ACCELERATED PROCESSING AND THE FUSION SYSTEM ARCHITECTURE

Mike Schulte, Fellow
AMD Research

Based on Slides from Phil Rogers, AMD Corporate Fellow
Make the unprecedented processing capability of the Accelerated Processing Unit (APU) as accessible to programmers as the CPU is today.
OUTLINE

- The APU today and its programming environment
- The future of the heterogeneous platform
- AMD Fusion System Architecture
- Roadmap
- Software evolution
- A visual view of the new command and data flow
The APU has arrived and it is a great advance over previous platforms.

Combines scalar processing on CPU with parallel processing on the GPU and high bandwidth access to memory.

How do we make it even better going forward?

- Easier to program
- Easier to optimize
- Easier to load balance
- Higher performance
- Lower power
LOW POWER E-SERIES AMD FUSION APU: “ZACATE”

E-Series APU

- 2 x86 Bobcat CPU cores
- Array of Radeon™ Cores
  - Discrete-class DirectX® 11 performance
  - 80 Stream Processors
- 3rd Generation Unified Video Decoder
- PCIe® Gen2
- Single-channel DDR3 @ 1066
- 18W TDP

Performance:

- Up to 8.5GB/s System Memory Bandwidth
- Up to 90 Gflop of Single Precision Compute
TABLET Z-SERIES AMD FUSION APU: “DESNA”

Z-Series APU

- 2 x86 “Bobcat” CPU cores
- Array of Radeon™ Cores
  - Discrete-class DirectX® 11 performance
  - 80 Stream Processors
- 3rd Generation Unified Video Decoder
- PCIe® Gen2
- Single-channel DDR3 @ 1066
- 6W TDP w/ Local Hardware Thermal Control

Performance:

- Up to 8.5GB/s System Memory Bandwidth
- Suitable for sealed, passively cooled designs
MAINSTREAM A-SERIES AMD FUSION APU: “LLANO”

A-Series APU

- Up to four x86 CPU cores
  - AMD Turbo CORE frequency acceleration
- Array of Radeon™ Cores
  - Discrete-class DirectX® 11 performance
- 3rd Generation Unified Video Decoder
- Blu-ray 3D stereoscopic display
- PCIe® Gen2
- Dual-channel DDR3
- 45W TDP

Performance:

- Up to 29GB/s System Memory Bandwidth
- Up to 500 Gflops of Single Precision Compute
COMMITTED TO OPEN STANDARDS

- AMD drives open and de-facto standards
  - Compete on the best implementation
- Open standards are the basis for large ecosystems
- Open standards always win over time
  - SW developers want their applications to run on multiple platforms from multiple hardware vendors
A NEW ERA OF PROCESSOR PERFORMANCE

Single-Core Era
- Enabled by:
  - ✓ Moore’s Law
  - ✓ Voltage Scaling
- Constrained by:
  - ❌ Power
  - ❌ Complexity

Multi-Core Era
- Enabled by:
  - ✓ Moore’s Law
  - ✓ SMP architecture
- Constrained by:
  - ❌ Power
  - ❌ Parallel SW
  - ❌ Scalability

Heterogeneous Systems Era
- Enabled by:
  - ✓ Abundant data parallelism
  - ✓ Power efficient GPUs
  - Temporarily
- Constrained by:
  - ❌ Programming models
  - ❌ Comm. overhead

Assembly ➔ C/C++ ➔ Java …

single-thread performance

we are here

threads ➔ OpenMP / TBB …

throughput performance

we are here

Shader ➔ CUDA ➔ OpenCL ➔ !!!

modern application performance

we are here

Single-thread Performance

Throughput Performance

Modern Application Performance

Time

Time (# of processors)

Time (Data-parallel exploitation)
EVOLUTION OF HETEROGENEOUS COMPUTING

Architectured Era
- Mainstream programmers
- Full C++
- GPU as a co-processor
- Unified coherent address space
- Task parallel runtimes
- Nested Data Parallel programs
- User mode dispatch
- Pre-emption and context switching

Standards Drivers Era
- OpenCL™, DirectCompute
- Driver-based APIs
- Expert programmers
- C and C++ subsets
- Compute centric APIs, data types
- Multiple address spaces, with explicit data movement
- Specialized work queue based structures
- Kernel mode dispatch

Proprietary Drivers Era
- Graphics & Proprietary Driver-based APIs
  - “Adventurous” programmers
  - Exploit early programmable “shader cores” in the GPU
  - Make your program look like “graphics” to the GPU
  - CUDA™, Brook+, etc

Architecture Maturity & Programmer Accessibility
- 2002 - 2008
- 2009 - 2011
- 2012 - 2020
- Excellent
- Poor

“Adventurous” programmers
- Exploit early programmable “shader cores” in the GPU
- Make your program look like “graphics” to the GPU
- CUDA™, Brook+, etc

Expert programmers
- C and C++ subsets
- Compute centric APIs, data types
- Multiple address spaces, with explicit data movement
- Specialized work queue based structures
- Kernel mode dispatch

Mainstream programmers
- Full C++
- GPU as a co-processor
- Unified coherent address space
- Task parallel runtimes
- Nested Data Parallel programs
- User mode dispatch
- Pre-emption and context switching
**FSA FEATURE ROADMAP**

**Physical Integration**
- Integrate CPU & GPU in silicon
- Unified Memory Controller
- Common Manufacturing Technology

**Optimized Platforms**
- GPU Compute C++ support
- User mode scheduling
- Bi-Directional Power Mgmt between CPU and GPU

**Architectural Integration**
- Unified Address Space for CPU and GPU
- GPU uses pageable system memory via CPU pointers
- Fully coherent memory between CPU & GPU

**System Integration**
- GPU compute context switch
- GPU graphics pre-emption
- Quality of Service
- Extend to Discrete GPU
FUSION SYSTEM ARCHITECTURE – AN OPEN PLATFORM

- Open Architecture, published specifications
  - FSAIL virtual ISA
  - FSA memory model
  - FSA dispatch

- ISA agnostic for both CPU and GPU

- Inviting partners to join us, in all areas
  - Hardware companies
  - Operating Systems
  - Tools and Middleware
  - Applications

- FSA review committee planned
FSAIL is a virtual ISA for parallel programs
- Finalized to ISA by a JIT compiler or “Finalizer”

Explicitly parallel
- Designed for data parallel programming

Support for exceptions, virtual functions, and other high level language features

Syscall methods
- GPU code can call directly to system services, IO, printf, etc

Debugging support
**FSA MEMORY MODEL**

- Designed to be compatible with C++0x, Java and .NET Memory Models
- Relaxed consistency memory model for parallel compute performance
- Loads and stores can be re-ordered by the finalizer
- Visibility controlled by:
  - `Load.Acquire`, `Store.Release`
  - Fences
  - Barriers
Accelerated Processing and the Fusion System Architecture | July 2011

Driver Stack

- Domain Libraries
- OpenCL™ 1.x, DX Runtimes, User Mode Drivers
- Graphics Kernel Mode Driver

FSA Software Stack

- FSA Domain Libraries
- FSA JIT
- Task Queuing Libraries
- FSA Runtime
- FSA Kernel Mode Driver

Hardware - APUs, CPUs, GPUs

- AMD user mode component
- AMD kernel mode component
- All others contributed by third parties or AMD
**OPENCL™ AND FSA**

- FSA is an optimized platform architecture for OpenCL™
  - Not an alternative to OpenCL™
- OpenCL™ on FSA will benefit from
  - Avoidance of wasteful copies
  - Low latency dispatch
  - Improved memory model
  - Pointers shared between CPU and GPU
- FSA also exposes a lower level programming interface, for those that want the ultimate in control and performance
  - Optimized libraries may choose the lower level interface
**TASK QUEUING RUNTIMES**

- Popular pattern for task and data parallel programming on SMP systems today

- Characterized by:
  - A work queue per core
  - Runtime library that divides large loops into tasks and distributes to queues
  - A work stealing runtime that keeps the system balanced

- FSA is designed to extend this pattern to run on heterogeneous systems
TASK QUEUING RUNTIME ON CPUS

- Work Stealing Runtime
- CPU Worker
  - X86 CPU
- CPU Worker
  - X86 CPU
- CPU Worker
  - X86 CPU
- CPU Worker
  - X86 CPU

Legend:
- Green: CPU Threads
- Red: GPU Threads
- Purple: Memory
TASK QUEUING RUNTIME ON THE FSA PLATFORM

Work Stealing Runtime

- CPU Worker
- GPU Manager
- Memory

Fetch and Dispatch

- X86 CPU

CPU Threads

GPU Threads

Memory
FSA SOFTWARE EXAMPLE - REDUCTION

float foo(float);
float myArray[...];

Task<float, ReductionBin> task([myArray]( IndexRange<1> index) [[device]] {
  float sum = 0.;
  for (size_t I = index.begin(); I != index.end(); i++) {
    sum += foo(myArray[i]);
  }
  return sum;
});

float result = task.enqueueWithReduce( Partition<1, Auto>(1920),
  [] (int x, int y) [[device]] { return x+y; }, 0.);
HETEROGENEOUS COMPUTE DISPATCH

How compute dispatch operates today in the **driver model**

---

How compute dispatch improves tomorrow **under FSA**
TODAY’S COMMAND AND DISPATCH FLOW

Application A → Direct3D → User Mode Driver → Soft Queue → Kernel Mode Driver → Command Buffer → DMA Buffer → Hardware Queue → GPU Hardware
TODAY'S COMMAND AND DISPATCH FLOW

Command Flow

Data Flow

Application A
Direct3D
User Mode Driver
Soft Queue
Kernel Mode Driver
Command Buffer
DMA Buffer

Application B
Direct3D
User Mode Driver
Soft Queue
Kernel Mode Driver
Command Buffer
DMA Buffer

Application C
Direct3D
User Mode Driver
Soft Queue
Kernel Mode Driver
Command Buffer
DMA Buffer

GPU HARDWARE
**FUTURE COMMAND AND DISPATCH FLOW**

- Application codes to the hardware
- User mode queuing
- Hardware scheduling
- Low dispatch times

- No APIs
- No Soft Queues
- No User Mode Drivers
- No Kernel Mode Transitions
- No Overhead!
FUTURE COMMAND AND DISPATCH CPU <-> GPU
FUTURE COMMAND AND DISPATCH CPU <-> GPU

Application / Runtime

CPU1  CPU2  GPU
FUTURE COMMAND AND DISPATCH CPU <-> GPU

Application / Runtime

CPU1

CPU2

GPU
FUTURE COMMAND AND DISPATCH CPU <-> GPU

Application / Runtime

CPU1

CPU2

GPU
WHERE ARE WE TAKING YOU?

Switch the compute, don’t move the data!

- Every processor now has serial and parallel cores
- All cores capable, with performance differences
- Simple and efficient program model

Platform Design Goals

- Easy support of massive data sets
- Support for task based programming models
- Solutions for all platforms
- Open to all
THE FUTURE OF HETEROGENEOUS COMPUTING

- The architectural path for the future is clear
  - Programming patterns established on Symmetric Multi-Processor (SMP) systems migrate to the heterogeneous world
  - An open architecture, with published specifications and an open source execution software stack
  - Heterogeneous cores working together seamlessly in coherent memory
  - Low latency dispatch
  - No software fault lines