU: high latency by using large cache and doing aggressive prefetching and branch prediction
  - GPU: high latency by multi-threading, swap to different thread when one thread is doing long latency event, such as memory accessing.

	Num. PE	Frequency (GHz)	Num. Transistors	BW (GB/sec)	SP SIMD width	DP SIMD width	Peak SP Scalar FLOPS (GFLOPS)	Peak SP SIMD Flops (GFLOPS)	Peak DP SIMD Flops (GFLOPS)
Core i7-960	4	3.2	0.7B	32	4	2	25.6	102.4	51.2
GTX280	30	1.3	1.4B	141	8	1	116.6	311.1/933.1	77.8

Not sure why the choose these specific machines, may be the unifying factor is their Die Area?
Corei7 die area is 263 mm2 and GTX280 die area is 576mm2 (is this fair?)

Performance Analysis - Bandwidth Bound

- Workload that are bandwidth bound in general perform better on GPU (since GPU has higher bandwidth) **SAXPY LMB**
- However, GPU cache capacity is smaller, some workload data has to sit in global memory, resulting in higher memory traffic => lower performance
  - Higher bandwidth workload + higher bandwidth machine don't neccessary translate to higher performance, like **SpMV** case.



Performance analysis - Compute Bound

- **SGEMM**, **MC**, **Conv**, **FFT** and **Bilat** are able to exploit all available flops on both CPU and GPU
- **SGEMM, Conv and FFT** -- use single precision flop
  - GTX280-to-Core i7 performance ratio in 2.8 - 4x range, comply with the fact GTX280-to-Core i7 SP flop ratio is 3 - 6X.
  - GPU does not achieve peak performance in the presence of shared buffer
- **Bilat** -- utilize fast transcendental operation on GPU (.i.e flush-to-zero, lower resolution trigonometric intrinsics)
  - 5X faster due to dedicated fixed function units (tailored to specific applications)
- **MC** -- double precision arithmetic
  - Performance ratio of 1.8x, flop ration 1.5x

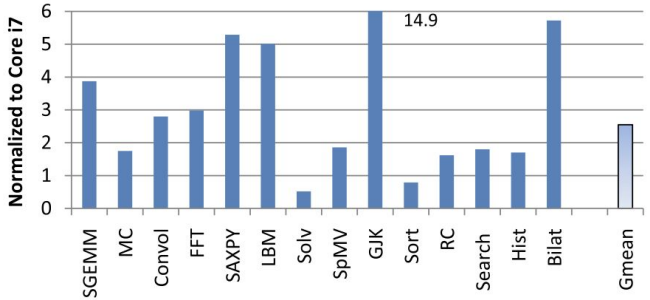
Performance Analysis - Cache Capacity

- Large on-chip memory provides huge advantage for workload that requires a lot of initial data or intermediate data.
  - A workload can transform from Compute Bound to Bandwidth Bound due to lack of resource of on-chip memory and increase in data, like **Search**.
  - Should standardize benchmark capture this behavior and offers both compute/bandwidth bound version of the same workload?

Performance Analysis - Gather/Scatter

- GPU has a switch fabric that allow combinational access pattern to shared memory (fast gather scatter), CPU doesn't have this.
  - However, GPU does not have this feature at global memory, making the feature less effective for larger datasets.
  - The switch fabric is bounded by 32 shared memory banks. Bank Conflict might result in stall and more context switch => Putting burden on the programmer to implement conflict free gather/scatter.
  - **Sort** doesn't benefit from this because it is dominated with scalar operations.

Constraint Solver (**Solv**) [14]	Rigid body physics	Gather/Scatter	Across constraints	Synchronization bound
SpMV (**SpMV**) [50, 8, 47]	Sparse Solver	Gather	Across non-zero	BW bound for typical large matrices
GJK (**GJK**) [38]	Collision Detection	Gather/Scatter	Across objects	Compute Bound
Sort (**Sort**) [15, 39, 40]	Database	Gather/Scatter	Across elements	Compute bound
Ray Casting (**RC**) [43]	Volume Rendering	Gather	Across rays	4-8MB first level working set, over 500MB last level working set
Search (**Search**) [27]	Database	Gather/Scatter	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree



Performance analysis - Reduction & Synchronization

- Reduction and Synchronization do not scale with increasing thread count and data-level parallelism
- Synchronization overhead dominates **Hist** and **Solv** runtime and and becomes bigger bottleneck as the number of cores/threads and the SIMD increase
  - **Hist** is limited by atomic updates
  - **Solv** is dominated by barrier overhead
    - CPU provides memory & cache consistent model while GPU doesn't. As a result the barrier execution time of GTX280 is order of magnitude slower than on Core i7

Although DLP can increase with more execution cores, more cores may impose more complex grouping, syncing constraints, preventing performance increase.

Performance analysis - Fixed Functions

- **Bilat** - consists of transcendental operation like exponential and power function
  - 66% of CPU runtime is spent one transcendental computation
- **MC** also benefits from fast transcendental computation
- **GJK** collision detection algorithm can exploit the fast texture lookup capability of GPUs, which result in further speedup.

This takes the opposite approach to Larrabee, which tries to minimize the number of fixed function units, even for Graphic Applications. What's the best approach to determine if a function should be implemented in HW?