

THE FUTURE OF THE APU A Software perspective - OpenCL and beyond

Benedict R. Gaster AMD Programming Models Architect





THE FUSION APU ARCHITECTURE



TAKING A REALISTIC LOOK AT THE APU

- GPUs are not magic
 - We've often heard about 100x performance improvements
 - These are usually the result of poor CPU code
- The APU offers a balance
 - GPU cores optimized for arithmetic workloads and latency hiding
 - CPU cores to deal with the branchy code for which branch prediction and out of order execution are so valuable
 - A tight integration such that shorter, more tightly bound, sections of code targeted at the different styles of device can be linked together
 - However, not without costs!

CPUS AND GPUS

- Different design goals:
 - CPU design is based on maximizing performance of a single thread
 - GPU design aims to maximize throughput at the cost of lower performance for each thread
- CPU use of area:
 - Transistors are dedicated to branch prediction, out of order logic and caching to reduce latency to memory, to allow efficient instruction prefetching and deep pipelines (fast clocks)

GPU use of area:

- Transistors concentrated on ALUs and registers
- Registers store thread state and allow fast switching between threads to cover (rather than reduce) latency

AMD

COSTS

The CPU approach:

- Requires large caches to maximize the number of memory operations caught
- Requires considerable transistor logic to support the out of order control

The GPU approach:

- Requires wide hardware vectors, not all code is easily vectorized
- Requires considerable state storage to support active threads
- Note: we need not pretend that OpenCL or CUDA are NOT vectorization

AMD

The entire point of the design is hand vectorization

These two approaches suit different algorithm designs

They require different hardware implementations

THE TRADEOFFS IN PICTURES

- AMD Phenomtm II X6:
 - 6 cores, 4-way SIMD (ALUs)
 - A single set of registers
 - Registers are backed out to memory on a thread switch, and this is performed by the OS
- AMD Radeontm HD6970:
 - 24 cores, 16-way SIMD (plus VLIW issue, but the Phenom processor does multi-issue too), 64-wide SIMD state
 - Multiple register sets (somewhat dynamic)
 - 8, 16 threads per core





COMBINING THE TWO

- So what did we see?
 - Diagrams were vague, but...
 - Large amount of orange! Lots of register state
 - Also more green on the GPU cores
- The APU combines the two styles of core:
 - The E350 has two "Bobcat" cores and two "Cedar"-like cores, for example
 - 2- and 8-wide physical SIMD







THINKING ABOUT PROGRAMMING



OPENCL, CUDA, ARE THESE GOOD MODELS?

- Designed for wide data-parallel computation
 - Pretty low level
 - There is a lack of good scheduling and coherence control
 - We see "cheating" all the time: the lane-wise programming model only becomes efficient when we program it like the vector model it really is, making assumptions about wave or warp synchronicity

However: they're better than SSE!

- We have proper scatter/gather memory access
- The lane wise programming does help: we still have to think about vectors, but it's much easier to do than in a vector language
- We can even program lazily and pretend that a single work item is a thread and yet it still (sortof) works
 - (NVIDIA trades ALUs against making this work well. An interesting design point, but if we have a CPU around do we need that?)

AMD

THE CPU PROGRAMMATICALLY: A TRIVIAL EXAMPLE

- What's the fastest way to perform an associative reduction across an array on a CPU?
 - Take an input array
 - Block it based on the number of threads (one per core usually, maybe 4 or 8 cores)
 - Iterate to produce a sum in each block
 - Reduce across threads
 - Vectorize

```
float4sum(00, 0, 0, 0)
for(i = n/toto (nb+)b)/4)
sum += input[i]
float scalarSum = sum.x + sum.y + sum.z + sum.w
float reductionValue(0)
for(t in threadCount)
reductionValue += t.sum
```



THE GPU PROGRAMMATICALLY: THE SAME TRIVIAL EXAMPLE

- What's the fastest way to perform an associative reduction across an array on a GPU?
 - Take an input array
 - Block it based on the number of threads (8 or so per core, usually, up to 24 cores)
 - Iterate to produce a sum in each block
 - Reduce across threads
 - Vectorize (this bit may be a different kernel dispatch given current models)

```
Current models ease programming by viewing the vector as a set of scalars
ALUs, apparently though not really independent, with varying degree of
hardware assistance (and hence overhead):
float sum( 0 )
for( i = n/64 to (n + b)/64; i += 64)
sum += input[i]
float scalarSum = waveReduceViaLocalMemory(sum)
```

THEY DON'T SEEM SO DIFFERENT!

- More blocks of data
 - More cores
 - More threads



- Wider threads
 - 64 on high end AMD GPUs
 - 4/8 on current CPUs
- Hard to develop efficiently for wide threads
- Lots of state, makes context switching and stacks problematic

```
float4 sum( 0, 0, 0, 0 )
for( i = n/4 to (n + b)/4 )
sum += input[i]
float scalarSum = sum.x + sum.y + sum.z + sum.w
```

```
float64 sum( 0, ..., 0 )
for( i = n/64 to (n + b)/64 )
    sum += input[i]
float scalarSum = waveReduce(sum)
```



THAT WAS TRIVIAL... MORE GENERALLY, WHAT WORKS WELL?

On GPU cores:

- We need a lot of data parallelism
- Algorithms that can be mapped to multiple cores and multiple threads per core
- Approaches that map efficiently to wide SIMD units
- So a nice simple functional "map" operation is great!



AMD

THAT WAS TRIVIAL... MORE GENERALLY, WHAT WORKS WELL?

On CPU cores:

- Some data parallelism for multiple cores
- Narrow SIMD units simplify the problem: pixels work fine rather than data-parallel pixel clusters
 - Does AVX change this?
- High clock rates and caches make serial execution efficient
- So in addition to the simple map (which boils down to a for loop on the CPU) we can do complex task graphs



SO TO SUMMARIZE THAT



CUE APU

Tight integration of narrow and wide vector kernels

Combination of high and low degrees of threading

- Fast turnaround
 - Negligible kernel launch time
 - Communication between kernels
 - Shared buffers

Data Data CPU kernel GPU kernel

- For example:
 - Generating a tree structure on the CPU cores, processing the scene on the GPU cores
 - Mixed scale particle simulations (see a later talk)



SO TO SUMMARIZE THAT



QUESTIONS





Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

AMD

© 2011 Advanced Micro Devices, Inc.