AMD

Fusion 11

DEVELOPER SUMMIT

AMD
Fusion11
DEVELOPER SUMMIT

THE FUTURE OF THE APU – BRAIDED PARALLELISM

Session 2901

Benedict R. Gaster
AMD
Programming Models Architect

Lee Howes
AMD
MTS Fusion System Software

AMD Fusion 11 DEVELOPER SUMMIT

PROGRAMMING MODELS
A Track Introduction

Benedict Gaster
AMD
Programming Models Architect

PROGRAMMING MODELS TRACK IS HERE

- GPU architectures are available as we speak
- x86 APU architectures are available as we speak
- Next generation GPU architectures will be covered at this summit!
- Next generation x86 APU architectures will be covered at this summit!
- For those people in the audience who want to develop for such devices:
  - How should we view these architectures?
  - What programming models exist or can we expect?
  - What are the new tricks and tips needed to program these architectures?
- The answers to these questions and many more is what this track is all about

PROGRAMMING MODELS TRACK | Tuesday and Wednesday

	Time	Tuesday	Wednesday	
Sessions	9:45 - 10:30	The Future of the APU – Braided Parallelism	Developing Scalable Applications with Microsoft's C++ Concurrency Runtime	
Sessions	10:45 - 11:30	Cilk Plus: Multi-core Extensions for C and C++	The Future of Parallel and Asynchronous Programming with the .NET Framework	
Lunch & Keynote	12:45 - 1:30			
Sessions	2:00 - 2:45	Diderot: A Parallel Domain-Specific Language for Image Analysis and Visualization // Pixel Bender // Domain Specific Tools to Expand the Code Synthesis Design Space	Heterogeneous Computing with Multi-core Processors, GPUs and FPGAs // Braided Parallelism for Heterogeneous Systems // Making OpenCL™ Simple with Haskell	Blazing-fast code using GPUs and more, with Microsoft Visual C++
Sessions	3:00 - 3:45			N/A
Sessions	4:00 - 5:00	AMD Graphics Core Next	Advanced Graphics Functionality on Windows Using DirectX	
Sessions	5:15 - 6:00	Automatic Intra-Application Load Balancing for Heterogeneous Systems	Towards High-Productivity on Heterogeneous Systems	

PROGRAMMING MODELS TRACK | Thursday

	Time	Thursday
Sessions	8:30 - 9:15	Real-Time Concurrent Linked List Construction on the GPU
Sessions	9:30 - 10:15	Physics Simulation on Fusion Architectures

THE FUSION APU ARCHITECTURE

A Programmer's Perspective

Lee Howes
AMD
MTS Fusion System Software

FUSION IS HERE

- x86 + Radeon APU architectures are available as we speak
- For those people in the audience who want to develop for such devices:
  - How should we view these architectures?
  - What programming models exist or can we expect?

TAKING A REALISTIC LOOK AT THE APU

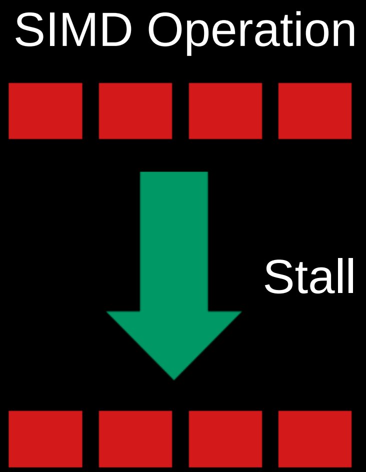
- GPUs are not magic
  - We've often heard about 100x performance improvements
  - These are usually the result of poor CPU code
- The APU offers a balance
  - GPU cores optimized for arithmetic workloads and latency hiding
  - CPU cores to deal with the branchy code for which branch prediction and out 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

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
- 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

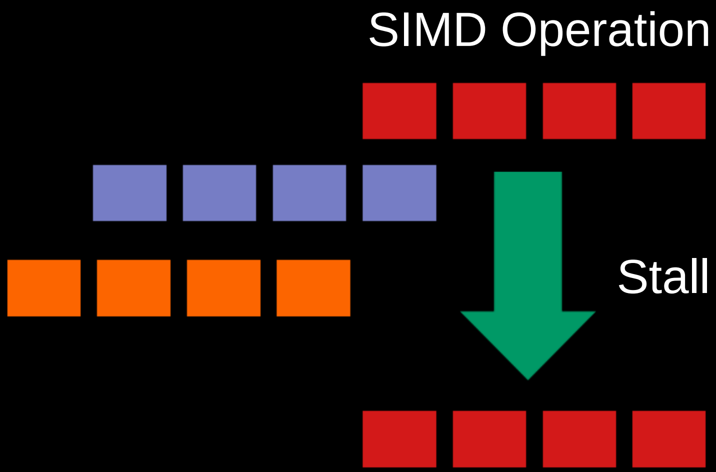
HIDING OR REDUCING LATENCY

- Any instruction issued might stall waiting on memory or at least in the computation pipeline



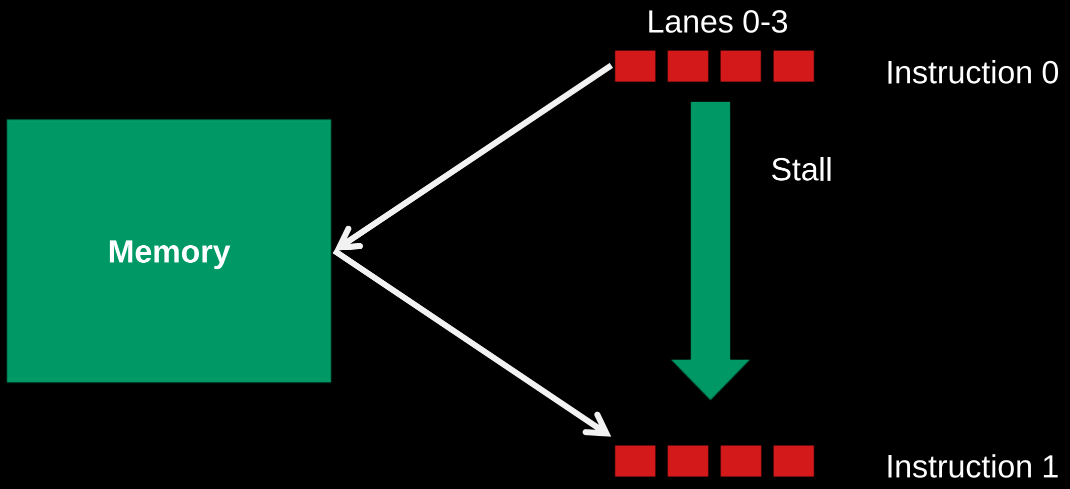
HIDING OR REDUCING LATENCY

- Any instruction issued might stall waiting on memory or at least in the computation pipeline
- Out of order logic attempts to issue as many instructions as possible as tightly as possible, filling small stalls with other instructions from the same thread



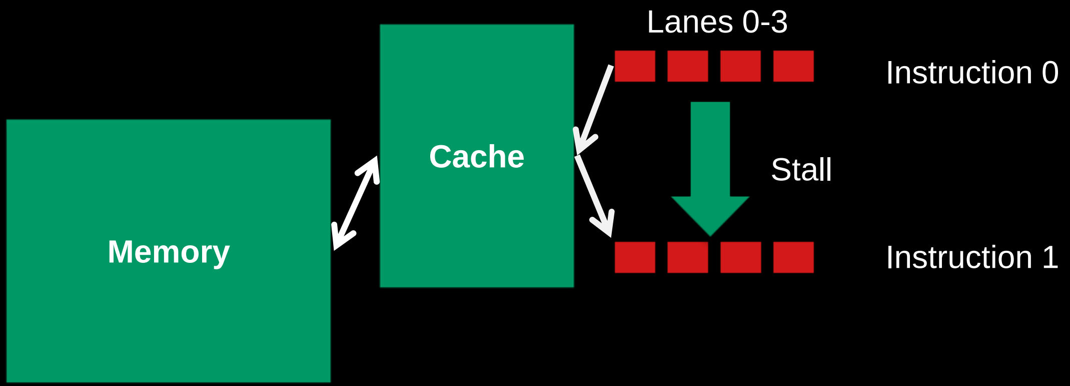
REDUCING MEMORY LATENCY ON THE CPU

- Larger memory stalls are harder to cover
- Memory can be some way from the ALU
  - Many cycles to access



REDUCING MEMORY LATENCY ON THE CPU

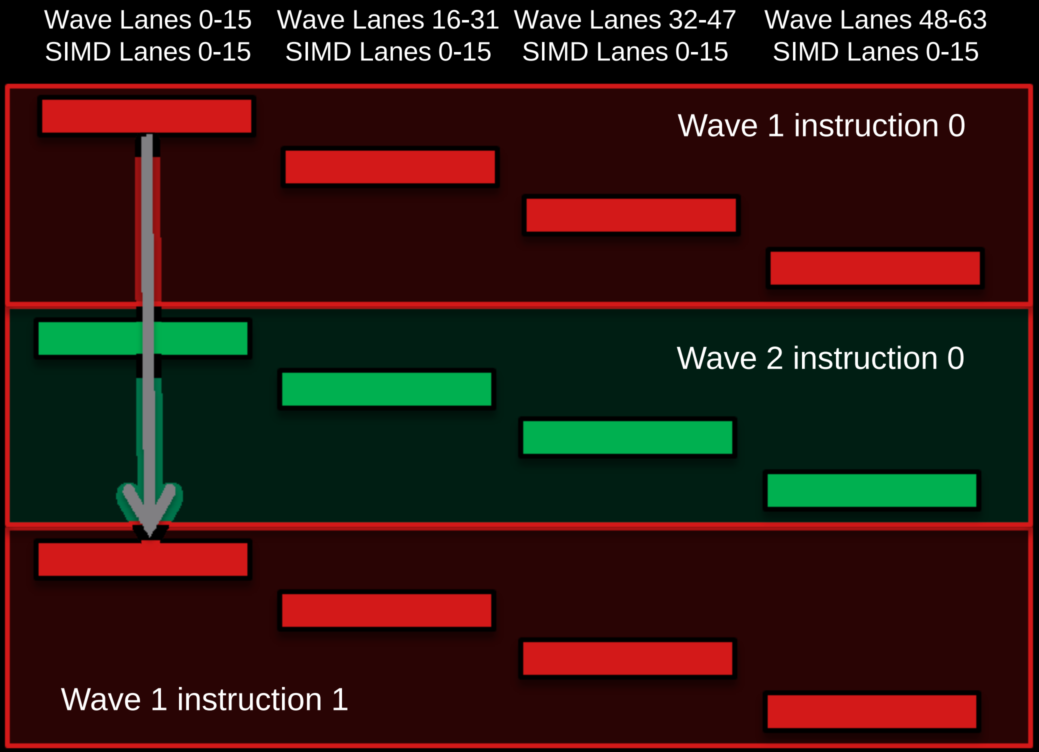
- Larger memory stalls are harder to cover
- Memory can be some way from the ALU
  - Many cycles to access
- CPU solution is to introduce an intermediate cache memory
  - Minimizes the latency of **most** accesses
  - Reduces stalls



HIDING LATENCY ON THE GPU

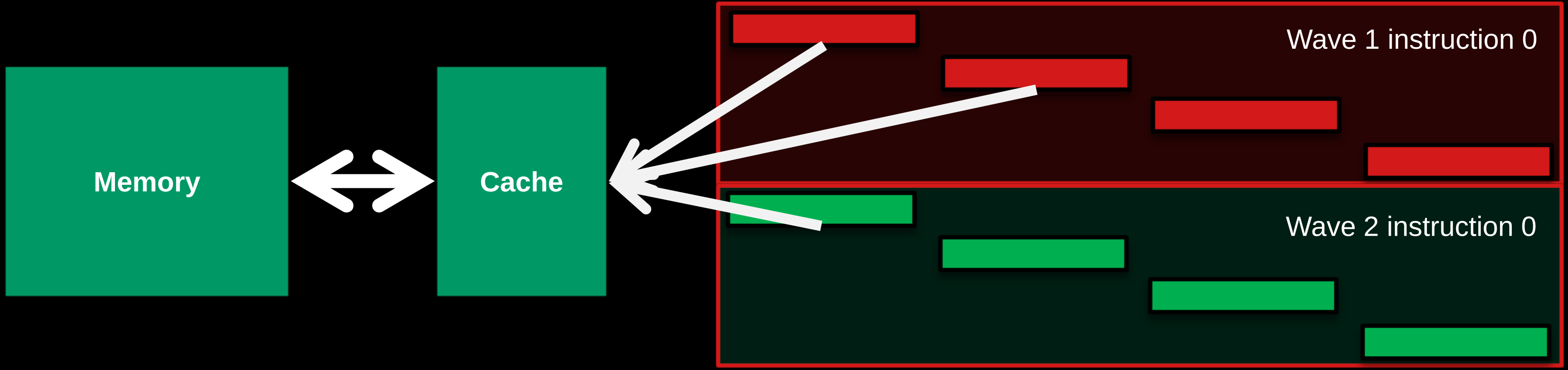
- AMD's GPU designs do two things:
  - Issue an instruction over multiple cycles
  - Interleave instructions from multiple threads
- Multi-cycle a large vector on a smaller vector unit
  - Reduces instruction decode overhead
  - Improves throughput
- Multiple threads fill gaps in instruction stream
- Single thread latency can actually INCREASE

AMD Radeon HD6970
64-wide wavefronts interleaved on 16-wide vector unit



GPU CACHES

- GPUs also have caches
  - The goal is generally to improve spatial locality rather than temporal



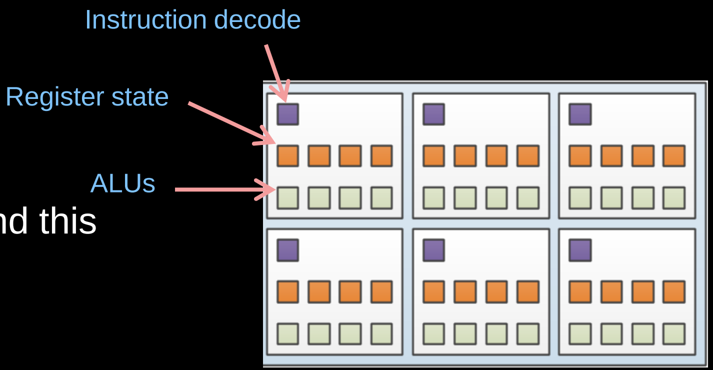
  - Different parts of a wide SIMD thread and different threads may require similar locations and share through the cache

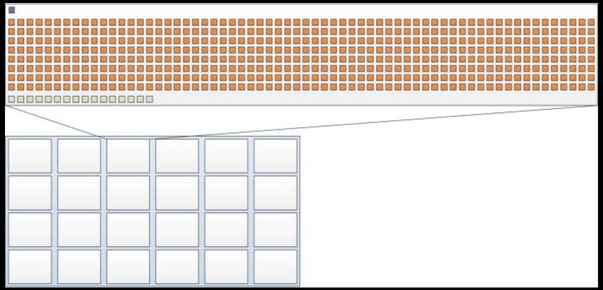
COSTS

- 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
- These two approaches suit different algorithm designs
- We cannot, unfortunately, have both in a single core

THE TRADEOFFS IN PICTURES

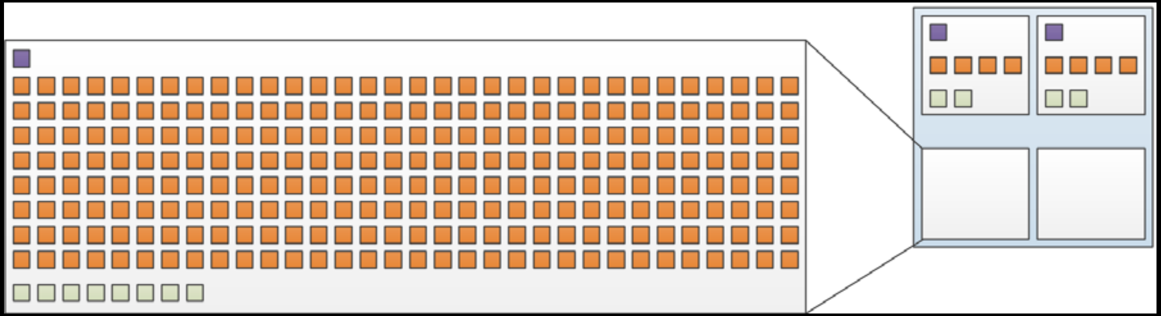
- AMD Phenom™ II X6:
  - 6 cores, 4-way SIMD (ALUs)
  - A single set of registers
  - Registers are backed out to memory on a thread switch, and this is performed by the OS
- AMD Radeon™ HD6970:
  - 24 cores, 16-way SIMD (plus VLIW issue, but the Phenom processor does multi-issue too), 64-wide SIMD state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





AREA EFFICIENCY TRADEOFFS

- 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



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

```
float sum( 0 )
for( i = n to n+b )
    sum += input[i]
```

```
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
```

```
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

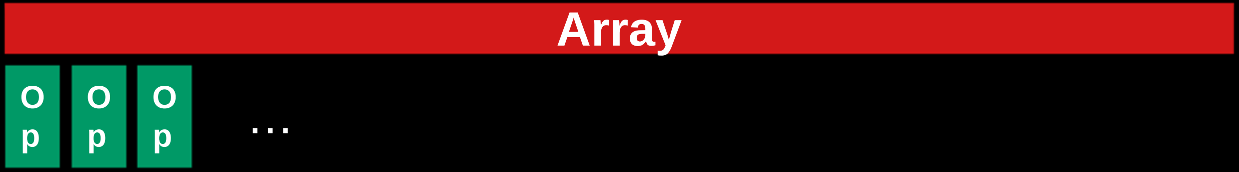
- Wider threads
  - 64 on high end AMD GPUs
  - 4/8 on current CPUs
- Hard to develop efficiently for wide threads
- Lots of state, makes context switching and stacks problematic

```
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!

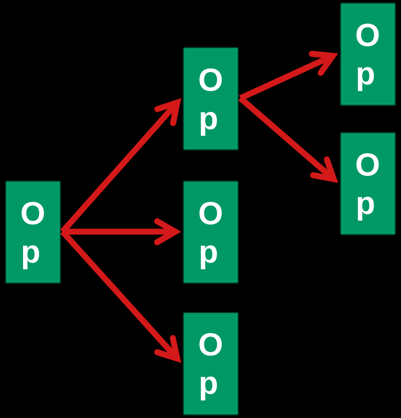


Array.map(Op)

  - This is basically the OpenCL™ model

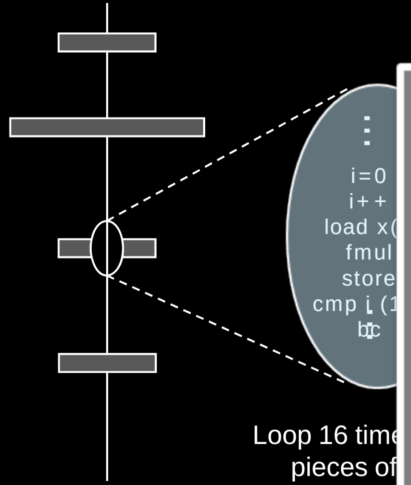
WHAT WORKS WELL?

- 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



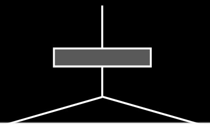
SO TO SUMMARIZE THAT

Fine-grain data parallel Code



Maps very well to integrated SIMD dataflow (ie: SSE)

Nested data parallel Code



Discrete GPU configurations suffer from communication latency.

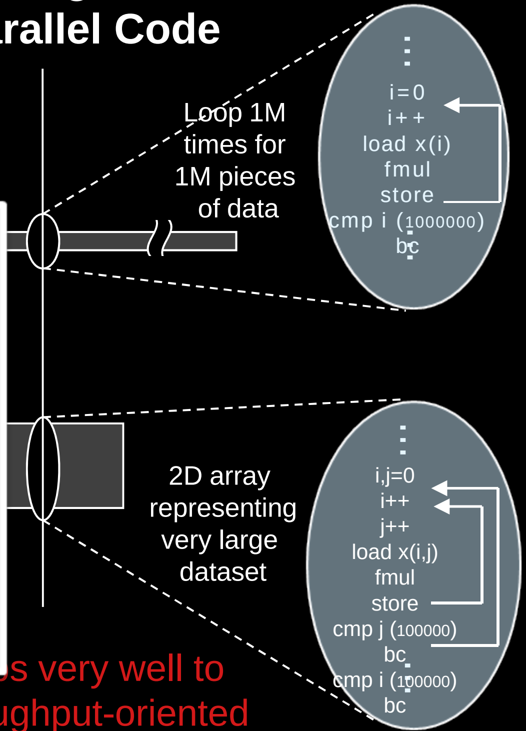
Nested data parallel/braided parallel code benefits from close coupling.

Discrete GPUs don’t provide it well.

But each individual core isn’t great at certain types of algorithm…

parallelism. Benefits from closer coupling between CPU & GPU

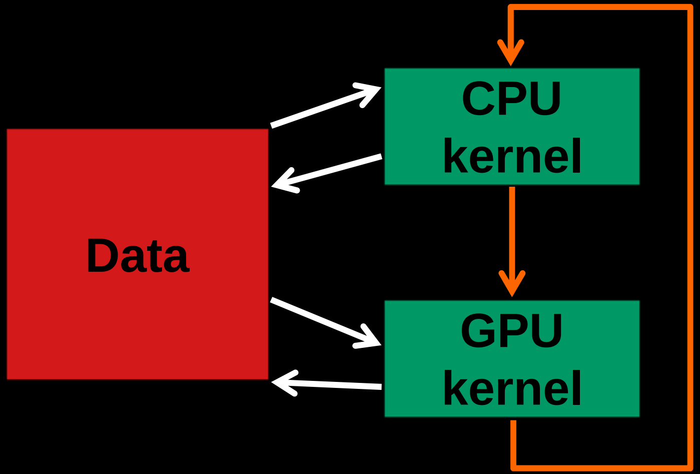
Coarse-grain data parallel Code



ps very well to Throughput-oriented data parallel engines

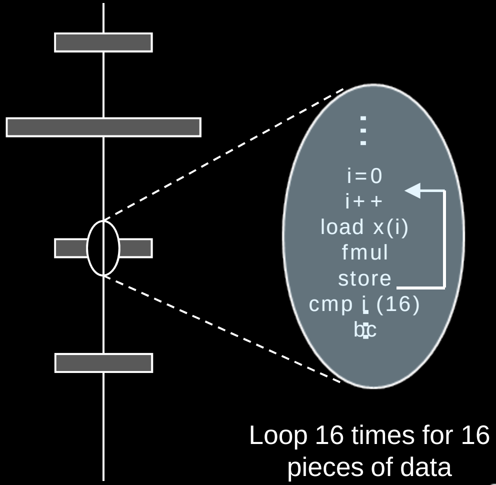
SO WHAT APPLICATIONS BENEFIT?

- Tight integration of narrow and wide vector kernels
- Combination of high and low degrees of threading
- Fast Turnaround
  - Communication between kernels
  - Shared buffers
- 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)



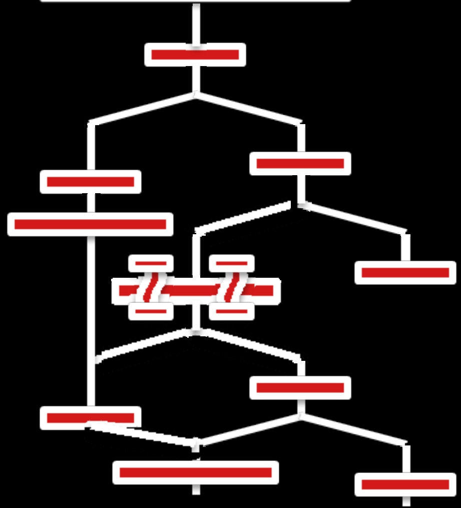
SO TO SUMMARIZE THAT

Fine-grain data parallel Code



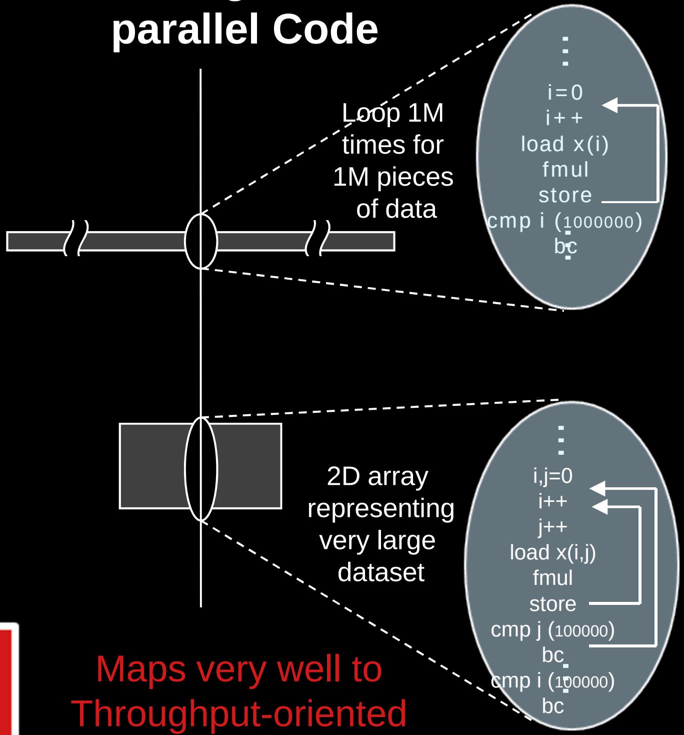
Maps very well to integrated SIMD dataflow (ie: SSE)

Nested data parallel Code



Lots of conditional data parallelism. Benefits from closer coupling between CPU & GPU

Coarse-grain data parallel Code



Maps very well to Throughput-oriented data parallel engines

HOW DO WE USE THESE DEVICES?

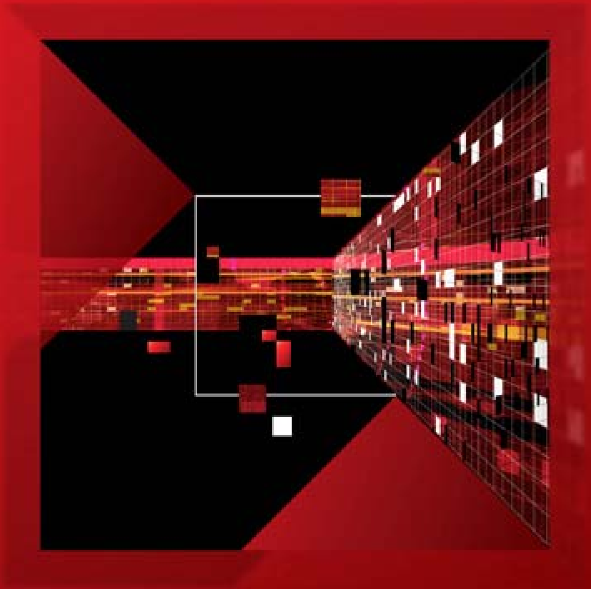
- Heterogeneous programming isn't easy
  - Particularly if you want performance
- To date:
  - CPUs with visible vector ISAs
  - GPUs mostly lane-wise (implicit vector) ISAs
  - Clunky separate programming models with explicit data movement
- How can we target both?
  - With a fair degree of efficiency
  - True shared memory with passable pointers
- Let's talk about the future of programming models…

AMD
Fusion 11
DEVELOPER SUMMIT

PROGRAMMING MODELS
For Heterogeneous Computing

Benedict Gaster
AMD
Programming models architect

TODAY



INDUSTRY STANDARD API STRATEGY

OpenCL™

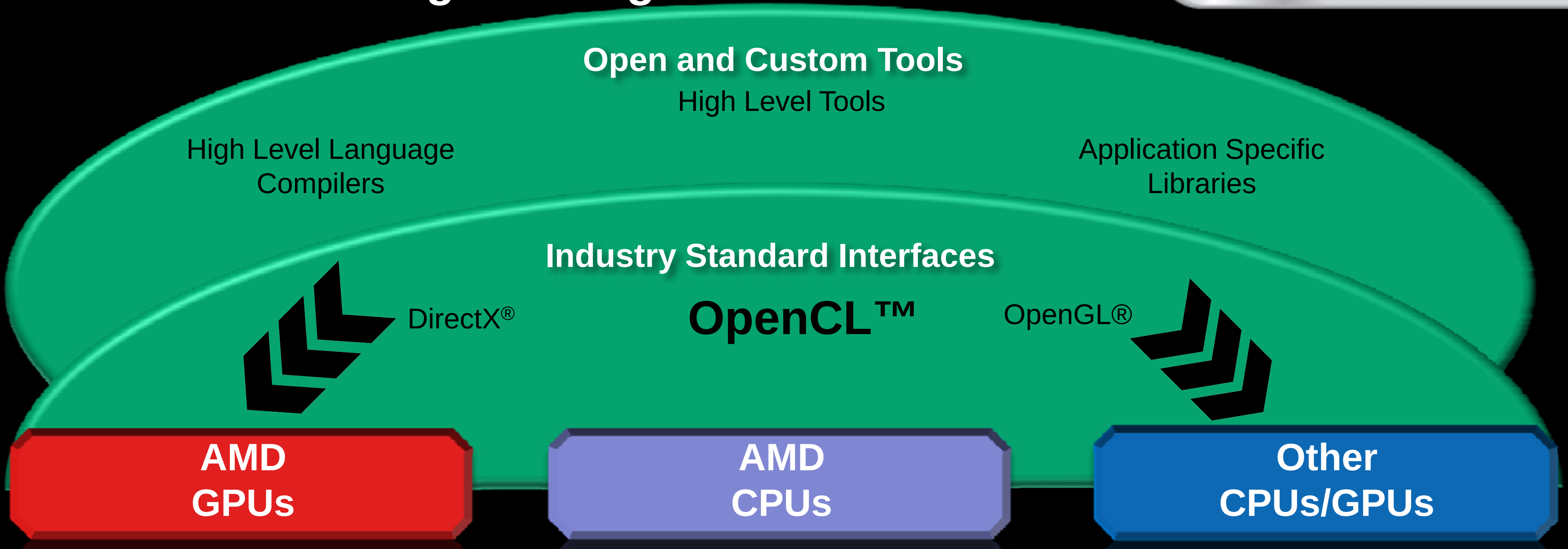
- Open development platform for multi-vendor heterogeneous architectures
- The power of AMD Fusion: Leverages CPUs and GPUs for balanced system approach
- Broad industry support: Created by architects from AMD, Apple, IBM, Intel, NVIDIA, Sony, etc. AMD is the first company to provide a complete OpenCL solution
- Momentum: Enthusiasm from mainstream developers and application software partners

DirectX® 11 DirectCompute

- Microsoft distribution
- Easiest path to add compute capabilities to existing DirectX applications

Moving Past Proprietary Solutions for Ease of Cross-Platform Programming

AMD Accelerated Parallel Processing TECHNOLOGY

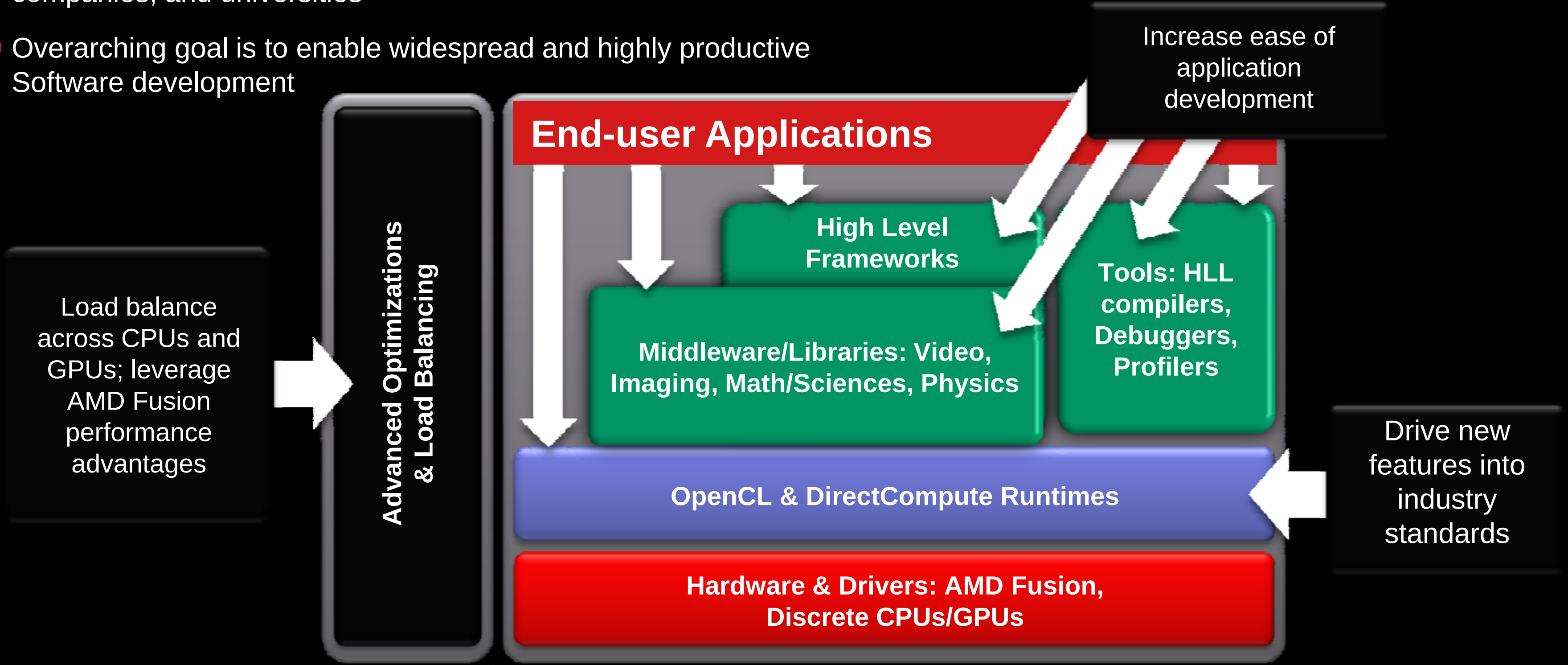


OpenCL -

- Cross-platform development
- Interoperability with OpenGL and DX
- CPU/GPU backends enable balanced platform approach

HETEROGENEOUS COMPUTING | Software Ecosystem

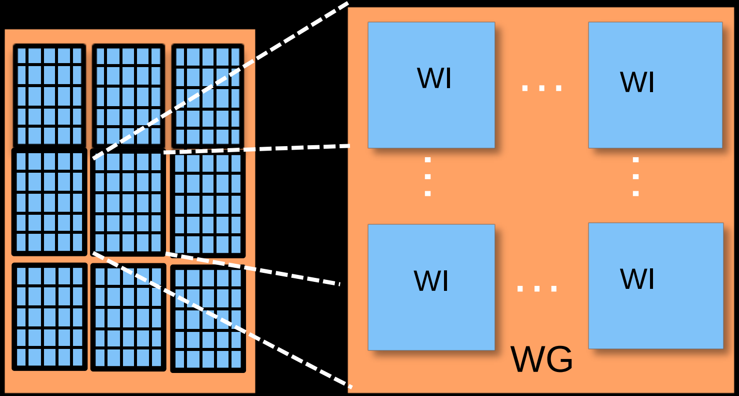
- Encourages contributions from established companies, new companies, and universities
- Overarching goal is to enable widespread and highly productive Software development



Increase ease of application development
End-user Applications
High Level Frameworks
Middleware/Libraries: Video, Imaging, Math/Sciences, Physics
Tools: HLL compilers, Debuggers, Profilers
OpenCL & DirectCompute Runtimes
Hardware & Drivers: AMD Fusion, Discrete CPUs/GPUs
Advanced Optimizations & Load Balancing
Load balance across CPUs and GPUs; leverage AMD Fusion performance advantages
Drive new features into industry standards

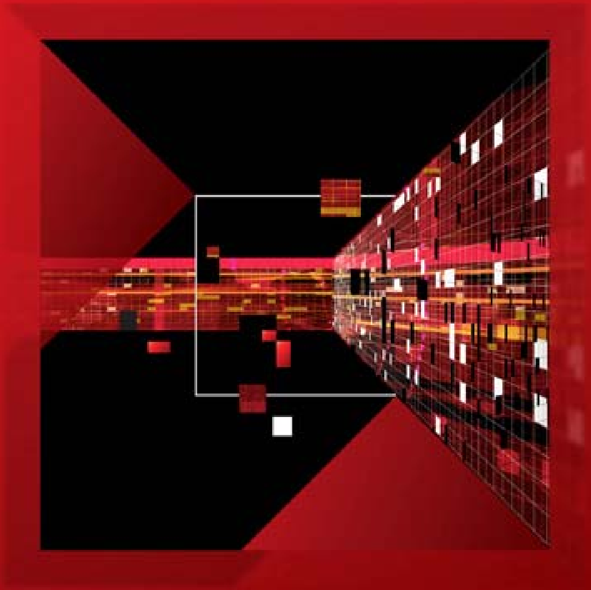
TODAY'S EXECUTION MODEL

- SPMD
  - Same kernel runs on:
    - All compute units
    - All processing elements
  - Purely "data parallel" mode
  - Device model:
    - Device runs a single kernel simultaneously
    - Separation between compute units is relevant for memory model only.



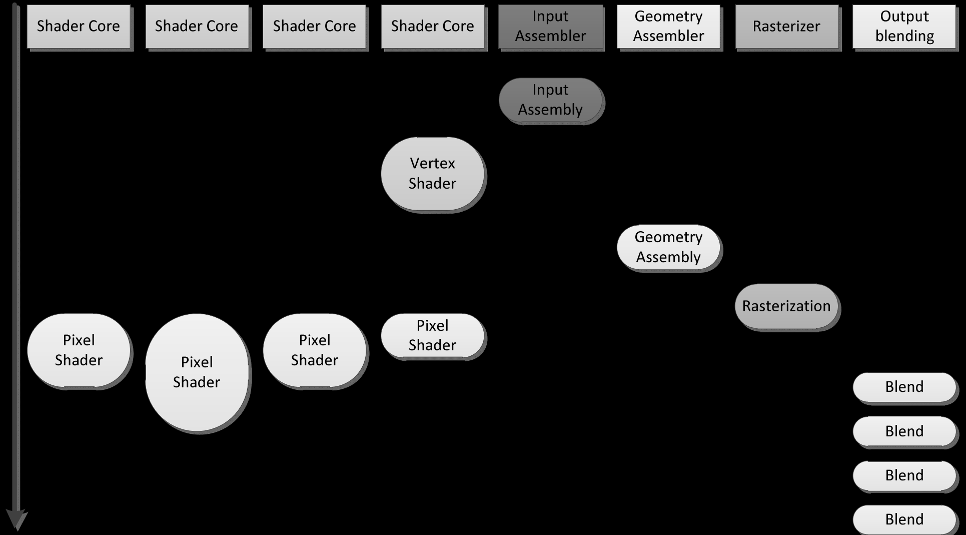
Modern CPUs & GPUs can support more!

TOMORROW



MODERN GPU (& CPU) CAPABILITIES

- Modern GPUs can execute a different instruction stream per core
  - Some even have a few HW threads per core (each runs separated streams)
- This is still a highly parallelized machine!
  - HW thread executes N-wide vector instructions (8-64 wide)
  - Scheduler switches HW threads on the fly to hide memory misses

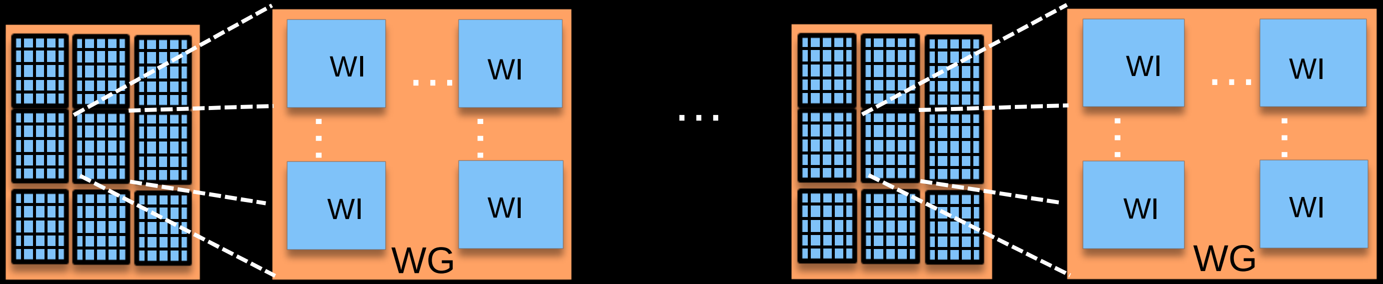


FUSION SYSTEM ARCHITECTURE ROADMAP

Physical Integration	Optimized Platforms	Architectural Integration	System Integration
Integrate CPU & GPU in silicon	AMD Fusion Interconnect	Unified Address Space for CPU and GPU	GPU compute context switch
Unified Memory Controller	GPU Compute C++ support	GPU uses pageable system memory via CPU pointers	GPU graphics pre-emption
Common Manufacturing Technology	Bi-Directional Power Mgmt between CPU and GPU	Fully coherent memory on both CPU and GPU	Quality of Service
		User mode scheduling	Coherent PCIe®

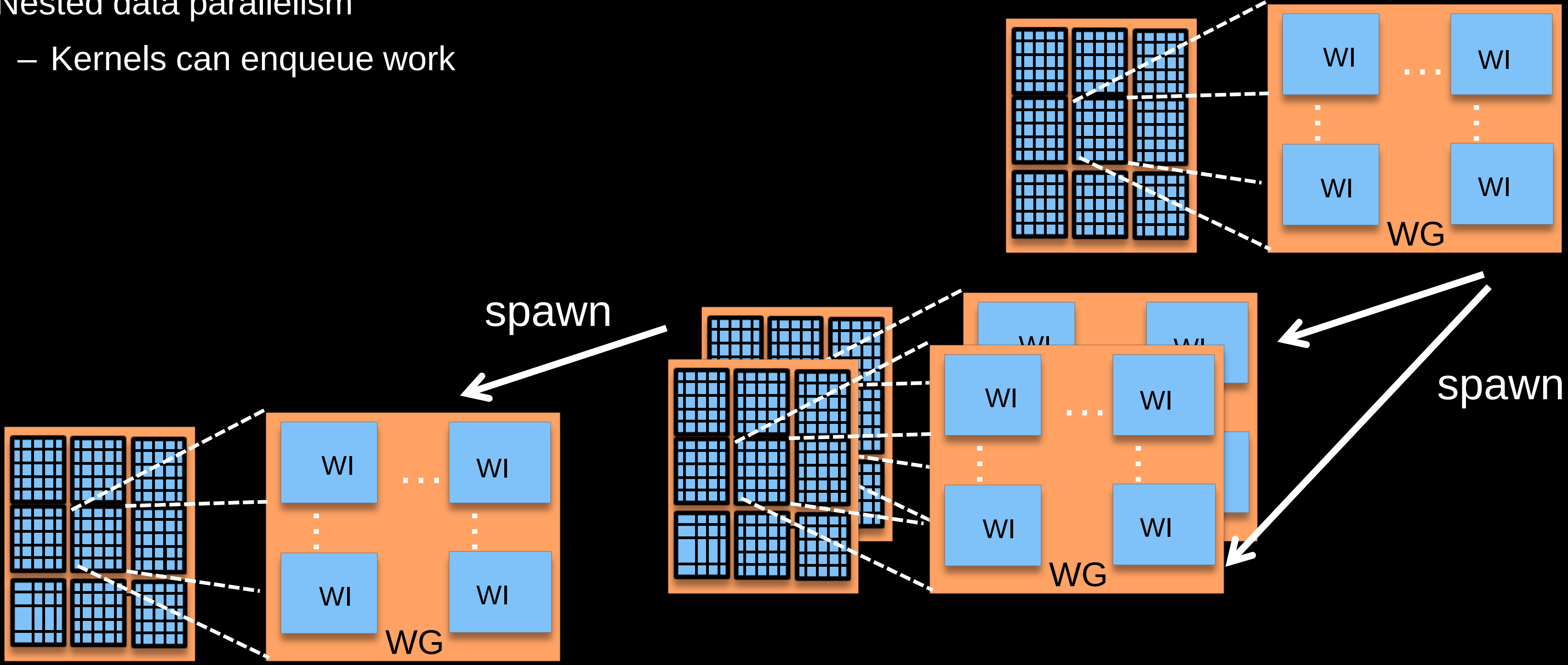
TOMORROW'S EXECUTION MODEL

- MPMD
  - Same kernel runs on:
    - 1 or more compute units
  - Still "data parallel"
  - Device model:
    - Device can runs multiple kernels simultaneously

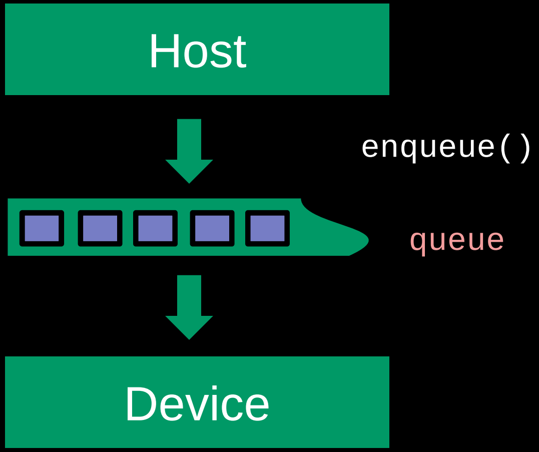


TOMORROW'S EXECUTION MODEL

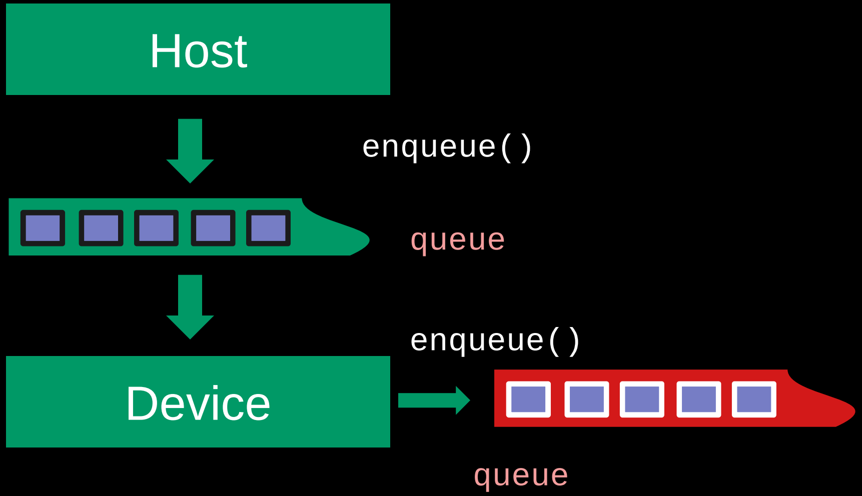
- Nested data parallelism
  - Kernels can enqueue work



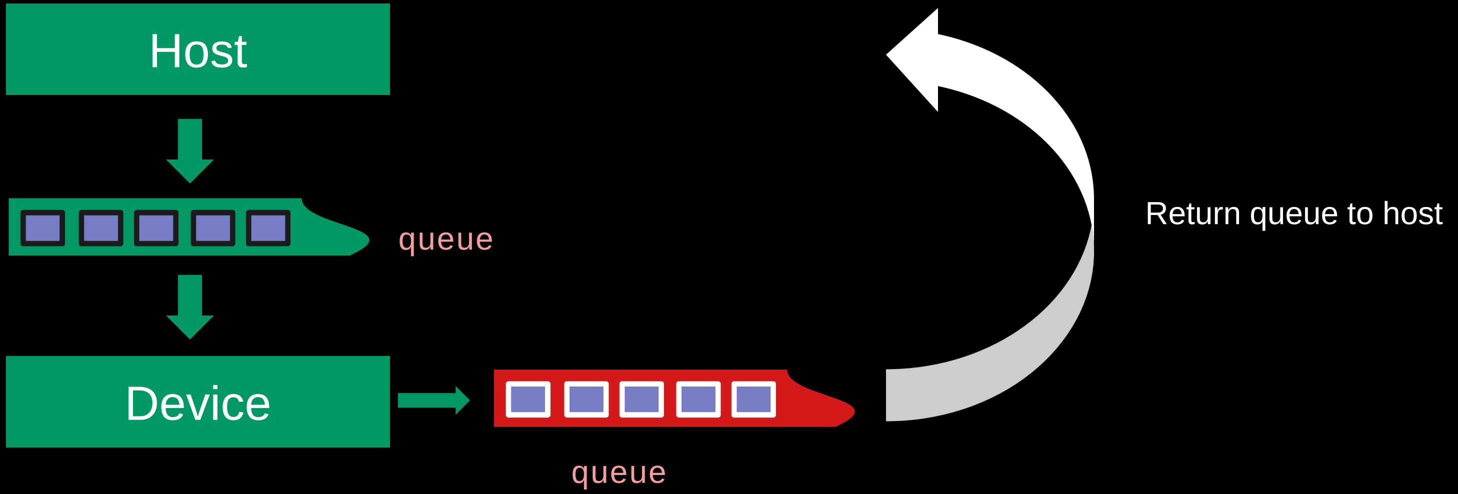
TOMORROW'S EXECUTION MODEL | Visual Example



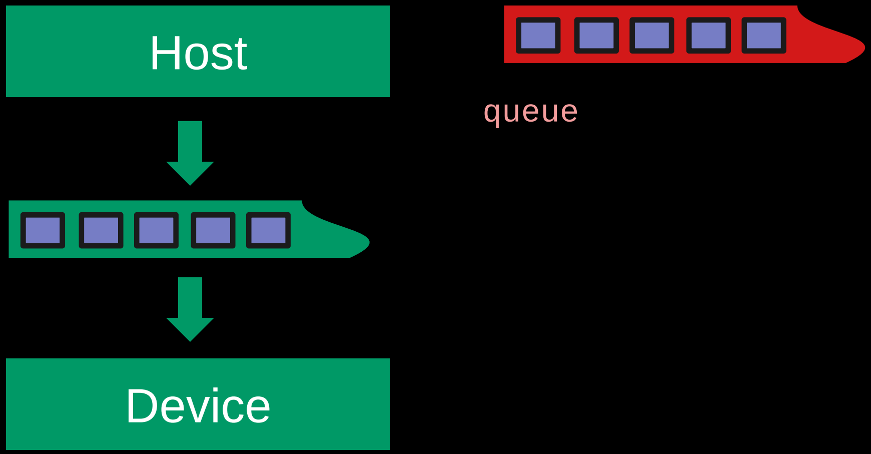
TOMORROW'S EXECUTION MODEL | Visual Example



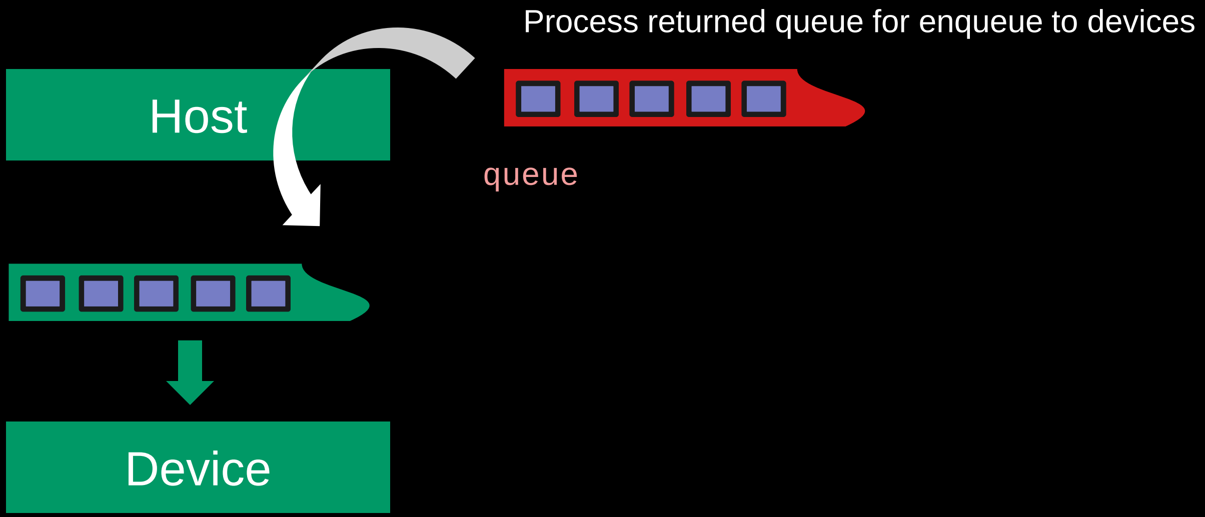
TOMORROW'S EXECUTION MODEL | Visual Example



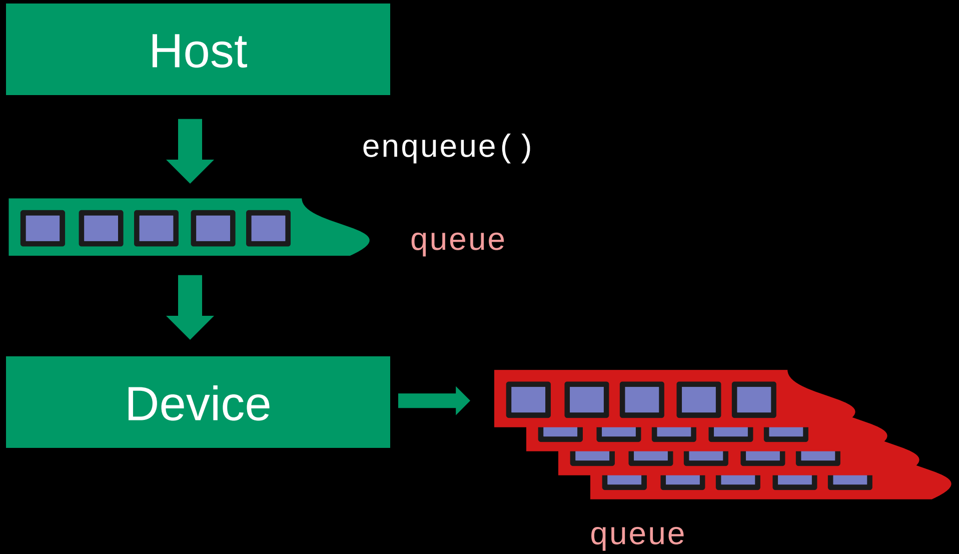
TOMORROW'S EXECUTION MODEL | Visual Example



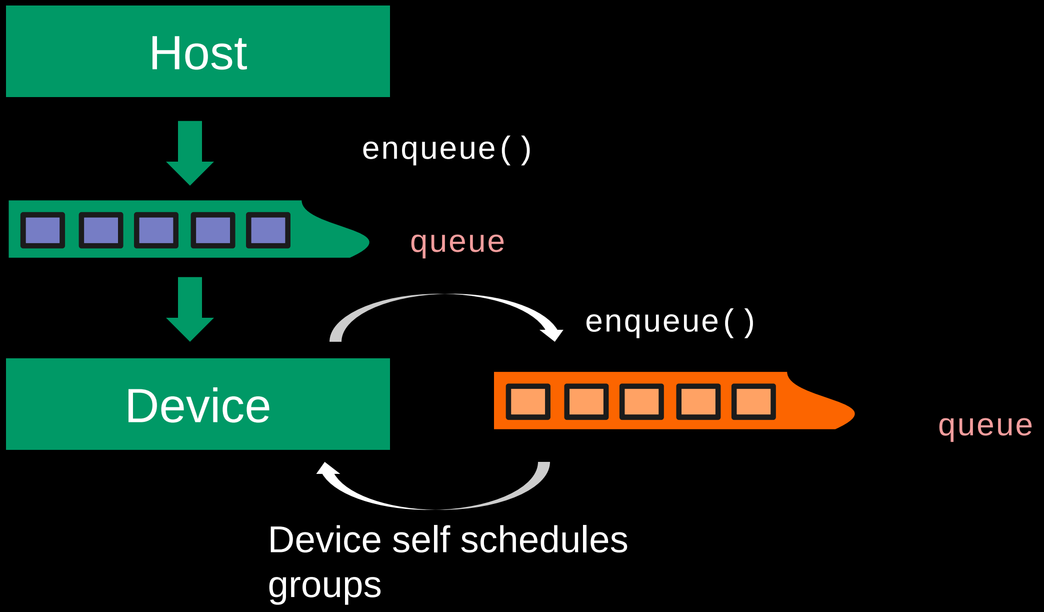
TOMORROW'S EXECUTION MODEL | Visual Example



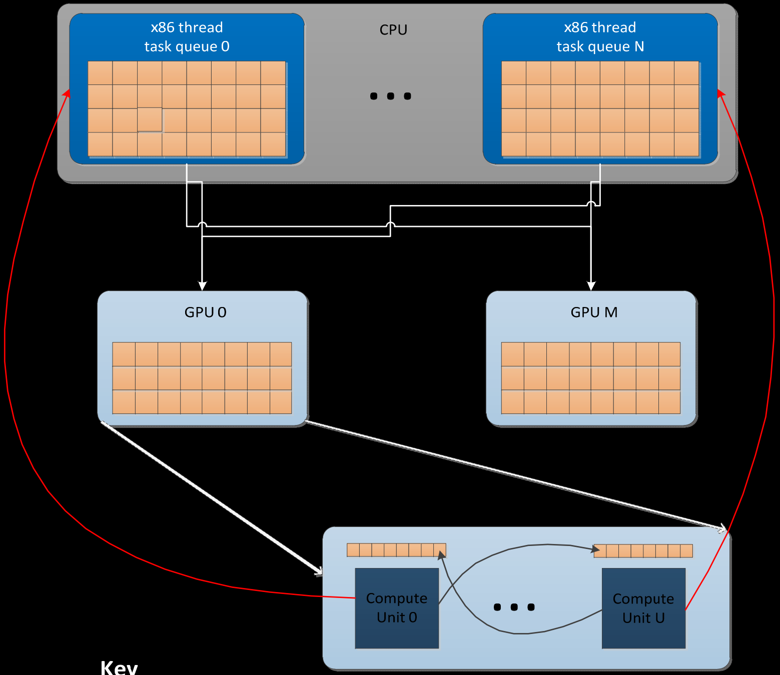
TOMORROW'S EXECUTION MODEL | Visual Example



TOMORROW'S EXECUTION MODEL | Visual Example



TOMORROW'S EXECUTION MODEL



Key

🟧 task of work

⬛ SIMD core

→ Schedule work (e.g. steal or push).

→ Schedule work through the CPU (e.g. spawn new work).This is not supported by today's persistent thread models.

TOMORROW'S HIGH LEVEL BRAIDED PARALLEL MODEL

- High-level programming model
  - Braided parallelism (task + data-parallel)
  - Hardware independent access to the memory model
  - Based on C++0x (Lambda, auto, decltype, and rvalue references)
- Will support all C++ capabilities
  - GPU kernels/functions can be written in C++
  - Not an implication that everything should be used on all devices

TOMORROW'S HIGH LEVEL BRAIDED PARALLEL MODEL

- Leverage host language (C++) concepts and capabilities
  - Separately authored functions, types and files
  - Libraries reuse (link-time code-gen required)
  - Binaries contain compiled device-independent IL and x86 code
- Dynamic online compilation still an option

TOMORROW'S HIGH LEVEL BRAIDED PARALLEL MODEL

- Multi dimensional: Range and Index, Partition, and IndexRange
- Data Parallel Algorithms
  - Examples: parallelFor, parallelForWithReduce
- Task Parallel Algorithms
  - Examples: Task<T>, Future<T>
- Task + Data Parallelism = Braided Parallelism
- Low-level concepts are still available for performance
  - Work-groups, Work-items, shared memory, shared memory barriers
  - A layered programming model with ease-of-use and performance tradeoffs

HELLO WORLD OF THE GPGPU WORLD (I.E. VECTOR ADD)

```
int main(void)
{
    float A[SIZE]; float B[SIZE]; float C[SIZE];

    srand ( time(NULL) );
    for (int i = 0; i < SIZE; i++) {
         A[i] = rand(); B[i] = rand();
    }

    parallelFor(Range<1>(size),  [A,B,C] (Index<1> index) [[device]]
{
          C[index.getX()] =  A[index.getX()] + B[index.getX()];
    });

    for (float i: C) {
       cout << i << endl;
    }
}
```

AMD Fusion11 DEVELOPER SUMMIT

TASKS AND FUTURES

```
int parallelFib(int n)
{
   if (n < 2) {
      return n;
   }

   Future<int> x([n] ()
[[device]] {
      return parallelFib(n-1)
   });

   int y = parallelFib(n-2);

   return x.get() + y;
}
```

```
#include <opp.hpp>
#include <iostream>
int main(void)
{
   Future<vector<Int>> fibs(
      Range<1>(100),
      [] (Index<1> index)
[[device]] {
      return
parallelFib(index.getX());
    });
    for(auto f = fibs.begin();
      f != fibs.end();
      f++) {
       cout << f << endl;
    }
}
```

TASKS AND REDUCTION

```
float myArray[…];

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

   return float;
});

float sum = task.enqueueWithReduce(Range<1>(1920), sums, plus<float>());
```

QUESTIONS

Disclaimer & Attribution