

MAKING GPGPU EASIER

Software and hardware improvements in GPU computing

Lee Howes

AMD Heterogeneous System Software



WE HEAR A LOT OF

- AMD's GPUs are hard to program
- OpenCL requires a lot of boiler plate code not needed by other compute models:
 - CUDA, clearly
 - pragma models... but do we really want to take that route?

SO IN THIS TALK

- Will introduce:
 - AMD's latest generation GPU that makes GPGPU programming a whole lot simpler and faster
 - The HSA architecture that extends this simpler model across the platform
 - OpenCL C++ a simpler programming model for GPGPU programming
 - And some thoughts about how we're trying to improve this in the future



A NEW ERA OF PROCESSOR PERFORMANCE

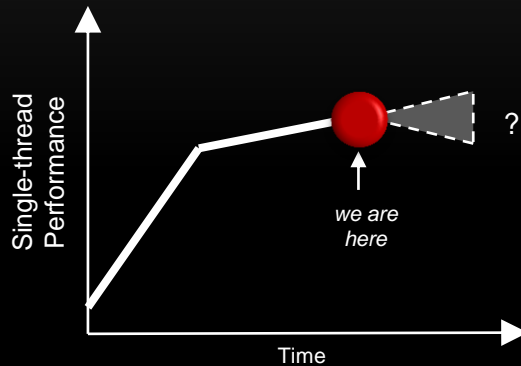
Single-Core Era

Enabled by:

- ✓ Moore's Law
- ✓ Voltage Scaling

Constrained by:

- ✗ Power
- ✗ Complexity



A NEW ERA OF PROCESSOR PERFORMANCE

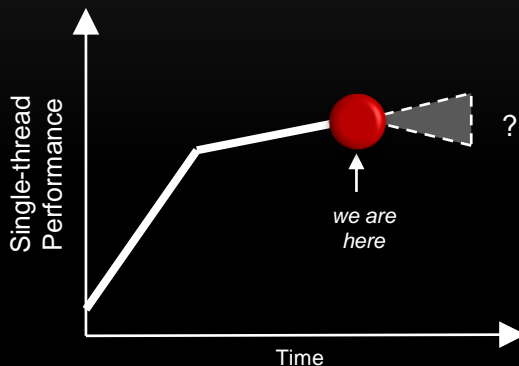
Single-Core Era

Enabled by:

- ✓ Moore's Law
- ✓ Voltage Scaling

Constrained by:

- ✗ Power
- ✗ Complexity



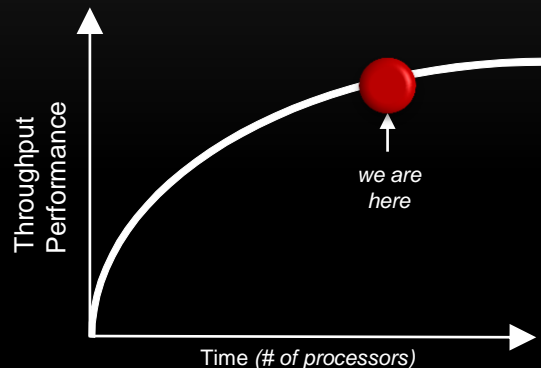
Multi-Core Era

Enabled by:

- ✓ Moore's Law
- ✓ SMP architecture

Constrained by:

- ✗ Power
- ✗ Parallel SW
- ✗ Scalability



A NEW ERA OF PROCESSOR PERFORMANCE

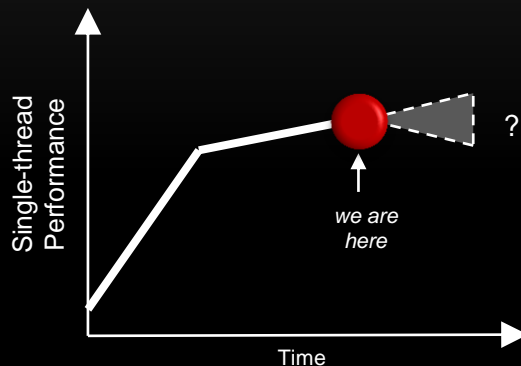
Single-Core Era

Enabled by:

- ✓ Moore's Law
- ✓ Voltage Scaling

Constrained by:

- ✗ Power
- ✗ Complexity



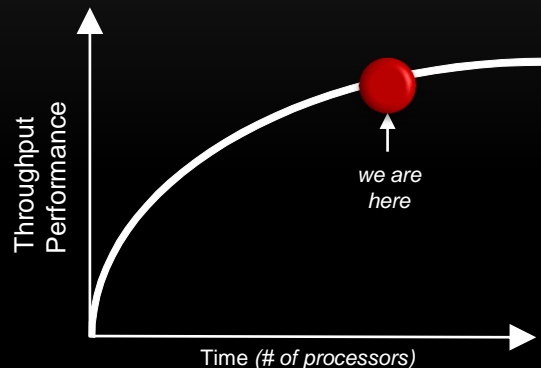
Multi-Core Era

Enabled by:

- ✓ Moore's Law
- ✓ SMP architecture

Constrained by:

- ✗ Power
- ✗ Parallel SW
- ✗ Scalability



Heterogeneous Systems Era

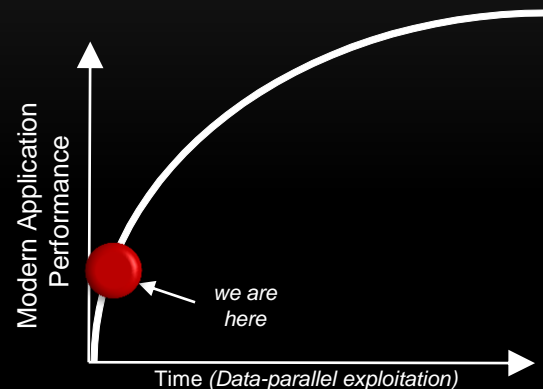
Enabled by:

- ✓ Abundant data parallelism
- ✓ Power efficient GPUs

Temporarily

Constrained by:

- ✗ Programming models
- ✗ Comm.overhead



A NEW ERA OF PROCESSOR PERFORMANCE

Single-Core Era

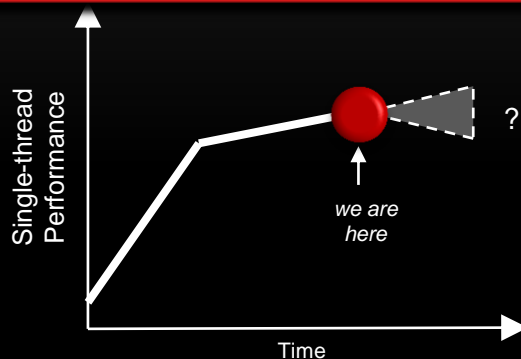
Enabled by:

- ✓ Moore's Law
- ✓ Voltage Scaling

Constrained by:

- ✗ Power
- ✗ Complexity

Assembly → C/C++ → Java ...



Multi-Core Era

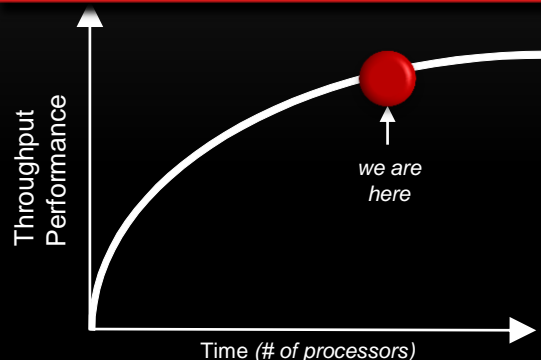
Enabled by:

- ✓ Moore's Law
- ✓ SMP architecture

Constrained by:

- ✗ Power
- ✗ Parallel SW
- ✗ Scalability

pthread → OpenMP / TBB ...



Heterogeneous Systems Era

Enabled by:

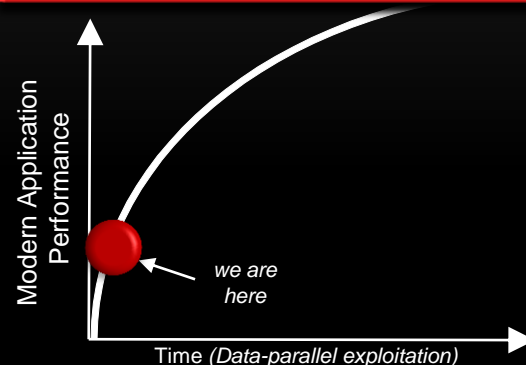
- ✓ Abundant data parallelism
- ✓ Power efficient GPUs

Temporarily

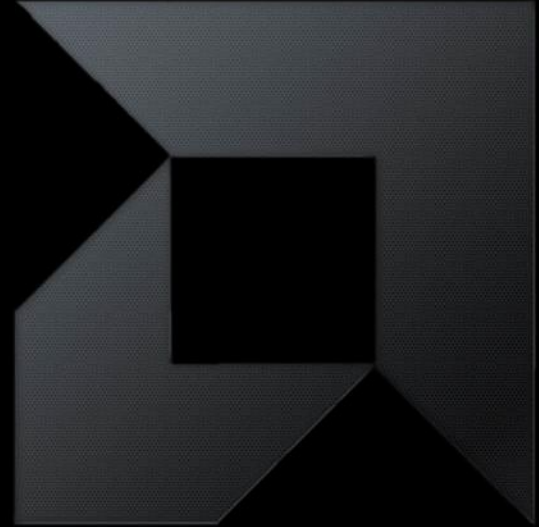
Constrained by:

- ✗ Programming models
- ✗ Comm.overhead

Shader → CUDA → OpenCL → !!!



GPUS AND CPUS
DESIGNING TO SOLVE A GIVEN PROBLEM



TAKING A REALISTIC LOOK AT GPU COMPUTING

- GPUs are not magic
 - We've often heard about 100x performance improvements
 - These are usually the result of poor CPU code

TAKING A REALISTIC LOOK AT GPU COMPUTING

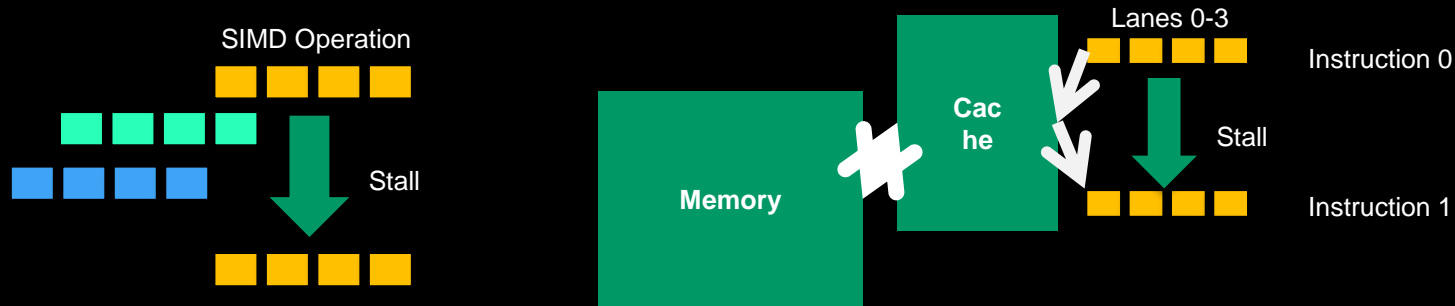
- GPUs are not magic
 - We've often heard about 100x performance improvements
 - These are usually the result of poor CPU code
- Usually?
 - Hmm...
- Some people talk about thousands of threads and cores, too...
 - Marketing and reality are rarely the same

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

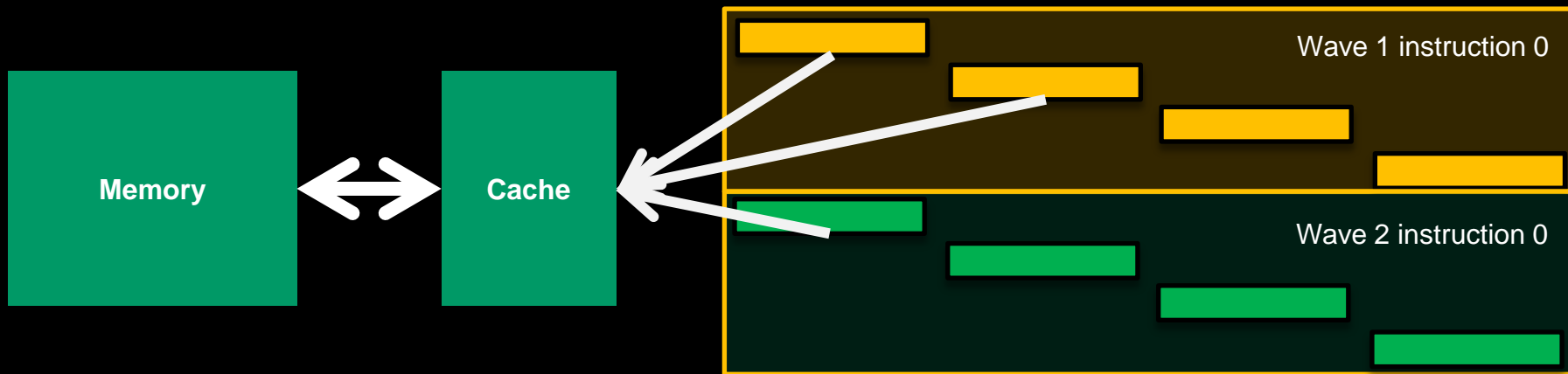
REDUCING LATENCY ON THE CPU

- Out of order execution to cover instruction latency (and increase parallelism)
- Caches to reduce time to memory



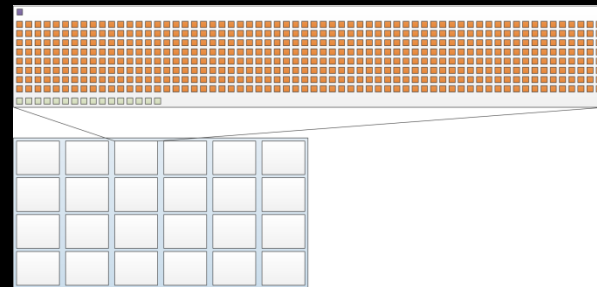
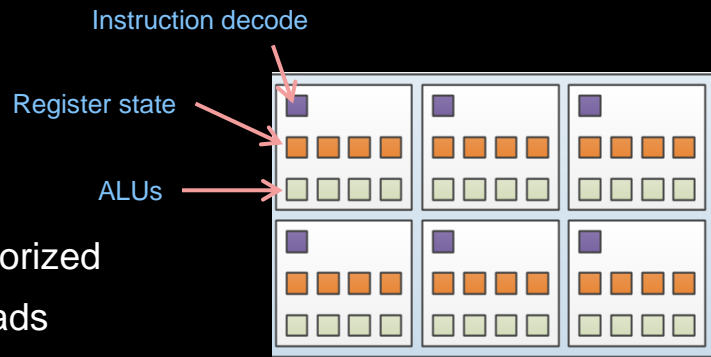
MAINTAINING THROUGHPUT ON THE GPU

- GPUs also have caches
 - The goal is generally to improve spatial locality rather than temporal
- Multi-cycle a wide vector thread
 - Reduce instruction decode overhead, cover instruction latency
- Run multiple threads concurrently, interleaving to cover latency

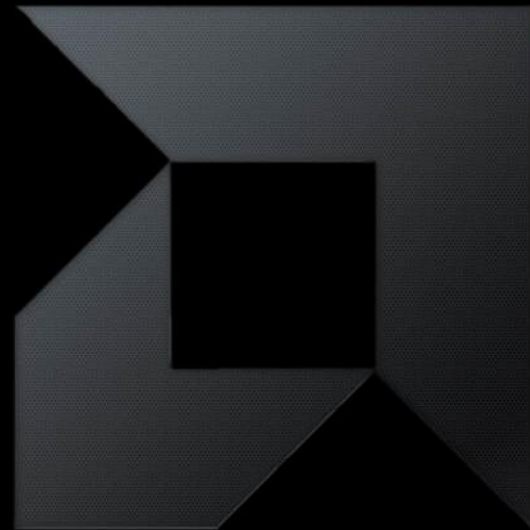


COSTS

- The CPU approach:
 - Requires large caches
 - Dedicates transistors to 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
 - The entire point of the design is hand vectorization
- These two approaches suit different algorithm designs



***THINKING ABOUT
PROGRAMMING***



THE CPU PROGRAMMATICALLY: A TRIVIAL EXAMPLE

- What's the fastest way to perform an associative reduction across an array on a CPU?

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



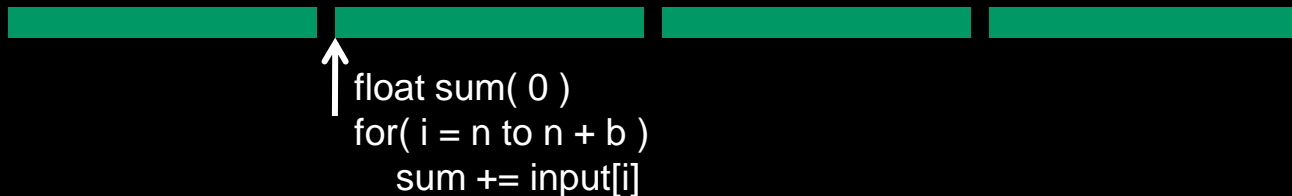
THE CPU PROGRAMMATICALLY: A TRIVIAL EXAMPLE

- 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)



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



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



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

```
float reductionValue( 0 )  
for( t in threadCount )  
    reductionValue += t.sum
```

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



```
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?

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



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)



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



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

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



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

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 involve a different kernel dispatch given current models)



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

```
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 involve 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



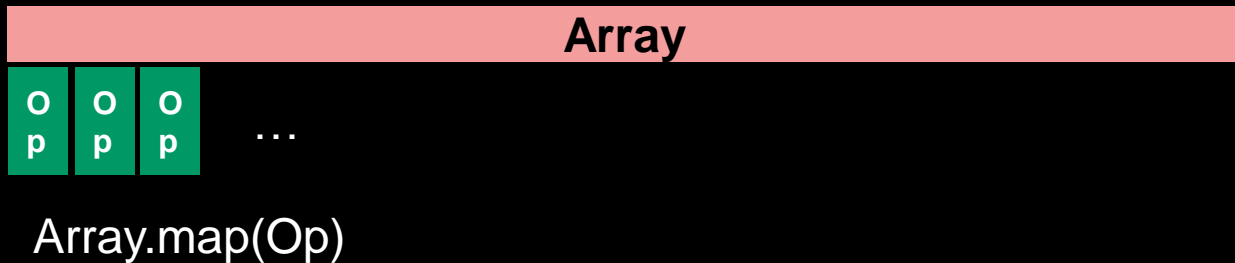
- Wider threads
 - 64 on high end AMD GPUs
 - 4/8 on current CPUs
- Hard to develop efficiently for wide threads
- Lots of state, makes context switching and stacks problematic

```
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?

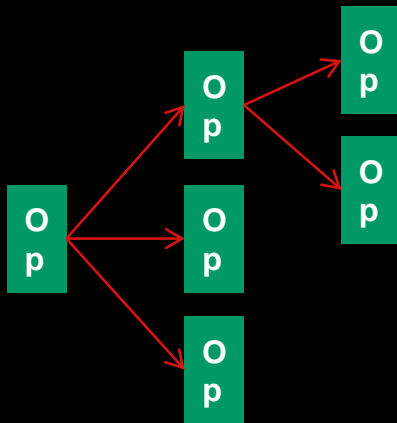
- On GPU cores:
 - We need a lot of data parallelism
 - Algorithms that can be mapped to multiple cores and multiple threads per core
 - Approaches that map efficiently to wide SIMD units
 - So a nice simple functional “map” operation is great!



- This is basically the OpenCL™ model

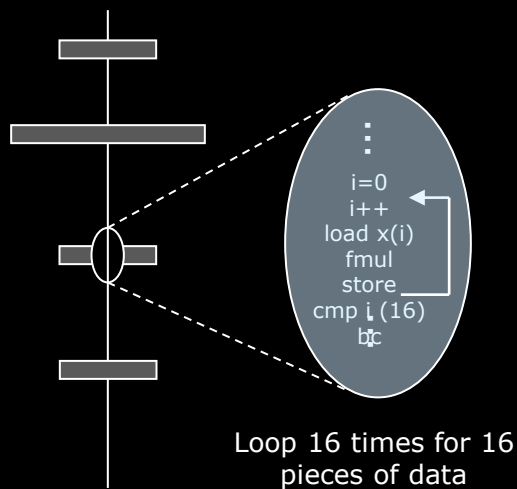
THAT WAS TRIVIAL... MORE GENERALLY, WHAT WORKS WELL?

- 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



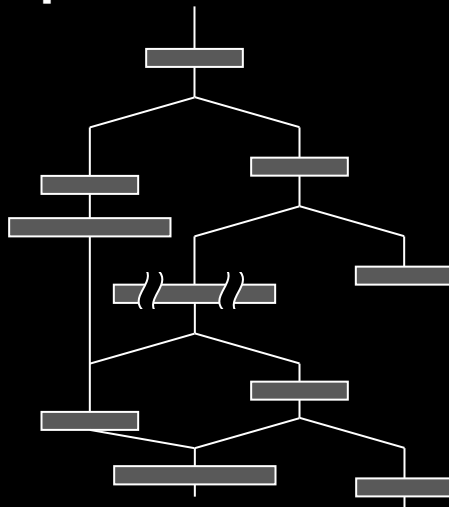
SO TO SUMMARIZE THAT

Fine-grain data parallel Code



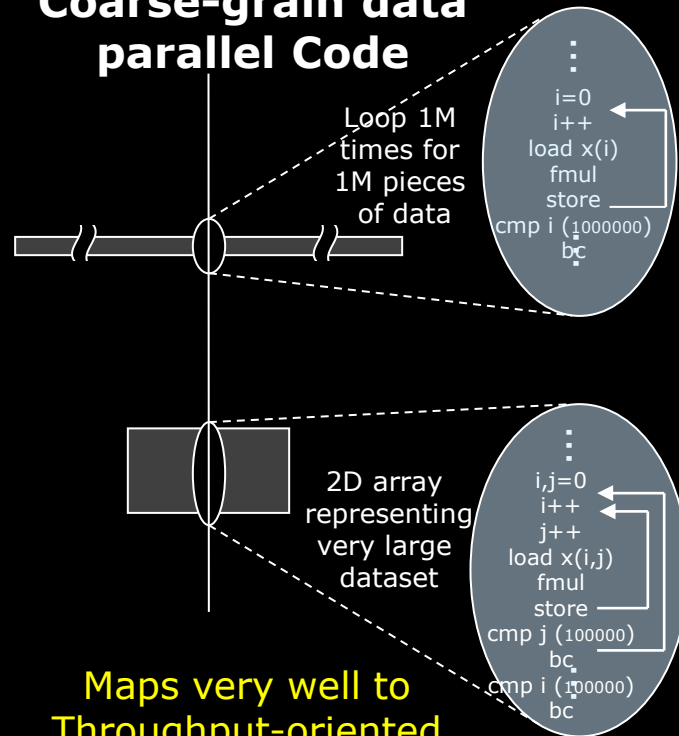
Maps very well to integrated SIMD dataflow (ie: SSE)

Nested data parallel Code



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

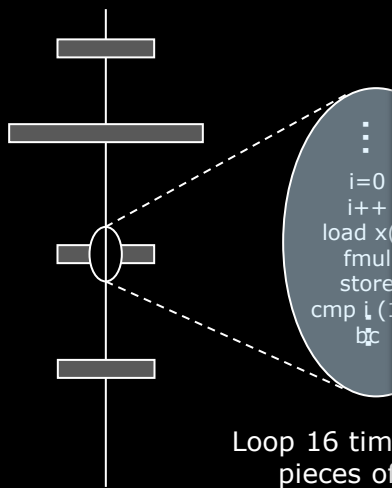
Coarse-grain data parallel Code



Maps very well to Throughput-oriented data parallel engines

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.

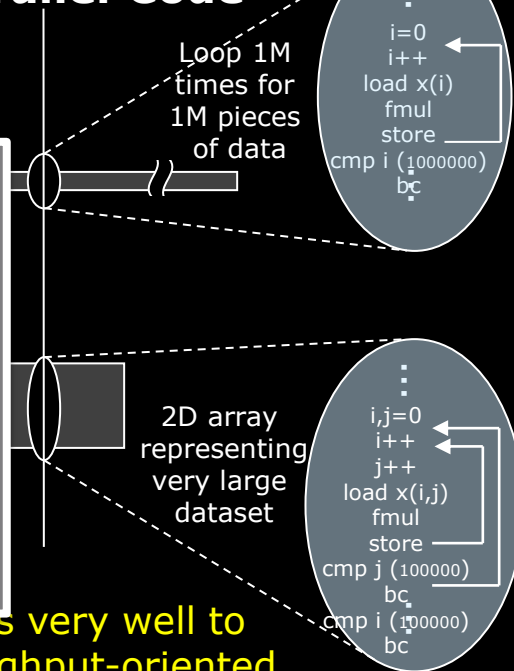
Nested data parallel/braided parallel code benefits from close coupling.

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

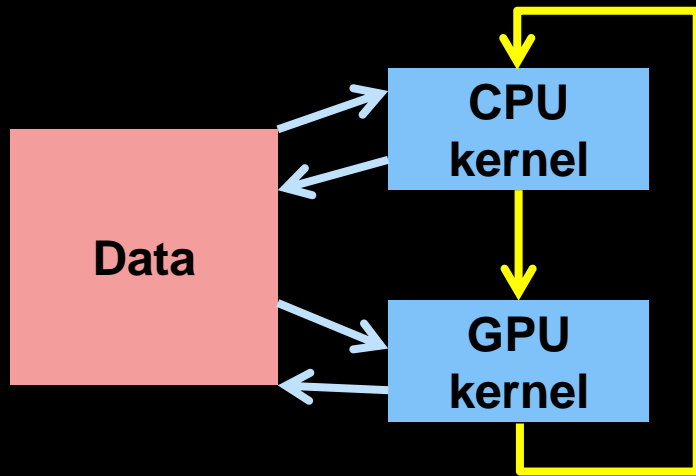
Coarse-grain data parallel Code



Maps very well to Throughput-oriented data parallel engines

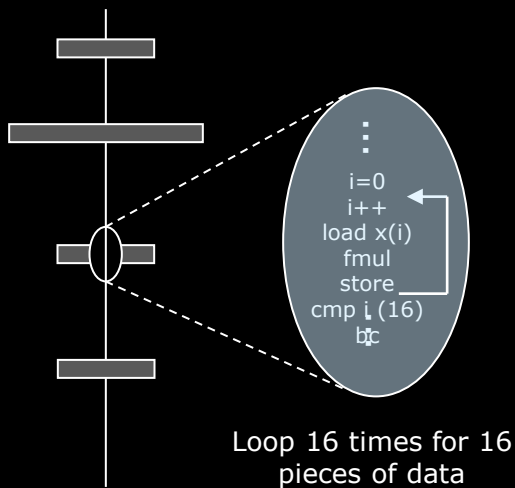
CUE APU

- 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
- 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)



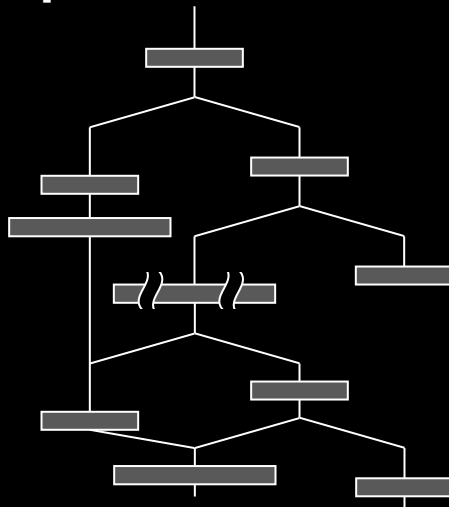
SO TO SUMMARIZE THAT

Fine-grain data parallel Code



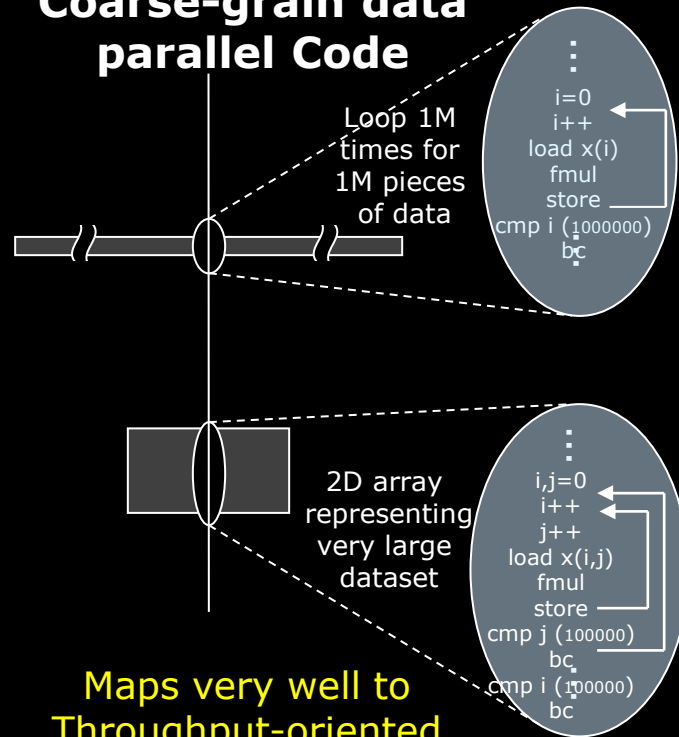
Maps very well to integrated SIMD dataflow (ie: SSE)

Nested data parallel Code



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

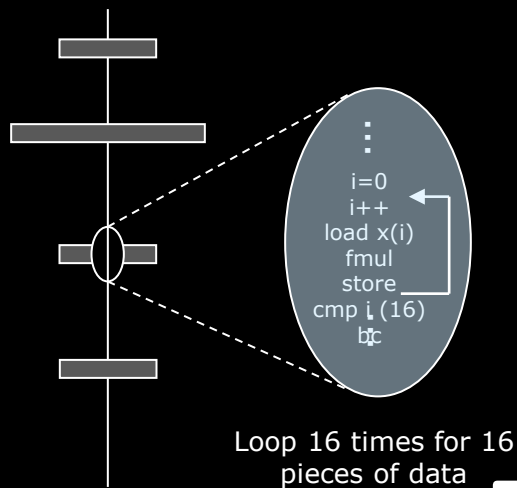
Coarse-grain data parallel Code



Maps very well to Throughput-oriented data parallel engines

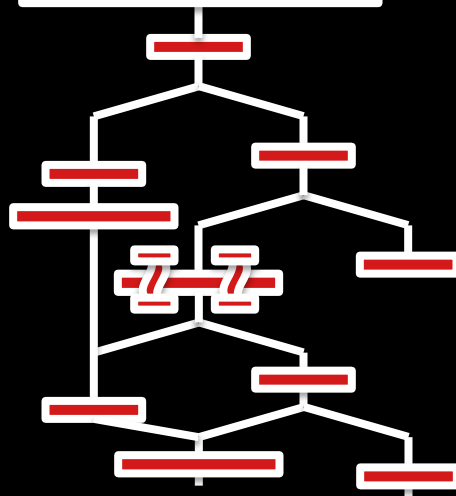
SO TO SUMMARIZE THAT

Fine-grain data parallel Code



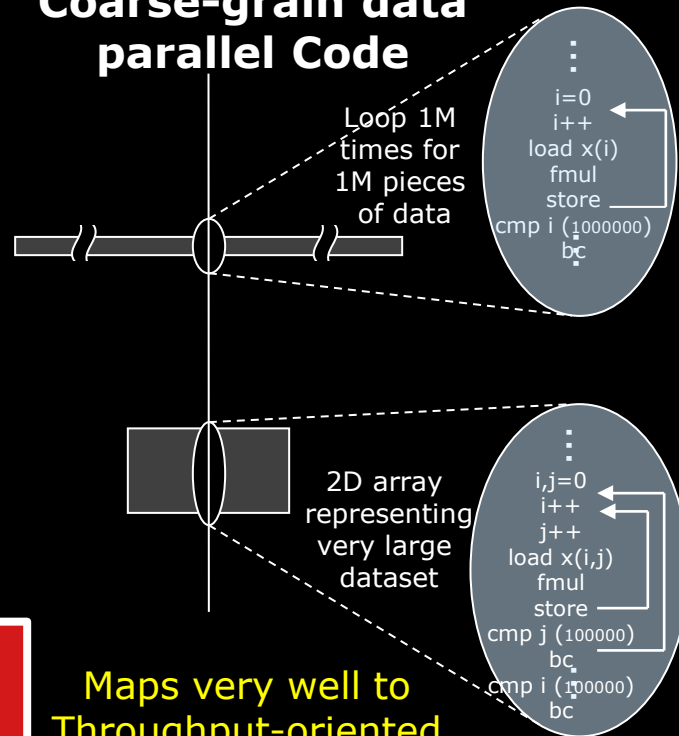
Maps very well to integrated SIMD dataflow (ie: SSE)

Nested data parallel Code



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

Coarse-grain data parallel Code



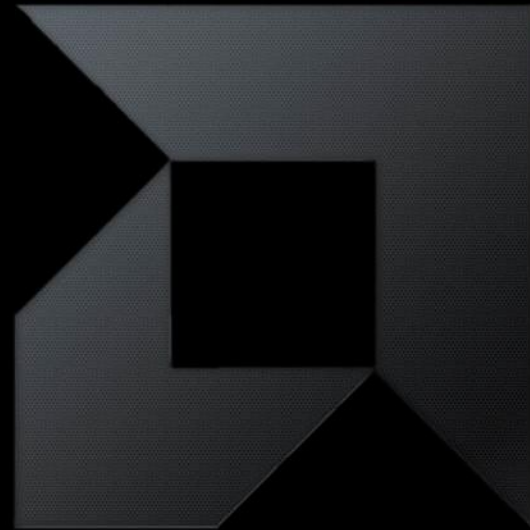
Maps very well to Throughput-oriented data parallel engines

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 programming models...



THE STATE OF THE ART
GPU PROGRAMMING MODELS IN THE PRESENT

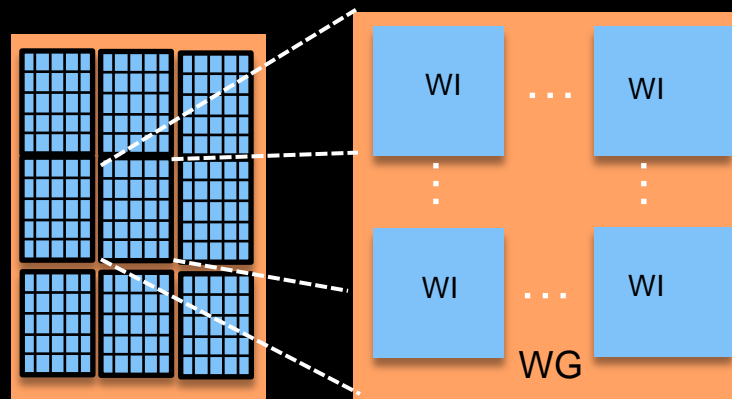


TODAY'S EXECUTION MODEL

- Single program multiple data (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.

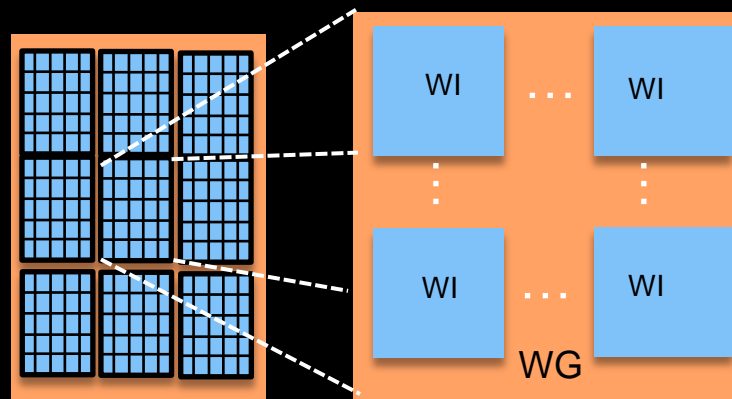
TODAY'S EXECUTION MODEL

- Single program multiple data (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.



TODAY'S EXECUTION MODEL

- Single program multiple data (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.



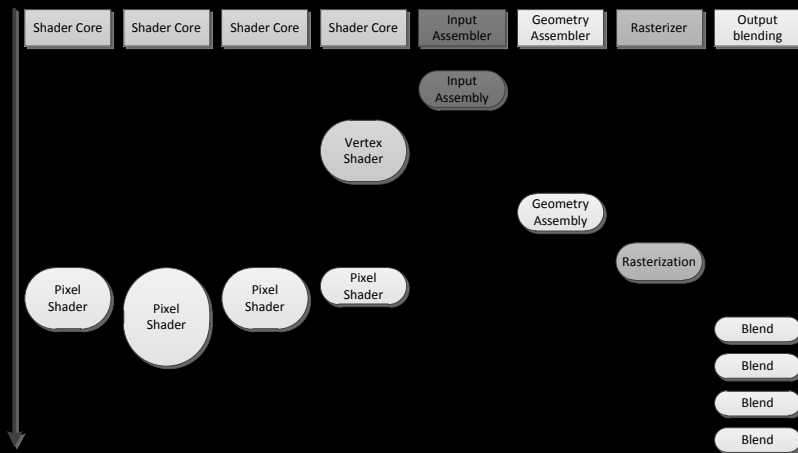
Modern CPUs & GPUs can support more !

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

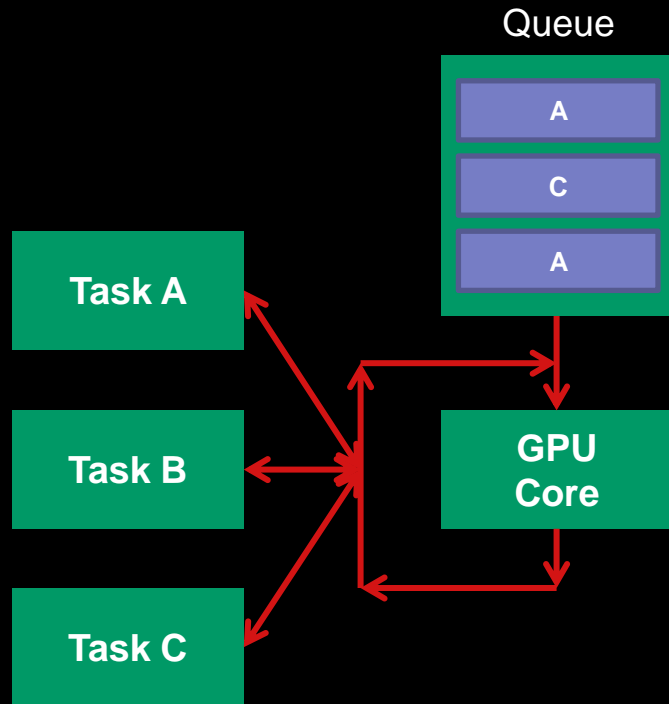
MODERN GPU (& CPU) CAPABILITIES

- 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



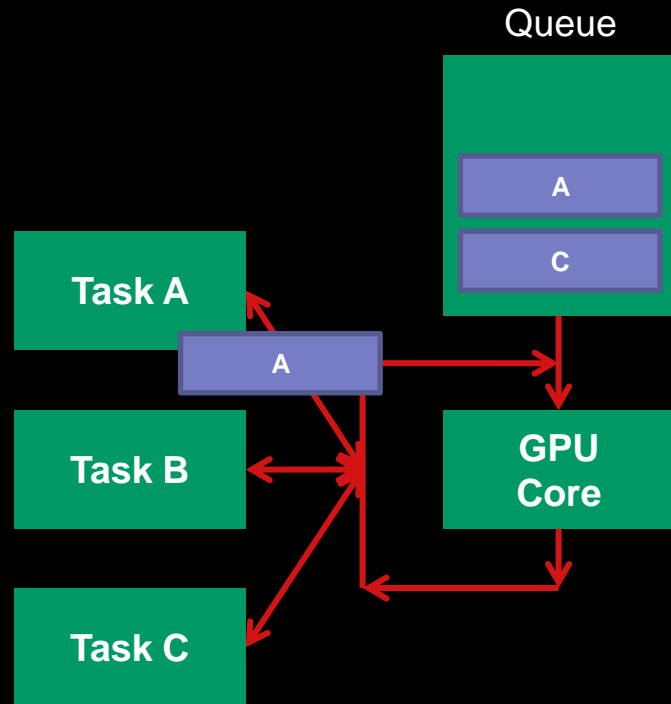
PERSISTENT THREADS

- People emulate more flexible MPMD models using “persistent threads”
 - Each core executes a scheduler loop
 - Takes tasks off a queue
 - Branches to particular code for that task
- This bypasses the hardware scheduler
 - Gives flexibility without necessary significant overhead
 - Reduces the ability of the hardware to flexibly schedule for power reduction in the absence of context switching
 - The more cores we add the worse this is



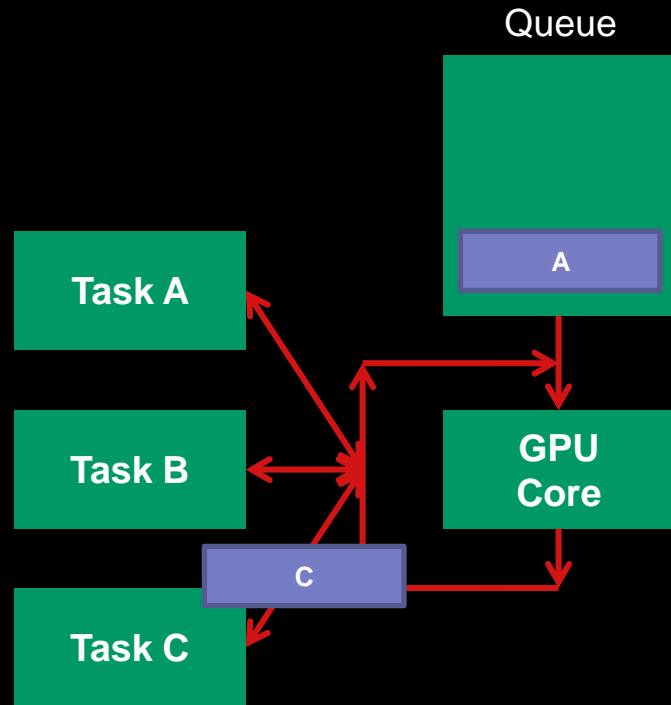
PERSISTENT THREADS

- People emulate more flexible MPMD models using “persistent threads”
 - Each core executes a scheduler loop
 - Takes tasks off a queue
 - Branches to particular code for that task
- This bypasses the hardware scheduler
 - Gives flexibility without necessary significant overhead
 - Reduces the ability of the hardware to flexibly schedule for power reduction in the absence of context switching
 - The more cores we add the worse this is



PERSISTENT THREADS

- People emulate more flexible MPMD models using “persistent threads”
 - Each core executes a scheduler loop
 - Takes tasks off a queue
 - Branches to particular code for that task
- This bypasses the hardware scheduler
 - Gives flexibility without necessary significant overhead
 - Reduces the ability of the hardware to flexibly schedule for power reduction in the absence of context switching
 - The more cores we add the worse this is



OPENCL, CUDA, C++AMP, 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

THE WORLD IS CHANGING, SLOWLY

- These models are not static
 - Obviously there are serious downsides to that...
- OpenCL 1.2 was recently released, 2.0 is in development and making good progress
- NVIDIA has been adding features to CUDA with some persistence

- Microsoft has decided on a single source model called C++AMP
 - Sits on top of DirectX
 - Highly limited as a result of this (some way behind CUDA and OpenCL currently)
 - However, from an ease of access point of view there are clear benefits
- For example:

```
void ampSquareExample(const std::vector<int> &in, std::vector<int> &out) {  
    concurrency::array_view<const int> avIn(in.size(), in);  
    concurrency::array_view<int> avOut(out.size(), out);  
    concurrency::parallel_for_each(avOut.extent, [=](concurrency::index<1> idx) restrict(amp) {  
        avOut[idx] = avIn[idx] * avIn[idx];  
    });  
    avOut.synchronize();  
}
```

OPENACC

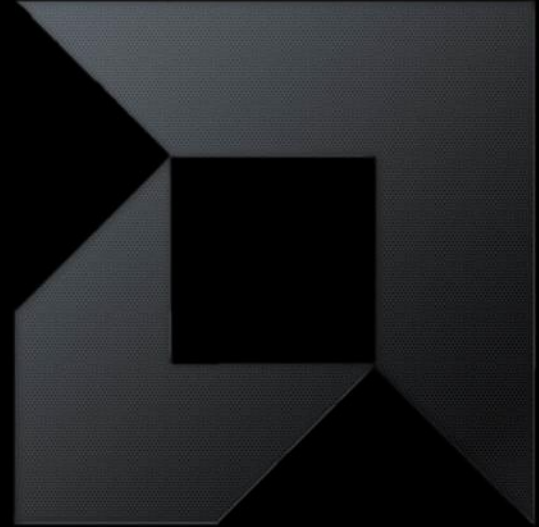
- Looking at it from the OpenMP pragma directive angle we see OpenACC
 - Initially developed by PGI, NVIDIA, Cray and CAPS
- Clear benefits for developers starting with C or fortran source
 - Though it's a shame to not be integrated cleanly with the language and type system
- The aim is to merge this technology with OpenMP and, quoting from the OpenACC web site:
 - *The intent is that the lessons learned from use an implementations of the OpenACC API will lead to a more complete and robust OpenMP heterogeneous computing standard...*

```
#pragma acc parallel...
```

HOW DO WE MOVE FORWARD?

- We should concentrate on ways to abstract the algorithms over the features that differ?
 - A difficult challenge
 - For now we seem to be stuck with inefficient raw-data-parallelism or expertly coded SIMD algorithms
- First of all let's look at making the architecture more flexible
 - From there we can more flexibly think about programming models
 - We become less restricted

***IMPROVING THE GPU'S
PROGRAMMABILITY
THE HD7970 AND GRAPHICS CORE NEXT***



GPU EXECUTION AS WAS

- We often view GPU programming as a set of independent threads, more reasonably known as “work items” in OpenCL:

```
kernel void blah(global float *input, global float *output) {  
    output[get_global_id(0)] = input[get_global_id(0)];  
}
```

- Which we flatten to an intermediate language known as AMD IL:

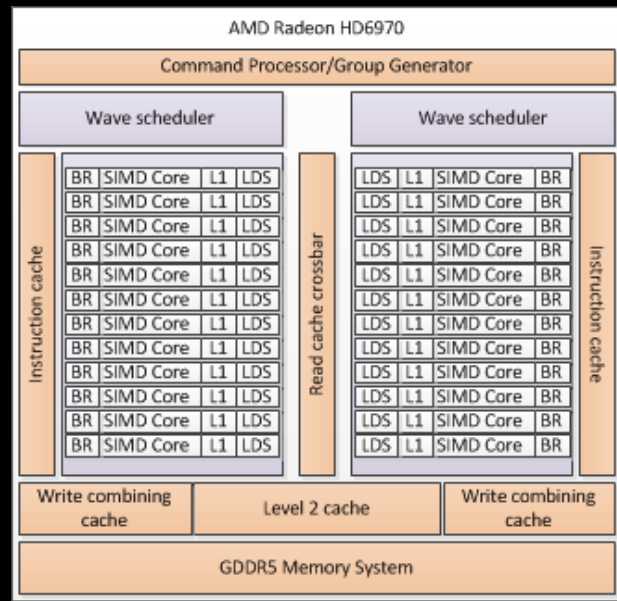
- Note that AMD IL contains short vector instructions

```
mov r255, r1021.xyz0  
mov r255, r255.x000  
mov r256, l9.xxxx  
ishl r255.x____, r255.xxxx, r256.xxxx  
iadd r253.x____, r2.xxxx, r255.xxxx  
mov r255, r1022.xyz0  
mov r255, r255.x000  
ishl r255.x____, r255.xxxx, r256.xxxx  
iadd r254.x____, r1.xxxx, r255.xxxx  
mov r1010.x____, r254.xxxx  
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx  
mov r254.x____, r1011.xxxx  
mov r1011.x____, r254.xxxx  
mov r1010.x____, r253.xxxx  
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx  
ret
```



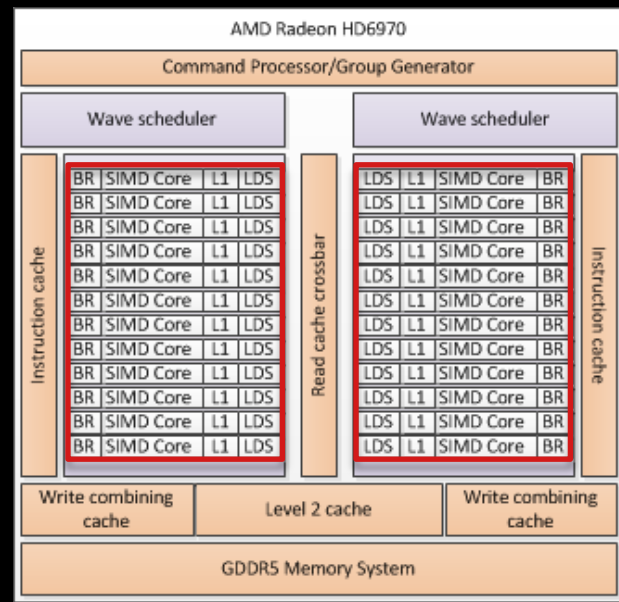
MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:



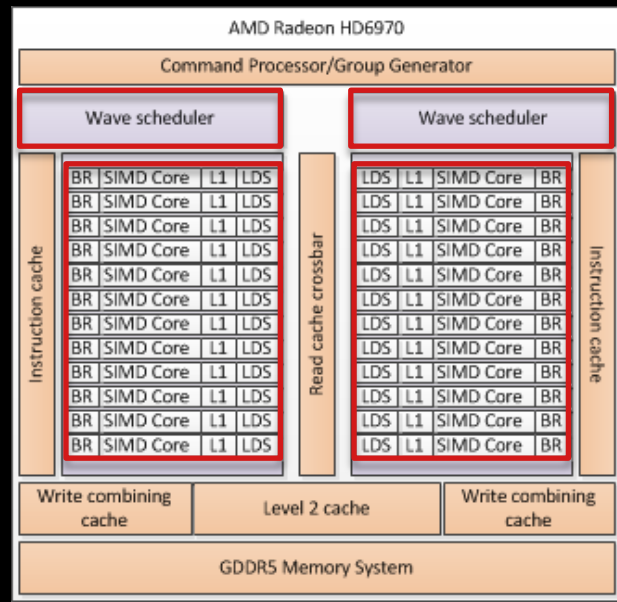
MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:
 - The HD6970 architecture has 24 vector cores



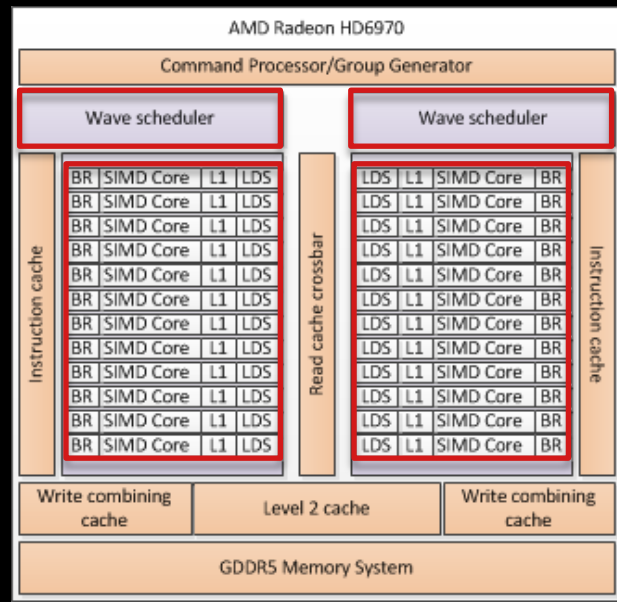
MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:
 - The HD6970 architecture has 24 vector cores
 - Each half of the device has a wave scheduler
 - This can be seen as a shared, massively threaded, scalar core



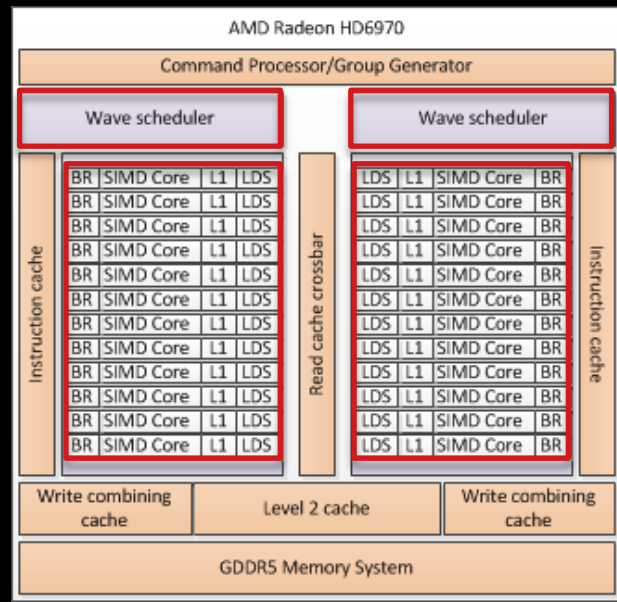
MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:
 - The HD6970 architecture has 24 vector cores
 - Each half of the device has a wave scheduler
 - This can be seen as a shared, massively threaded, scalar core
- The device as a whole can run up to about 500 threads



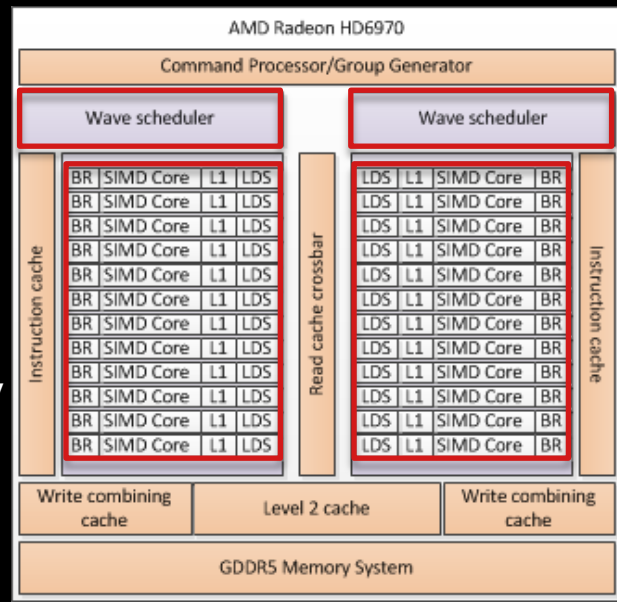
MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:
 - The HD6970 architecture has 24 vector cores
 - Each half of the device has a wave scheduler
 - This can be seen as a shared, massively threaded, scalar core
- The device as a whole can run up to about 500 threads
 - 500 program counters across the two schedulers



MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:
 - The HD6970 architecture has 24 vector cores
 - Each half of the device has a wave scheduler
 - This can be seen as a shared, massively threaded, scalar core
- The device as a whole can run up to about 500 threads
 - 500 program counters across the two schedulers
 - The device can execute around 32000 work items concurrently



MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
 - Each work item describes a lane of execution
 - Multiple work items execute together in SIMD fashion with a single program counter
 - Some clever automated stack management to handle divergent control flow across the vector



MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
 - Each work item describes a lane of execution
 - Multiple work items execute together in SIMD fashion with a single program counter
 - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
 - Each work item describes a lane of execution
 - Multiple work items execute together in SIMD fashion with a single program counter
 - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
 - Each work item describes a lane of execution
 - Multiple work items execute together in SIMD fashion with a single program counter
 - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, r9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, r9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, r9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
 - Each work item describes a lane of execution
 - Multiple work items execute together in SIMD fashion with a single program counter
 - Some clever automated stack management to handle divergent control flow across the vector

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r253.x____, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```


MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads
- The reality is that high-end GPUs follow a SIMD architecture
 - Each work item describes a lane of execution
 - Multiple work items execute together in SIMD fashion with a single program counter
 - Some clever automated stack management to handle divergent control flow across the vector

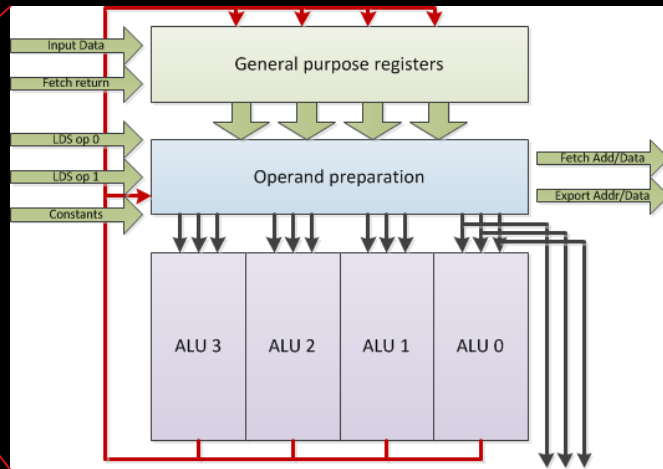
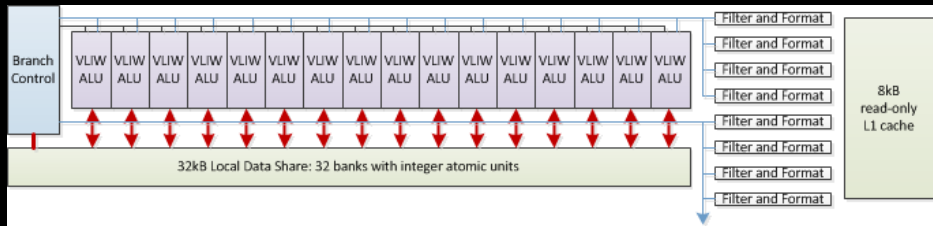
```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
ladd r253.x____, r2.xxxx, r253.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
ladd r253.x____, r2.xxxx, r253.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x____, r255.xxxx, r256.xxxx
ladd r253.x____, r2.xxxx, r253.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x____, r255.xxxx, r256.xxxx
iadd r254.x____, r1.xxxx, r255.xxxx
mov r1010.x____, r254.xxxx
uav_raw_load_id(11)_cached r1011.x____, r1010.xxxx
mov r254.x____, r1011.xxxx
mov r1011.x____, r254.xxxx
mov r1010.x____, r253.xxxx
uav_raw_store_id(11) mem.x____, r1010.xxxx, r1011.xxxx
ret
```

IT WAS NEVER QUITE THAT SIMPLE

- The HD6970 architecture and its predecessors were combined multicore SIMD/VLIW machines
 - Data-parallel through hardware vectorization
 - Instruction parallel through both multiple cores and VLIW units
- The HD6970 issued a 4-way VLIW instruction per work item
 - Architecturally you could view that as a 4-way VLIW instruction issue per SIMD lane
 - Alternatively you could view it as a 4-way VLIW issue of SIMD instructions



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; ----- Disassembly -----  
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)  
  0 w: LSHL    ____, R0.x, 2  
  1 z: ADD_INT ____, KC0[0].x, PV0.w  
  2 y: LSHR    R0.y, PV1.z, 2  
  3 x: MULLO_INT R1.x, R1.x, KC1[1].x  
    y: MULLO_INT ____, R1.x, KC1[1].x  
    z: MULLO_INT ____, R1.x, KC1[1].x  
    w: MULLO_INT ____, R1.x, KC1[1].x  
01 TEX: ADDR(48) CNT(1)  
  4 VFETCH R2.x____, R0.y, fc153  
    FETCH_TYPE(NO_INDEX_OFFSET)  
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)  
  5 w: ADD_INT ____, R0.x, R1.x  
  6 z: ADD_INT ____, PV5.w, KC0[6].x  
  7 y: LSHL    ____, PV6.z, 2  
  8 x: ADD_INT ____, KC1[1].x, PV7.y  
  9 x: LSHR    R0.x, PV8.x, 2  
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x____, R2, ARRAY_SIZE(4) MARK VPM  
04 END  
END_OF_PROGRAM
```



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

: ----- Disassembly -----

00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)

0 w: LSHL _____, R0.x, 2

1 z: ADD_INT _____, KC0[0].x, PV0.w

2 y: LSHR R0.y, PV1.z, 2

3 x: MULLO_INT R1.x, R1.x, KC1[1].x

y: MULLO_INT _____, R1.x, KC1[1].x

z: MULLO_INT _____, R1.x, KC1[1].x

w: MULLO_INT _____, R1.x, KC1[1].x

01 TEX: ADDR(48) CNT(1)

4 VFETCH R2.x____, R0.y, fc153

FETCH_TYPE(NO_INDEX_OFFSET)

02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)

5 w: ADD_INT _____, R0.x, R1.x

6 z: ADD_INT _____, PV5.w, KC0[6].x

7 y: LSHL _____, PV6.z, 2

8 x: ADD_INT _____, KC1[1].x, PV7.y

9 x: LSHR R0.x, PV8.x, 2

03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x____, R2, ARRAY_SIZE(4) MARK VPM

04 END

END_OF_PROGRAM

Clause header

Work executed by the
shared scalar unit



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

: ----- Disassembly -----

00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)

```
0 w: LSHL    ____, R0.x, 2
1 z: ADD_INT ____, KC0[0].x, PV0.w
2 y: LSHR    R0.y, PV1.z, 2
3 x: MULLO_INT R1.x, R1.x, KC1[1].x
  y: MULLO_INT ____, R1.x, KC1[1].x
  z: MULLO_INT ____, R1.x, KC1[1].x
  w: MULLO_INT ____, R1.x, KC1[1].x
```

01 TEX: ADDR(48) CNT(1)

```
4 VFETCH R2.x ____, R0.y, fc153
  FETCH_TYPE(NO_INDEX_OFFSET)
```

02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)

```
5 w: ADD_INT ____, R0.x, R1.x
6 z: ADD_INT ____, PV5.w, KC0[6].x
7 y: LSHL    ____, PV6.z, 2
8 x: ADD_INT ____, KC1[1].x, PV7.y
9 x: LSHR    R0.x, PV8.x, 2
```

03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x ____, R2, ARRAY_SIZE(4) MARK VPM

04 END

END_OF_PROGRAM

Clause header

Work executed by the
shared scalar unit

Clause body

Units of work dispatched
by the shared scalar unit



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

: ----- Disassembly -----

00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)

0 w: LSHL _____, R0.x, 2

1 z: ADD_INT _____, KC0[0].x, PV0.w

2 y: LSHR R0.y, PV1.z, 2

3 x: MULLO_INT R1.x, R1.x, KC1[1].x

y: MULLO_INT _____, R1.x, KC1[1].x

z: MULLO_INT _____, R1.x, KC1[1].x

w: MULLO_INT _____, R1.x, KC1[1].x

01 TEX: ADDR(48) CNT(1)

4 VFETCH R2.x____, R0.y, fc153

FETCH_TYPE(NO_INDEX_OFFSET)

02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)

5 w: ADD_INT _____, R0.x, R1.x

6 z: ADD_INT _____, PV5.w, KC0[6].x

7 y: LSHL _____, PV6.z, 2

8 x: ADD_INT _____, KC1[1].x, PV7.y

9 x: LSHR R0.x, PV8.x, 2

03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x____, R2, ARRAY_SIZE(4) MARK VPM

04 END

END_OF_PROGRAM

Clause header

Work executed by the
shared scalar unit

VLIW instruction packet

Compiler-generated instruction level
parallelism for the VLIW unit.
Each instruction (x, y, z, w) executed
across the vector.

Clause body

Units of work dispatched
by the shared scalar unit



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

: ----- Disassembly -----

00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)

0 w: LSHL _____, R0.x, 2

1 z: ADD_INT _____, KC0[0].x, PV0.w

2 y: LSHR _____, R0.y, PV1.z, 2

3 x: MULLO_INT R1.x, R1.x, KC1[1].x

y: MULLO_INT _____, R1.x, KC1[1].x

z: MULLO_INT _____, R1.x, KC1[1].x

w: MULLO_INT _____, R1.x, KC1[1].x

01 TEX: ADDR(48) CNT(1)

4 VFETCH R2.x____, R0.y, fc153

FETCH_TYPE(NO_INDEX_OFFSET)

02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)

5 w: ADD_INT _____, R0.x, R1.x

6 z: ADD_INT _____, PV5.w, KC0[6].x

7 y: LSHL _____, PV6.z, 2

8 x: ADD_INT _____, KC1[1].x, PV7.y

9 x: LSHR _____, R0.x, PV8.x, 2

03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x____, R2, ARRAY_SIZE(4) MARK VPM

04 END

END_OF_PROGRAM

Clause header

Work executed by the
shared scalar unit

VLIW instruction packet

Compiler-generated instruction level
parallelism for the VLIW unit.
Each instruction (x, y, z, w) executed
across the vector.

Clause body

Units of work dispatched
by the shared scalar unit

Notice the poor occupancy of VLIW slots

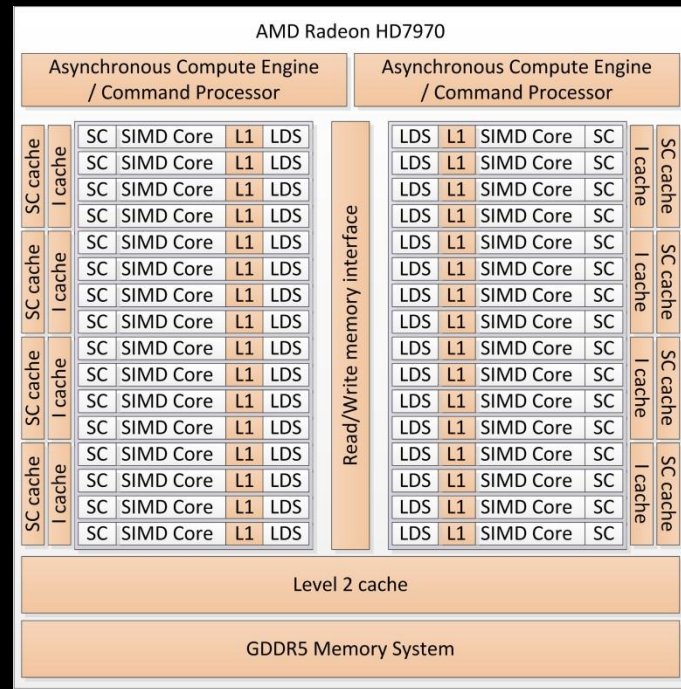


WHY DID WE SEE INEFFICIENCY?

- The architecture was well suited to graphics workloads:
 - VLIW was easily filled by the vector-heavy graphics kernels
 - Minimal control flow meant that the monolithic, shared thread scheduler was relatively efficient
- Unfortunately, workloads change with time.
- So how did we change the architecture to improve the situation?

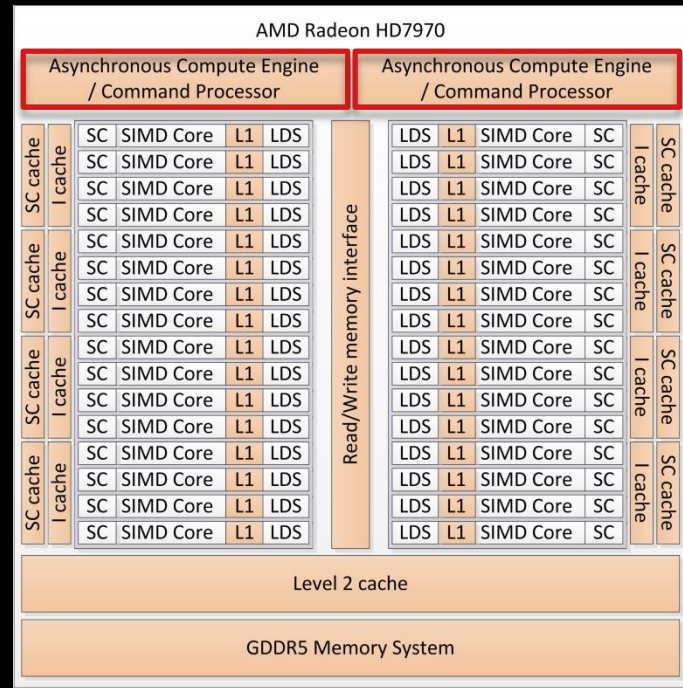
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different



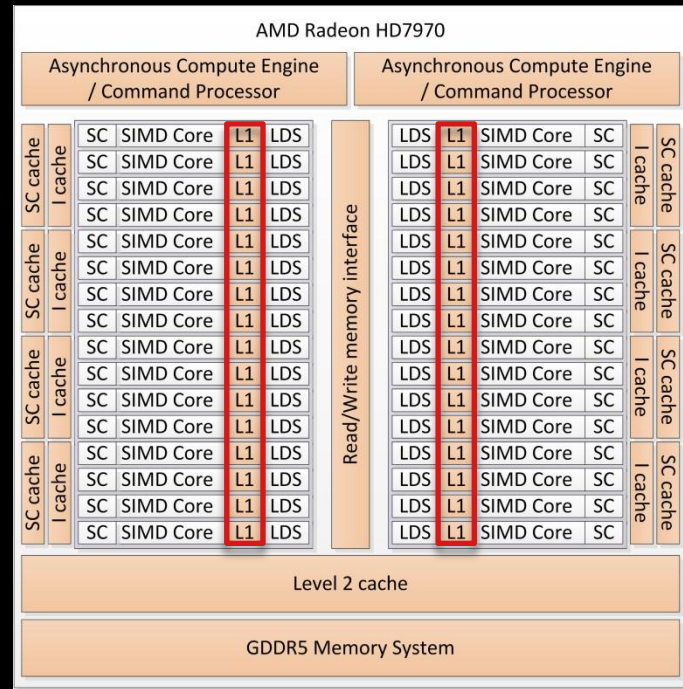
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different
- Two command processors
 - Capable of processing two command queues concurrently



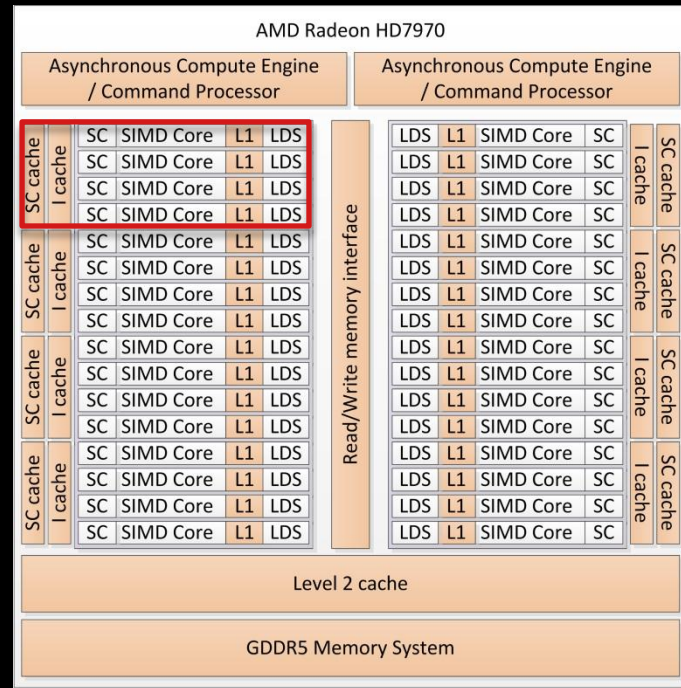
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different
- Two command processors
 - Capable of processing two command queues concurrently
- Full read/write L1 data caches
- SIMD cores grouped in fours
 - Scalar data and instruction cache per cluster
 - L1, LDS and scalar processor per core
- Up to 32 cores / compute units



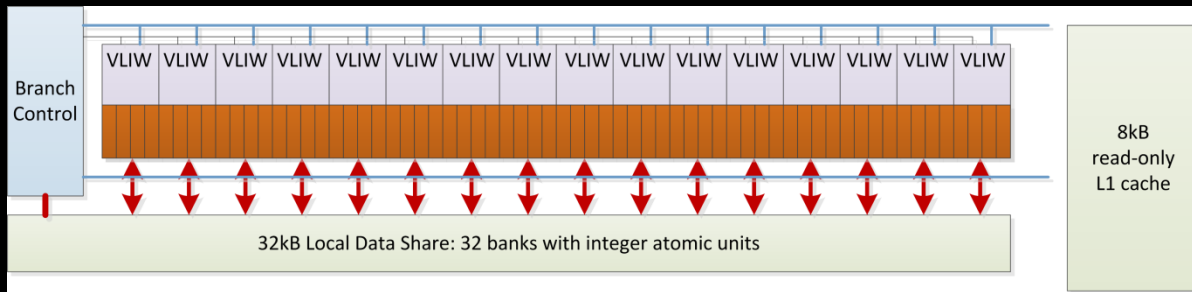
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different
- Two command processors
 - Capable of processing two command queues concurrently
- Full read/write L1 data caches
- SIMD cores grouped in fours
 - Scalar data and instruction cache per cluster
 - L1, LDS and scalar processor per core
- Up to 32 cores / compute units



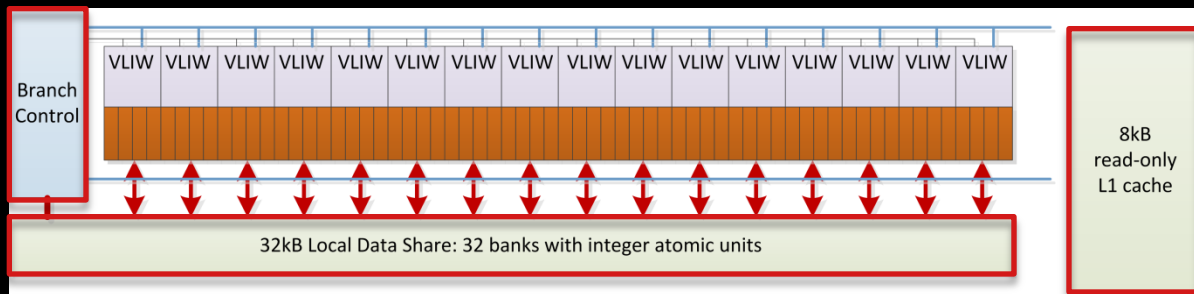
THE SIMD CORE

- The SIMD unit on the HD6970 architecture had a branch control but full scalar execution was performed globally



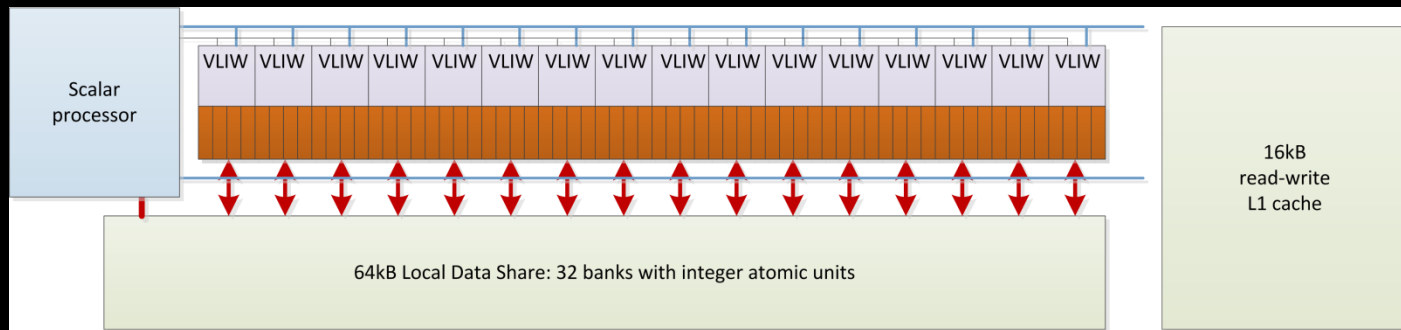
THE SIMD CORE

- The SIMD unit on the HD6970 architecture had a branch control but full scalar execution was performed globally



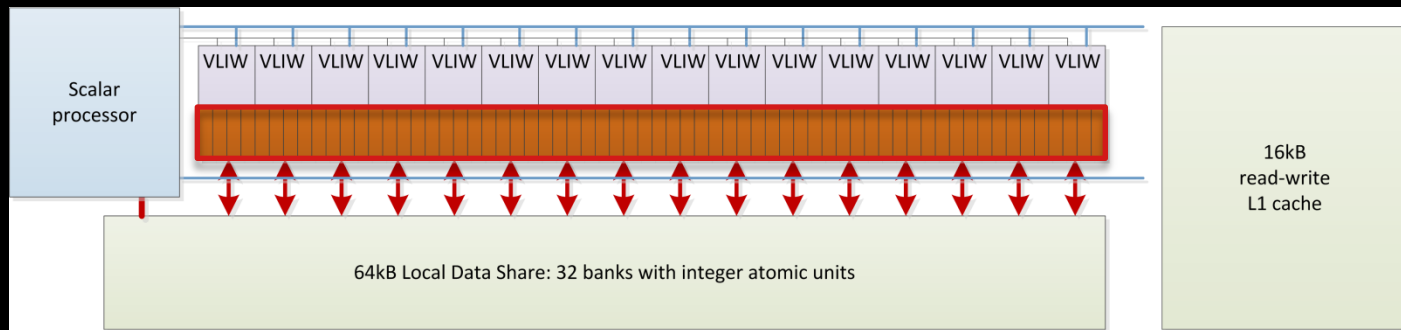
THE SIMD CORE

- On the HD7970 we have a full scalar processor and the L1 cache and LDS have been doubled in size



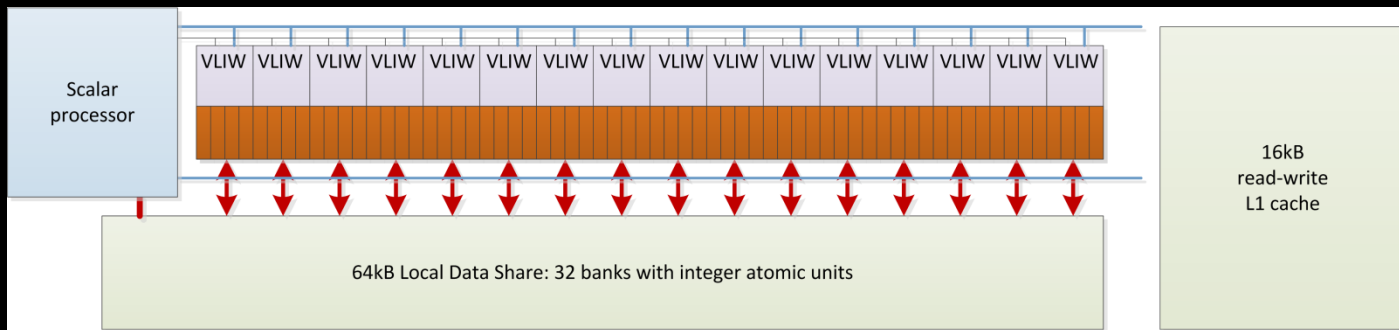
THE SIMD CORE

- On the HD7970 we have a full scalar processor and the L1 cache and LDS have been doubled in size
- Then let us consider the VLIW ALUs



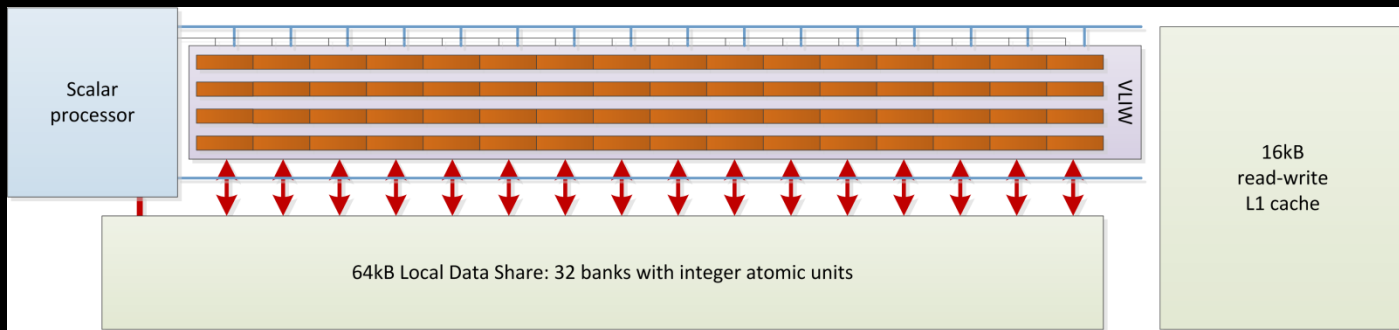
THE SIMD CORE

- Remember we could view the architecture two ways:
 - An array of VLIW units



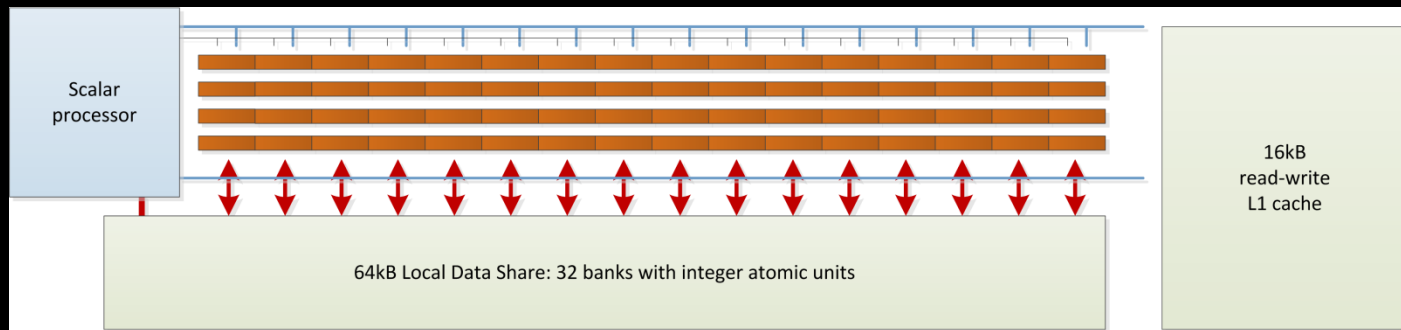
THE SIMD CORE

- Remember we could view the architecture two ways:
 - An array of VLIW units
 - A VLIW cluster of vector units



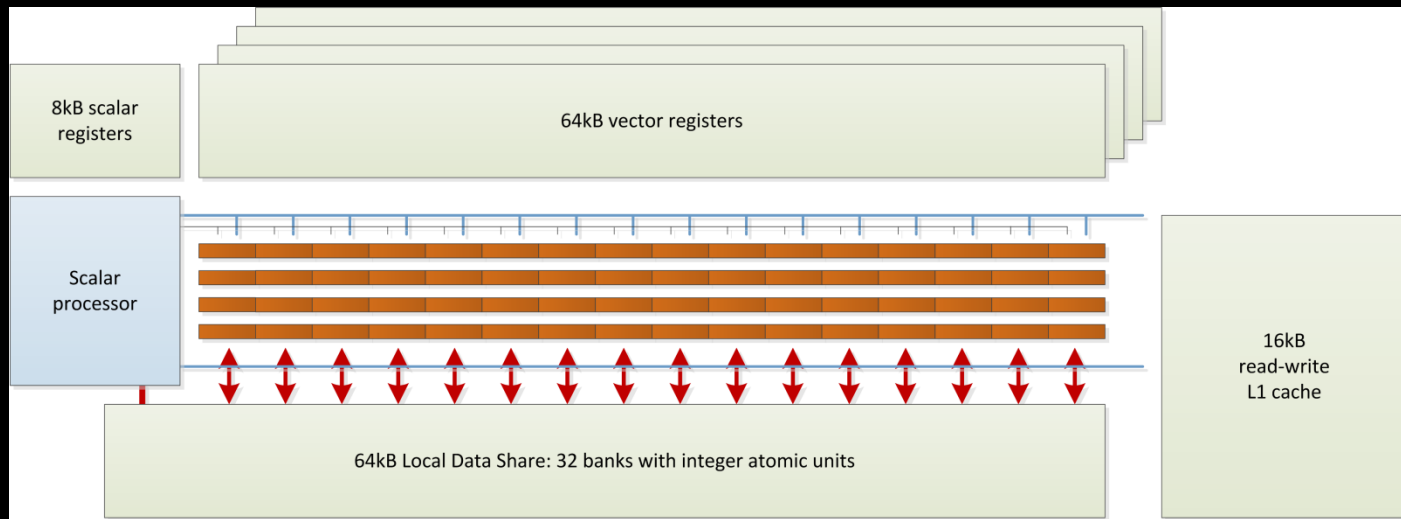
THE SIMD CORE

- Now that we have a scalar processor we can dynamically schedule instructions rather than relying on the compiler
- No VLIW!



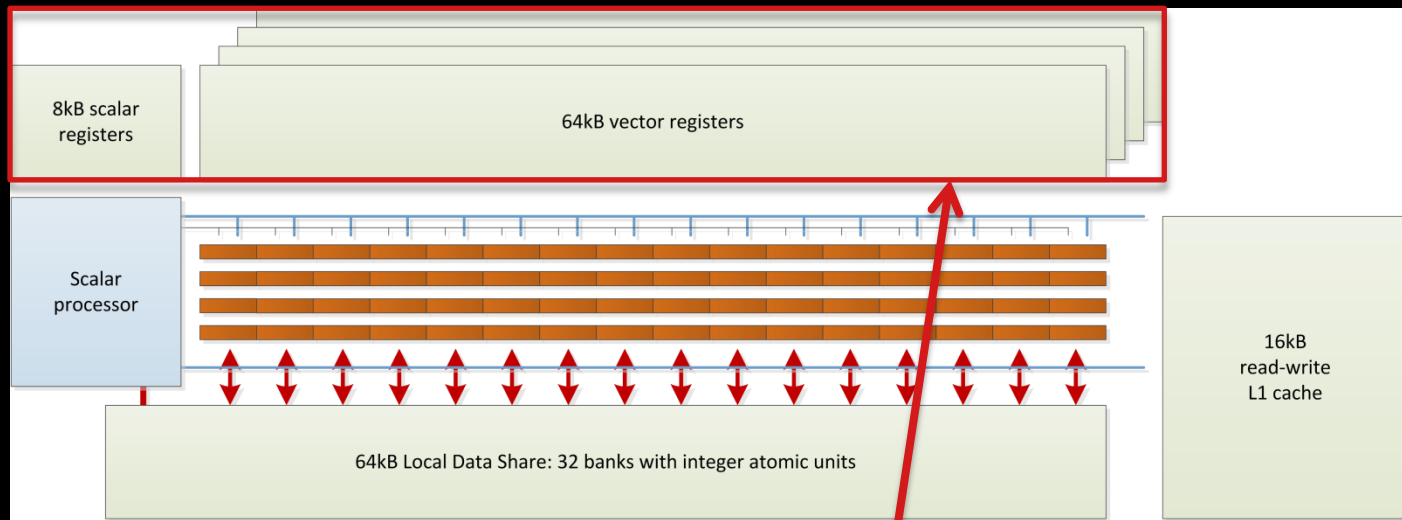
- The heart of Graphics Core Next:
 - A scalar processor with four 16-wide vector units
 - Each lane of the vector, and hence each IL work item, is now scalar

THE SIMD CORE



- The scalar core manages a large number of threads
 - Each thread requires its set of vector registers
 - Significant register state for both scalar and vector storage
 - 10 waves per SIMD, 40 waves per CU (core), 2560 work items per CU, 81920 work items on the HD7970

THE SIMD CORE



- The scalar core manages a large number of threads
 - Each thread requires its set of vector registers
 - Significant register state for both scalar and vector storage
 - 10 waves per SIMD, 40 waves per CU (core), 2560 work items per CU, 81920 work items on the HD7970

THE NEW ISA

- Simpler and more efficient
- Instructions for both sets of execution units inline

```
s_buffer_load_dword s0, s[4:7], 0x04
s_buffer_load_dword s1, s[4:7], 0x18
s_buffer_load_dword s4, s[8:11], 0x00
s_waitcnt lgkmcnt(0)
s_mul_i32 s0, s12, s0
s_add_i32 s0, s0, s1
v_add_i32 v0, vcc, s0, v0
v_lshlrev_b32 v0, 2, v0
v_add_i32 v1, vcc, s4, v0
s_load_dwordx4 s[4:7], s[2:3], 0x50
s_waitcnt lgkmcnt(0)
tbuffer_load_format_x v1, v1, s[4:7],
    0 offcn format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_buffer_load_dword s0, s[8:11], 0x04
s_load_dwordx4 s[4:7], s[2:3], 0x58
s_waitcnt lgkmcnt(0)
v_add_i32 v0, vcc, s0, v0
s_waitcnt vmcnt(0)
tbuffer_store_format_x v1, v0, s[4:7],
    0 offcn format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_endpgm
```



THE NEW ISA

- Simpler and more efficient
- Instructions for both sets of execution units inline

```
s_buffer_load_dword s0, s[4:7], 0x04
s_buffer_load_dword s1, s[4:7], 0x18
s_buffer_load_dword s4, s[8:11], 0x00
s_waitcnt lgkmcnt(0)
s_mul_i32 s0, s12, s0
s_add_i32 s0, s0, s1
v_add_i32 v0, vcc, s0, v0
v_lshlrev_b32 v0, 2, v0
v_add_i32 v1, vcc, s4, v0
s_load_dwordx4 s[4:7], s[2:3], 0x50
s_waitcnt lgkmcnt(0)
tbuffer_load_format_x v1, v1, s[4:7],
    0 open format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_buffer_load_dword s0, s[8:11], 0x04
s_load_dwordx4 s[4:7], s[2:3], 0x58
s_waitcnt lgkmcnt(0)
v_add_i32 v0, vcc, s0, v0
s_waitcnt vmcnt(0)
tbuffer_store_format_x v1, v0, s[4:7],
    0 open format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_endpgm
```



THE NEW ISA

- Simpler and more efficient
- Instructions for both sets of execution units inline
- No clauses
 - Lower instruction scheduling latency
 - Improved performance in previously clause-bound cases
 - Lower power handling of control flow as control is closer
- No VLIW
 - Fewer compiler-induced bubbles in the instruction schedule
- Full support for exceptions, function calls and recursion

```
s_buffer_load_dword s0, s[4:7], 0x04
s_buffer_load_dword s1, s[4:7], 0x18
s_buffer_load_dword s4, s[8:11], 0x00
s_waitcnt lgkmcnt(0)
s_mul_i32 s0, s12, s0
s_add_i32 s0, s0, s1
v_add_i32 v0, vcc, s0, v0
v_lshlrev_b32 v0, 2, v0
v_add_i32 v1, vcc, s4, v0
s_load_dwordx4 s[4:7], s[2:3], 0x50
s_waitcnt lgkmcnt(0)
tbuffer_load_format_x v1, v1, s[4:7],
    0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_buffer_load_dword s0, s[8:11], 0x04
s_load_dwordx4 s[4:7], s[2:3], 0x58
s_waitcnt lgkmcnt(0)
v_add_i32 v0, vcc, s0, v0
s_waitcnt vmcnt(0)
tbuffer_store_format_x v1, v0, s[4:7],
    0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_endpgm
```


BRANCHING

```
float fn0(float a,float b)
{
    if(a>b)
        return((a-b)*a);
    else
        return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2
```

```
v_cmp_gt_f32      r0,r1          //a > b, establish VCC
s_mov_b64         s0,exec        //Save current exec mask
s_and_b64         exec,vcc,exec  //Do "if"
s_cbranch_vccz    label0         //Branch if all lanes fail
v_sub_f32         r2,r0,r1       //result = a - b
v_mul_f32         r2,r2,r0       //result=result * a

:
s_andn2_b64       exec,s0,exec   //Do "else"(s0 & !exec)
s_cbranch_execz   label1         //Branch if all lanes fail
v_sub_f32         r2,r1,r0       //result = b - a
v_mul_f32         r2,r2,r1       //result = result * b

s_mov_b64         exec,s0        //Restore exec mask
```

BRANCHING

```
float fn0(float a,float b)
{
    if(a>b)
        return((a-b)*a);
    else
        return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2
```

```
v_cmp_gt_f32    r0,r1    //a > b, establish VCC
s_mov_b64       s0,exec  //Save current exec mask
s_and_b64       exec,vcc,exec //Do "if"
s_cbranch_vccz   label0   //Branch if all lanes fail
v_sub_f32       r2,r0,r1  //result = a - b
v_mul_f32       r2,r2,r0  //result=result * a

:
s_andn2_b64     exec,s0,exec //Do "else"(s0 & !exec)
s_cbranch_execz label1     //Branch if all lanes fail
v_sub_f32       r2,r1,r0   //result = b - a
v_mul_f32       r2,r2,r1   //result = result * b

s_mov_b64       exec,s0    //Restore exec mask
```

BRANCHING

```
float fn0(float a,float b)
{
    if(a>b)
        return((a-b)*a);
    else
        return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2
```

```
v_cmp_gt_f32    r0,r1    //a > b, establish VCC
s_mov_b64       s0,exec   //Save current exec mask
s_and_b64       exec,vcc,exec //Do "if"
s_cbranch_vccz   label0    //Branch if all lanes fail
v_sub_f32       r2,r0,r1   //result = a - b
v_mul_f32       r2,r2,r0   //result=result * a

:
s_andn2_b64     exec,s0,exec //Do "else"(s0 & !exec)
s_cbranch_execz  label1    //Branch if all lanes fail
v_sub_f32       r2,r1,r0   //result = b - a
v_mul_f32       r2,r2,r1   //result = result * b

s_mov_b64       exec,s0    //Restore exec mask
```

BRANCHING

```
float fn0(float a,float b)
{
    if(a>b)
        return((a-b)*a);
    else
        return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2
```

```
v_cmp_gt_f32    r0,r1        //a > b, establish VCC
s_mov_b64       s0,exec       //Save current exec mask
s_and_b64       exec,vcc,exec //Do "if"
s_cbranch_vccz   label0       //Branch if all lanes fail
v_sub_f32       r2,r0,r1      //result = a - b
v_mul_f32       r2,r2,r0      //result=result * a
```

:

```
s_andn2_b64     exec,s0,exec //Do "else"(s0 & !exec)
s_cbranch_execz  label1       //Branch if all lanes fail
v_sub_f32       r2,r1,r0      //result = b - a
v_mul_f32       r2,r2,r1      //result = result * b

s_mov_b64       exec,s0       //Restore exec mask
```

BRANCHING

```
float fn0(float a,float b)
{
    if(a>b)
        return((a-b)*a);
    else
        return((b-a)*b);
}
```

Optional:

Use based on the number of instruction in conditional section.

- Executed in branch unit

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2
```

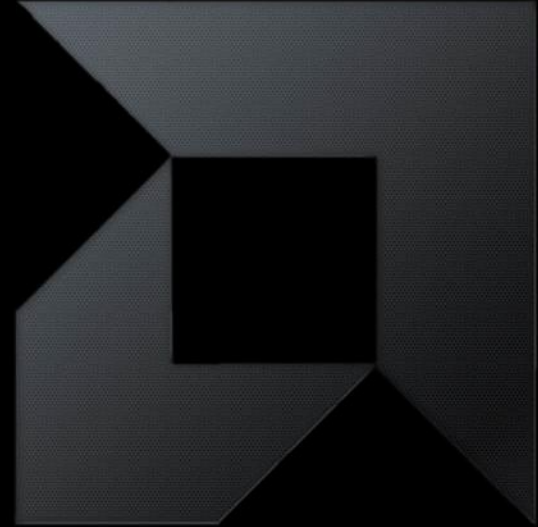
```
v_cmp_gt_f32    r0,r1        //a > b, establish VCC
s_mov_b64       s0,exec       //Save current exec mask
s_and_b64       exec,vcc,exec //Do "if"
s_cbranch_vccz   label0       //Branch if all lanes fail
v_sub_f32       r2,r0,r1      //result = a - b
v_mul_f32       r2,r2,r0      //result=result * a
```

:

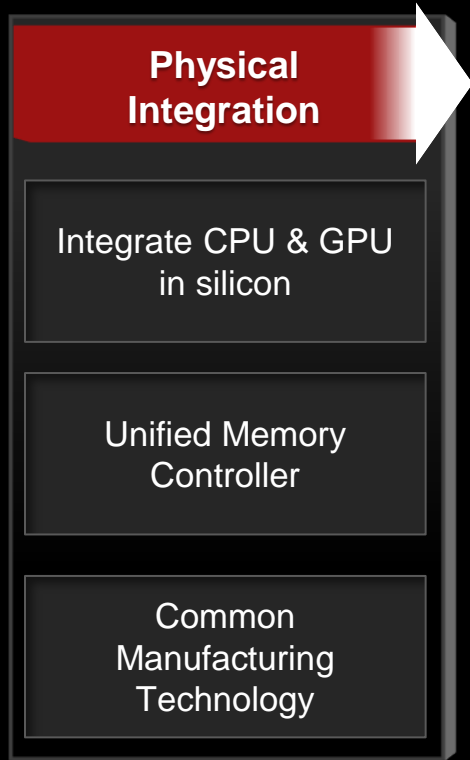
```
s_andn2_b64     exec,s0,exec //Do "else"(s0 & !exec)
s_cbranch_execz  label1       //Branch if all lanes fail
v_sub_f32       r2,r1,r0      //result = b - a
v_mul_f32       r2,r2,r1      //result = result * b

s_mov_b64       exec,s0       //Restore exec mask
```

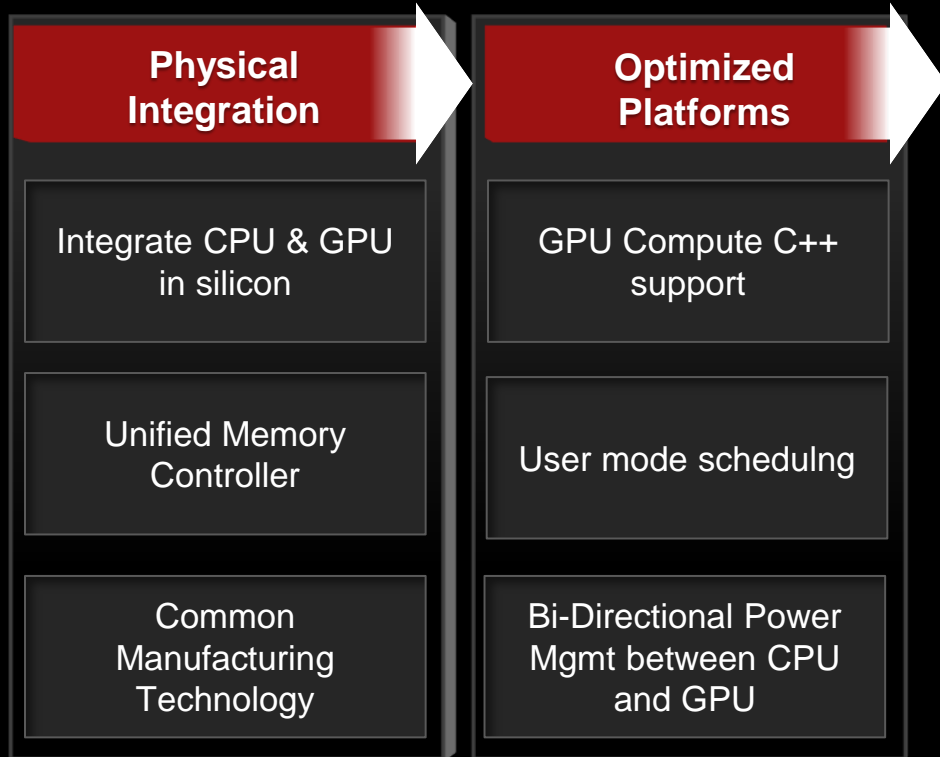
***IMPROVING SYSTEM
PROGRAMMABILITY
THE HETEROGENEOUS SYSTEM ARCHITECTURE***



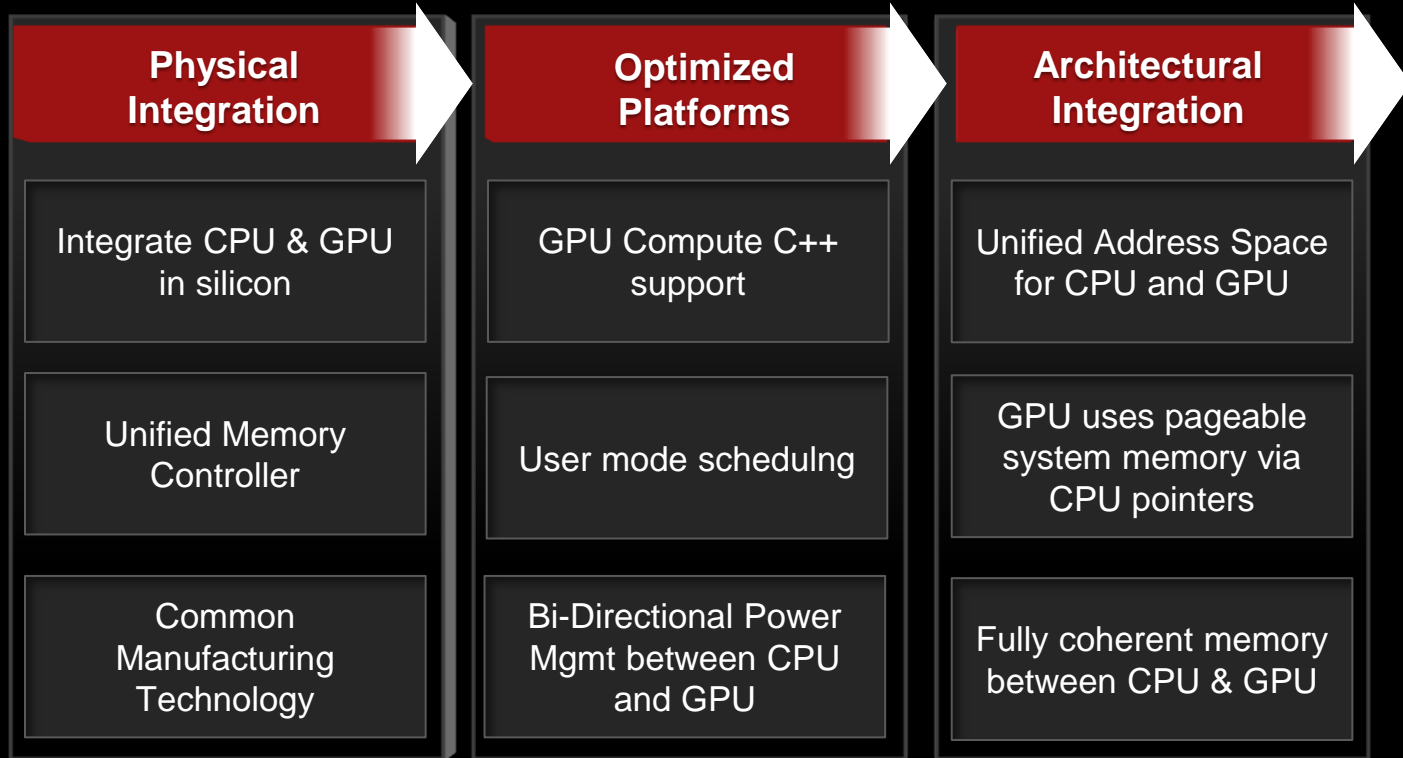
HETEROGENEOUS SYSTEM ARCHITECTURE



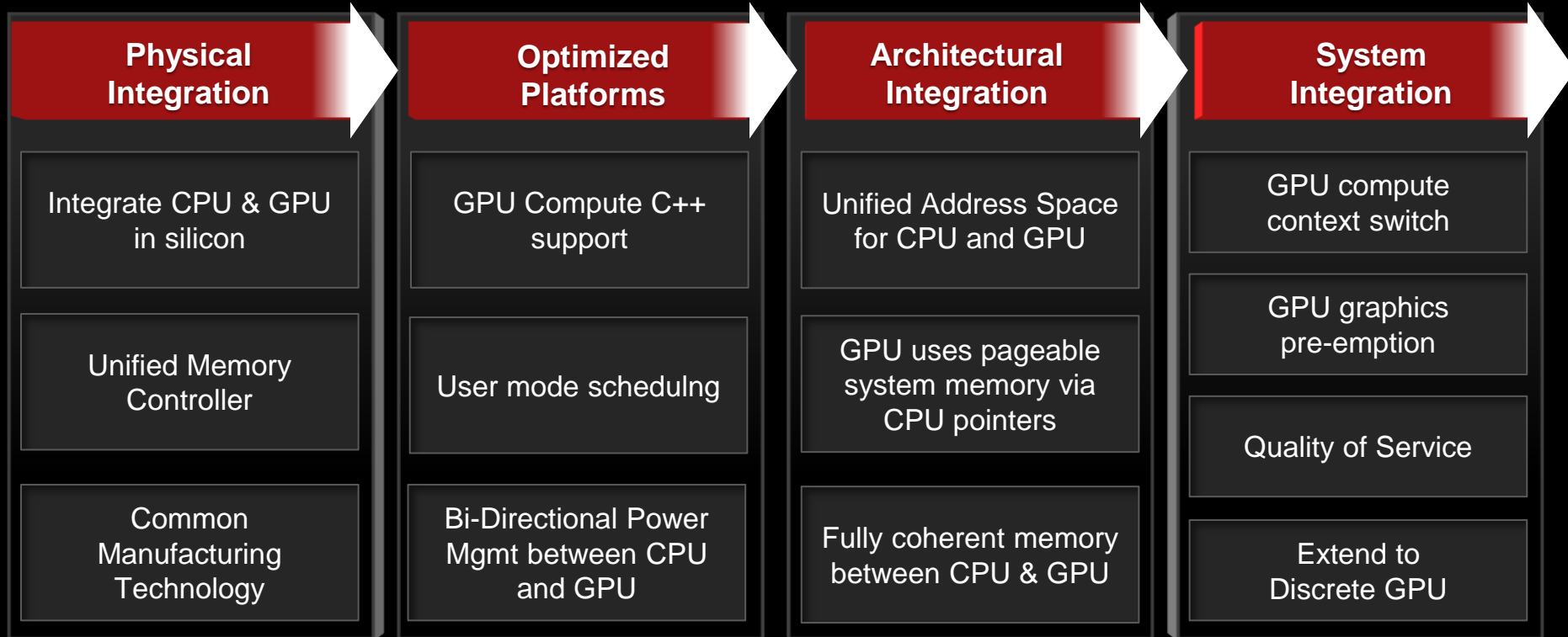
HETEROGENEOUS SYSTEM ARCHITECTURE



HETEROGENEOUS SYSTEM ARCHITECTURE



HETEROGENEOUS SYSTEM ARCHITECTURE



HETEROGENEOUS SYSTEM ARCHITECTURE – AN OPEN PLATFORM

- Open Architecture, published specifications
 - HSAIL virtual ISA
 - HSA memory model
 - HSA dispatch
- ISA agnostic for both CPU and GPU



HSA INTERMEDIATE LAYER - HSAIL

- HSAIL 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

HSA MEMORY MODEL

- Designed to be compatible with C++11, 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*, Load.Dep, Store.Release*
 - Barriers

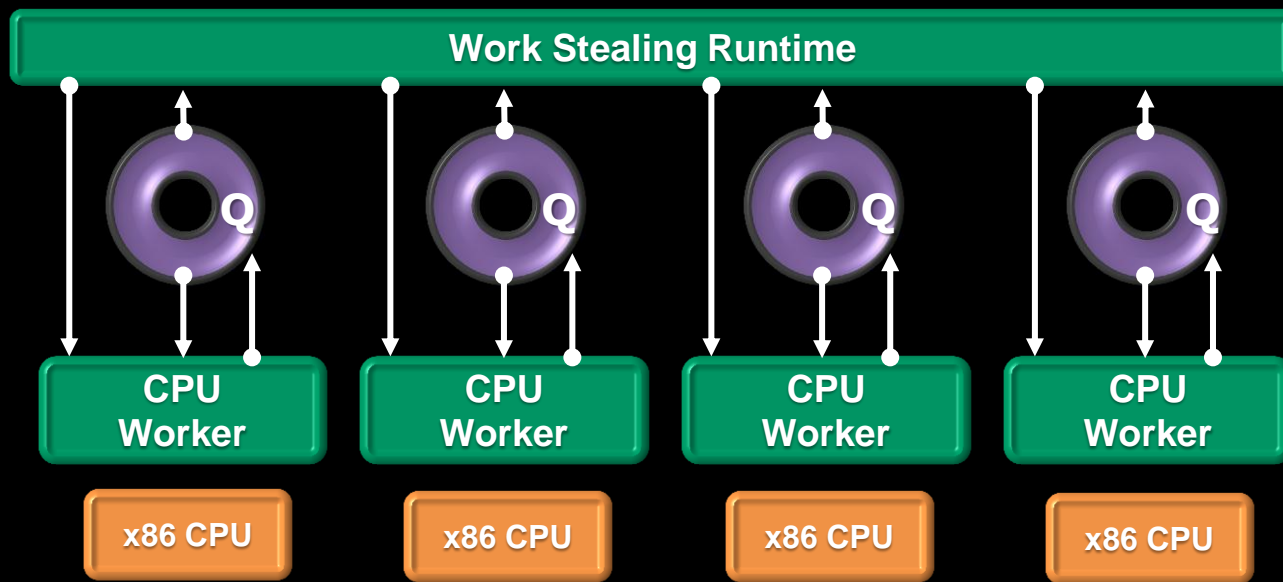
*sequential consistent ordering



HSA ENABLES 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
- HSA is designed to extend this pattern to run on heterogeneous systems

TASK QUEUING RUNTIME ON CPUS



CPU Threads

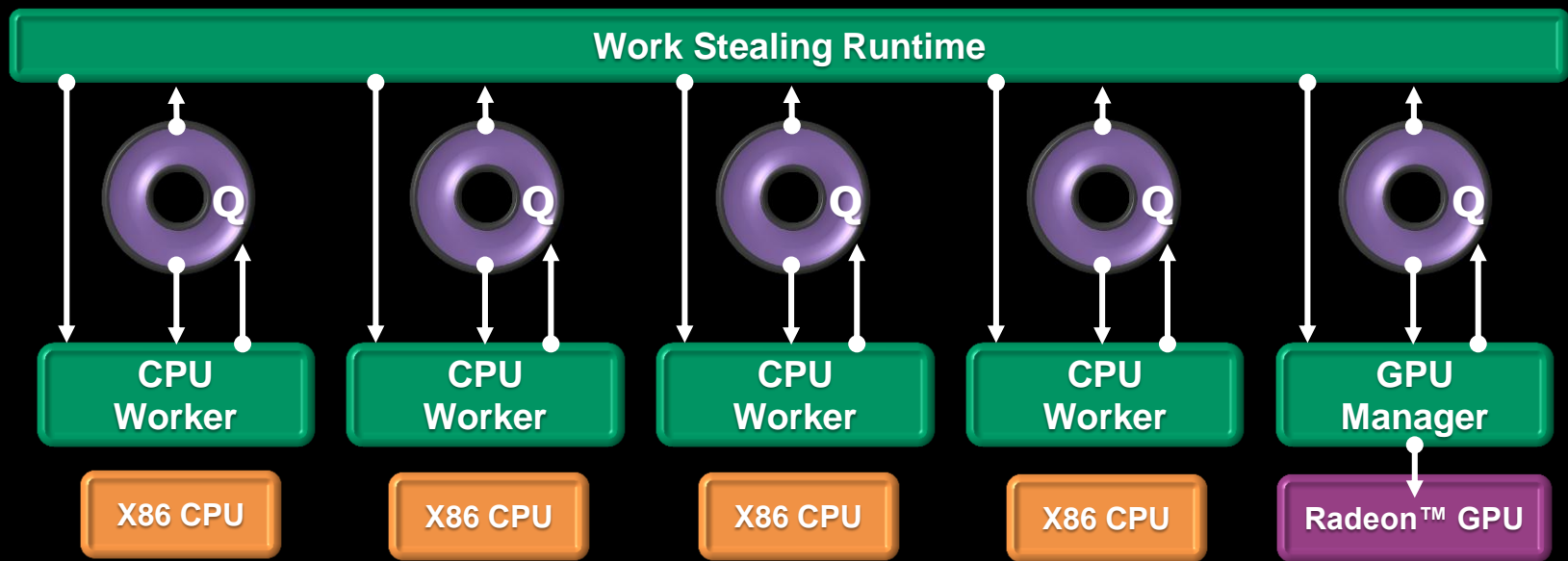


GPU Threads




Memory

TASK QUEUING RUNTIME ON THE FSA PLATFORM

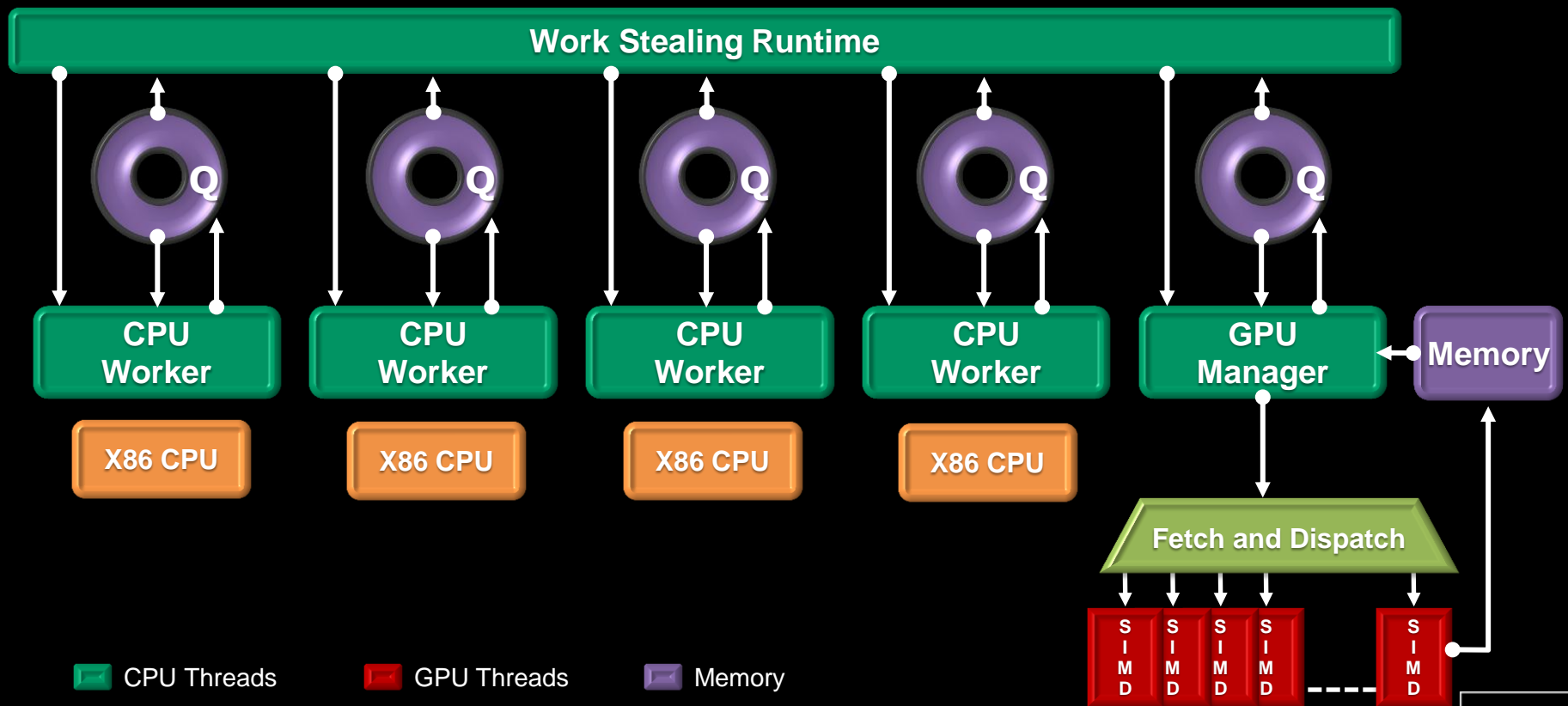


 CPU Threads

 GPU Threads

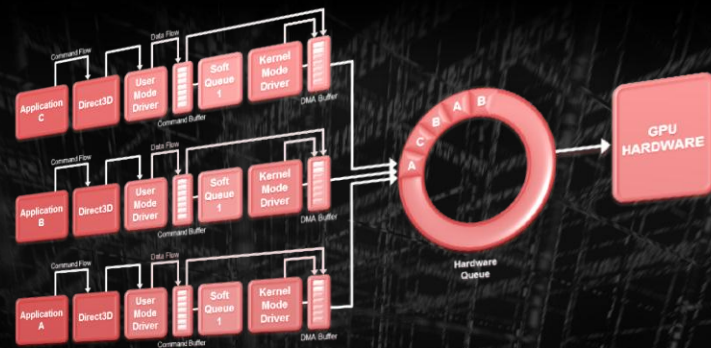
 Memory

TASK QUEUING RUNTIME ON THE FSA PLATFORM

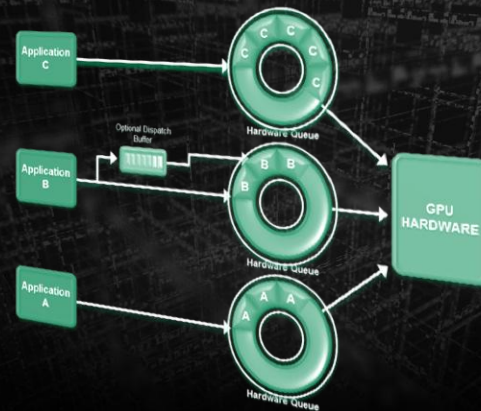


HETEROGENEOUS COMPUTE DISPATCH

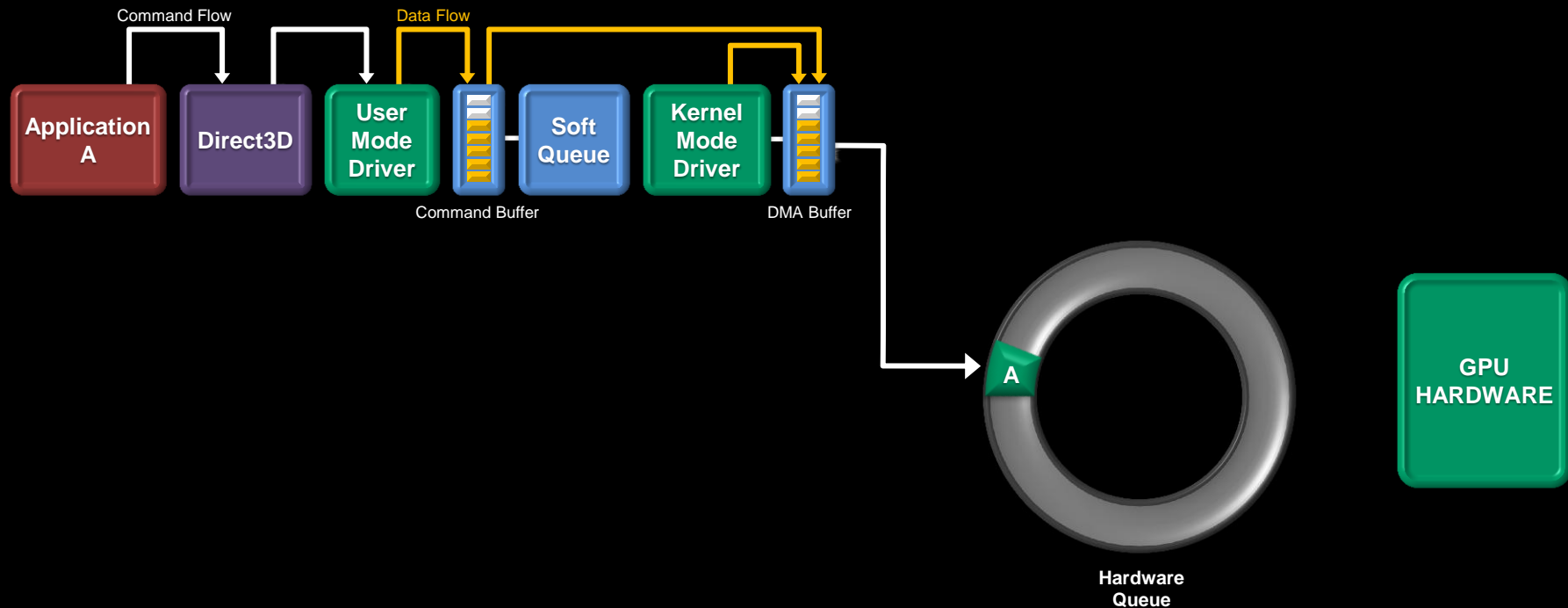
*How compute dispatch operates today in the **driver model***



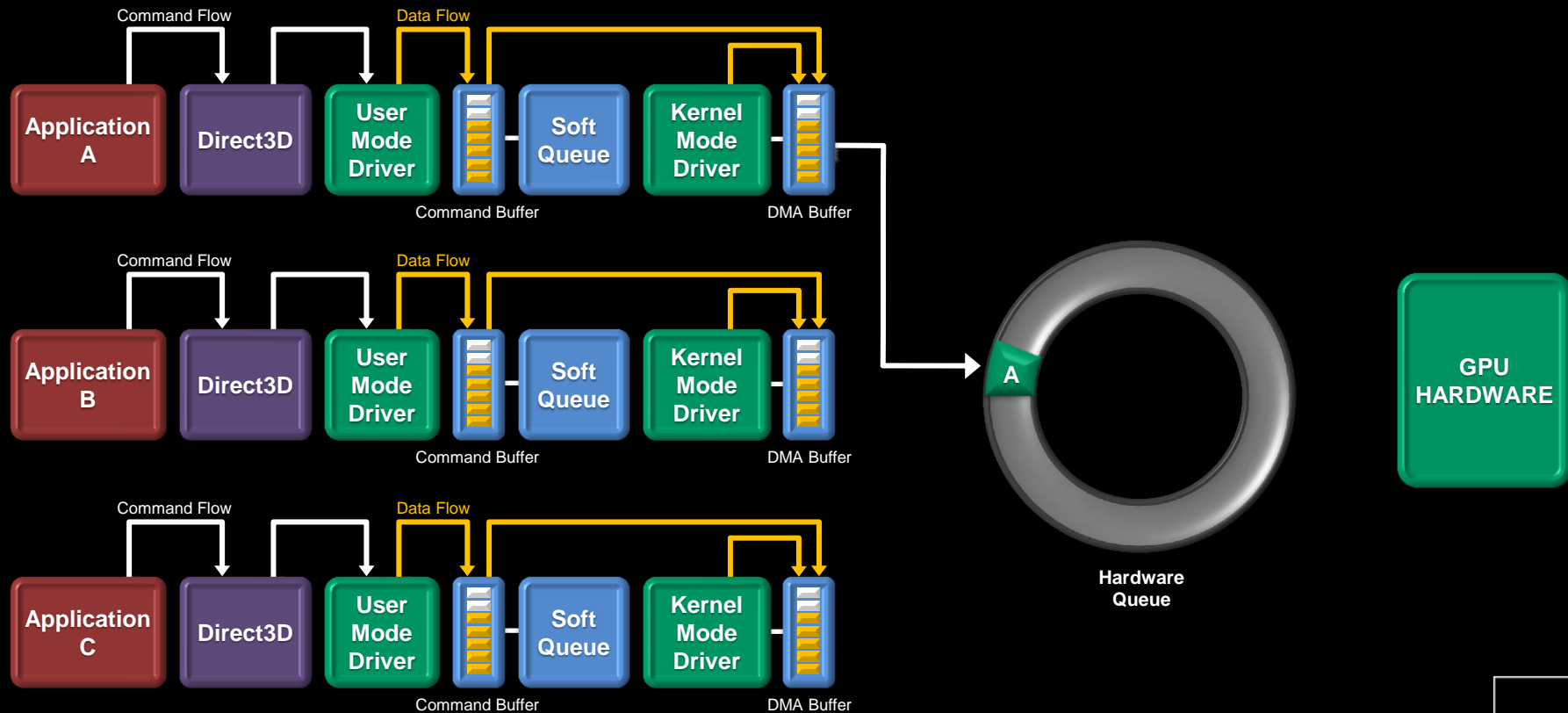
*How compute dispatch improves tomorrow under **HSA***



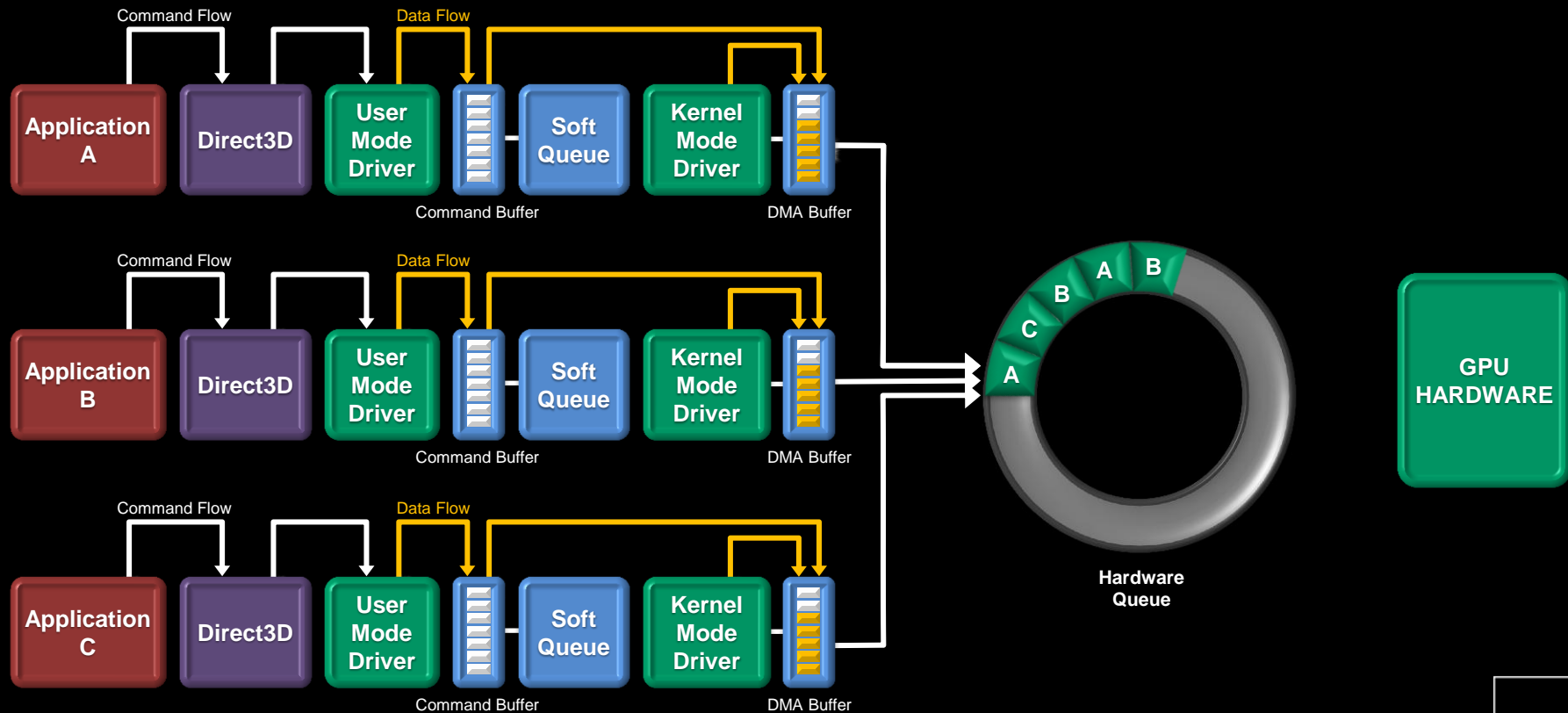
TODAY'S COMMAND AND DISPATCH FLOW



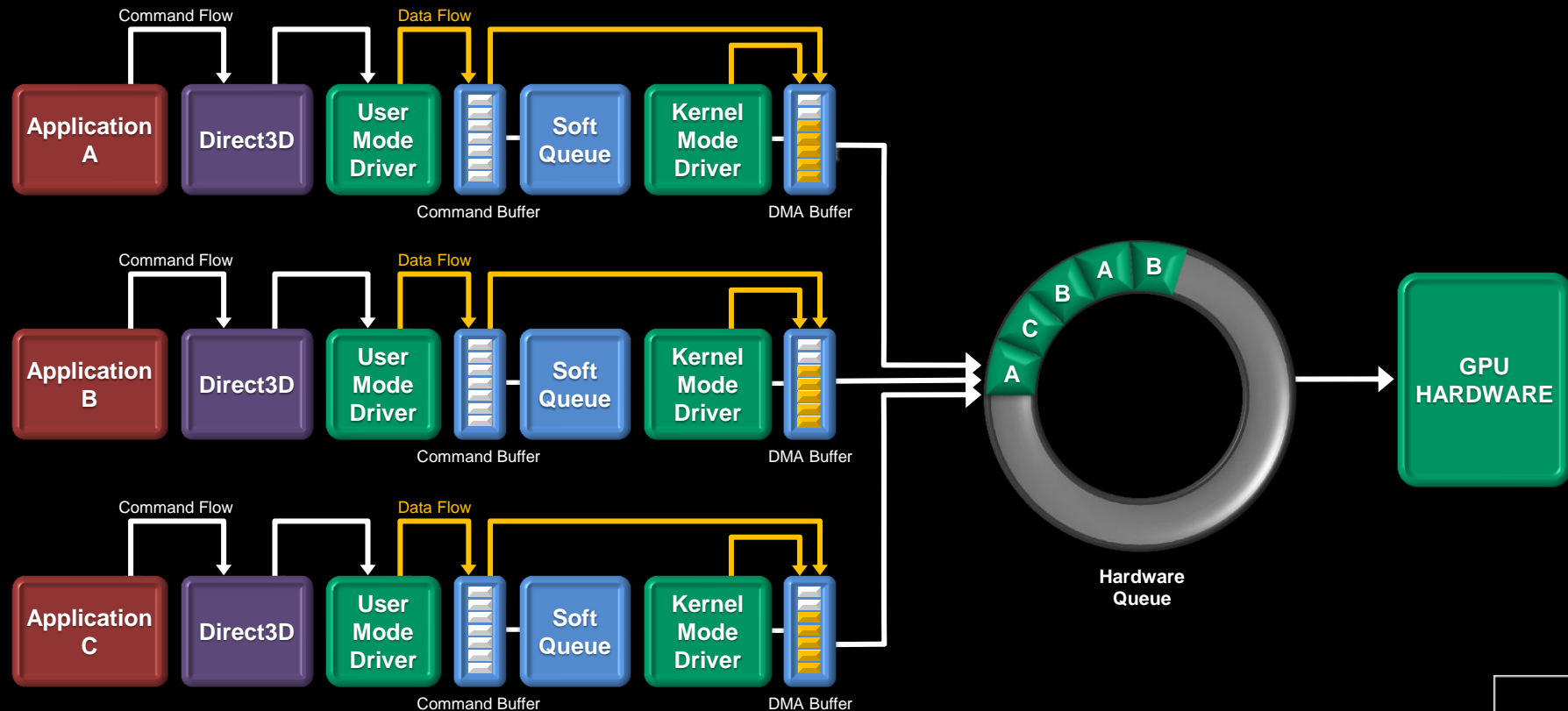
TODAY'S COMMAND AND DISPATCH FLOW



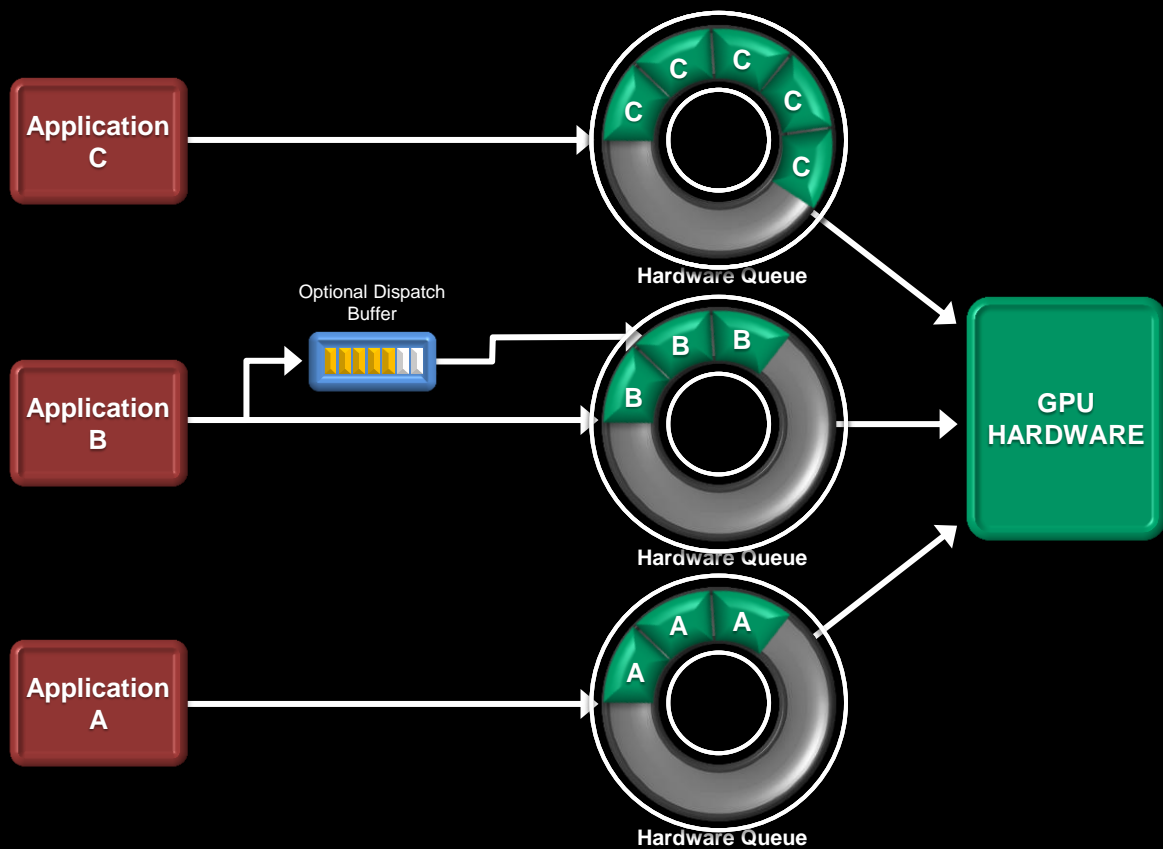
TODAY'S COMMAND AND DISPATCH FLOW



TODAY'S COMMAND AND DISPATCH FLOW



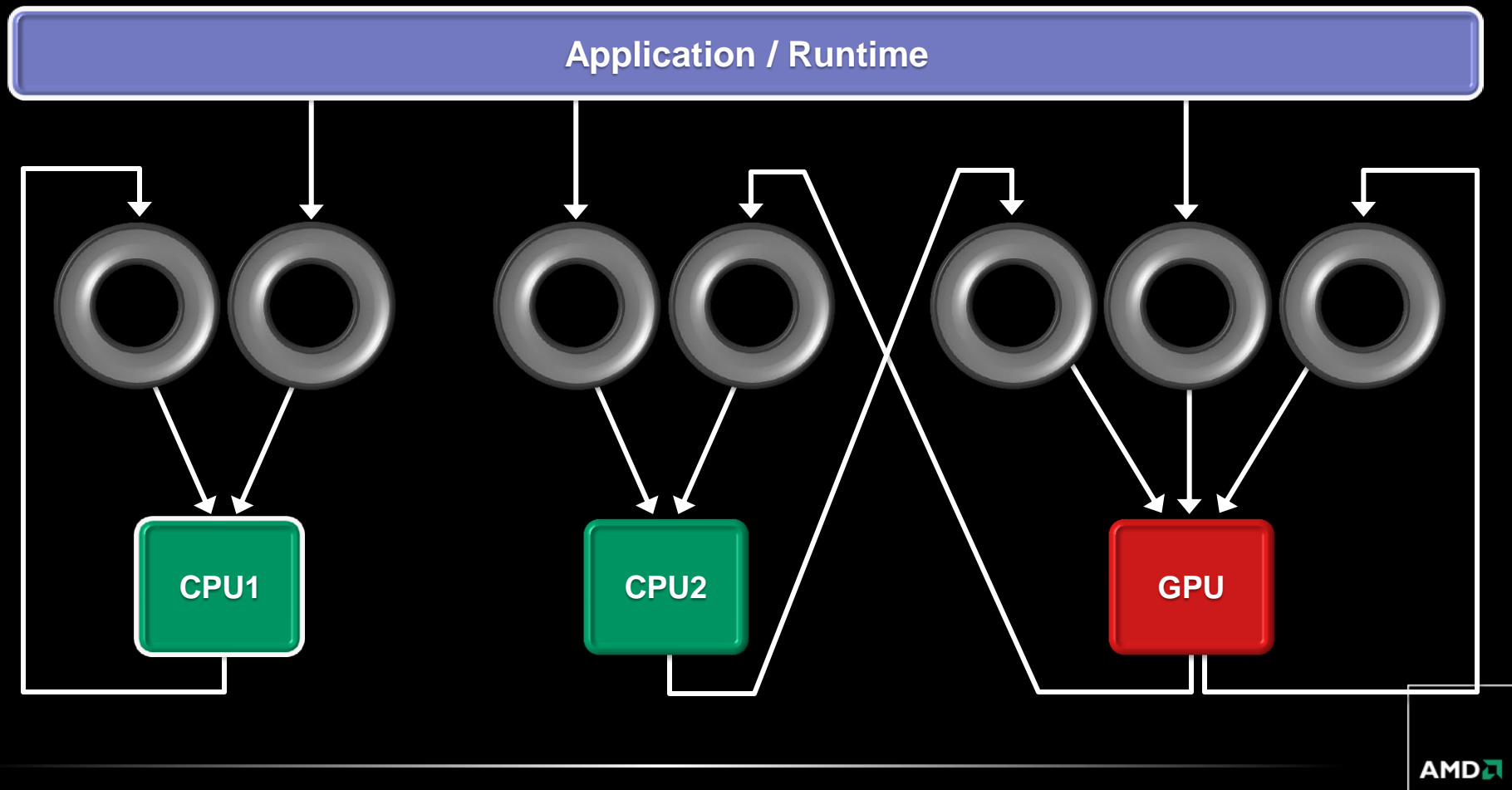
FUTURE COMMAND AND DISPATCH FLOW



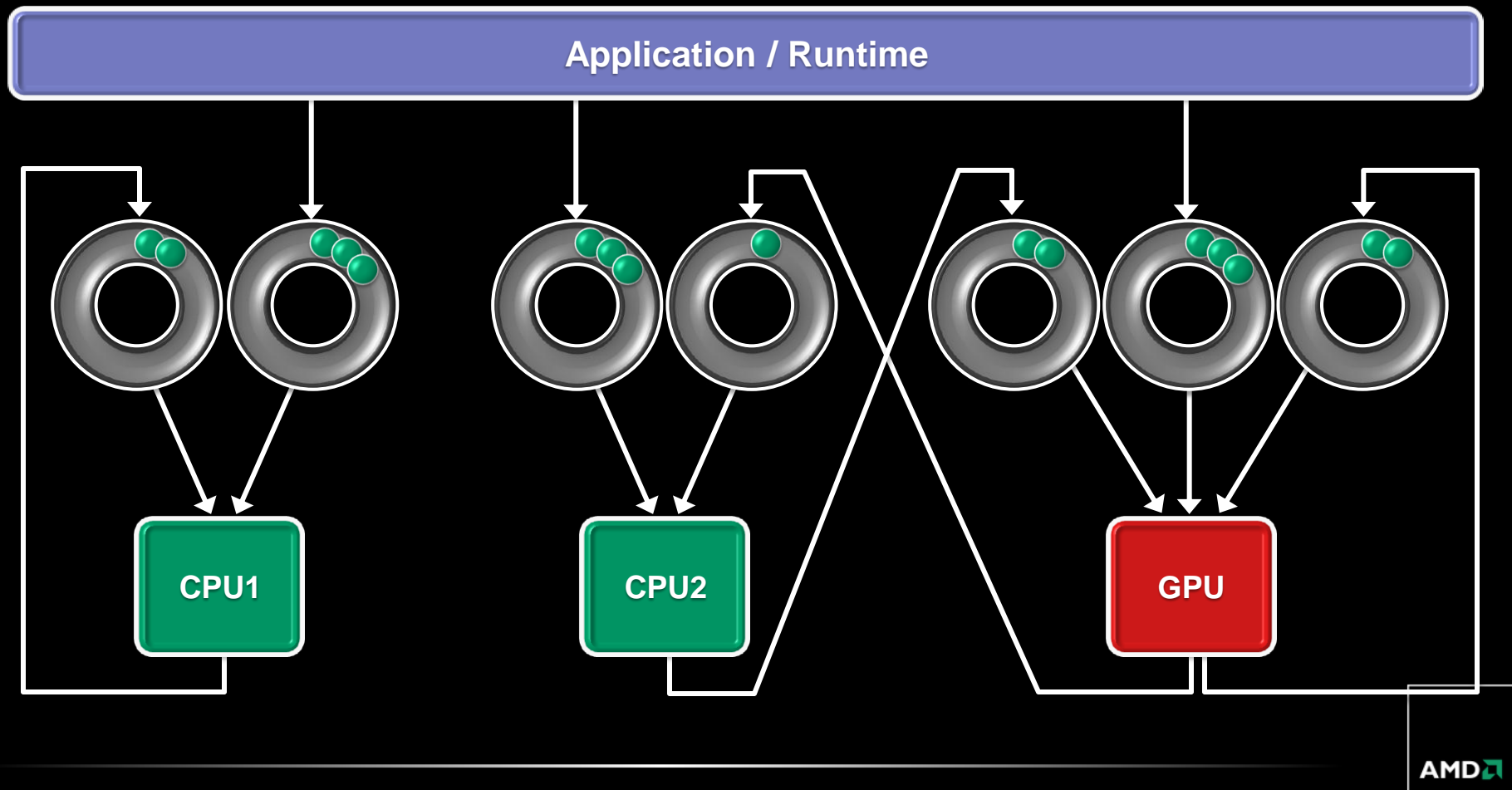
- Application codes to the hardware
- User mode queuing
- Hardware scheduling
- Low dispatch times

- No APIs (required)
- No soft queues
- No user mode drivers
- No kernel mode transitions
- Far less overhead!

FUTURE COMMAND AND DISPATCH CPU <-> GPU

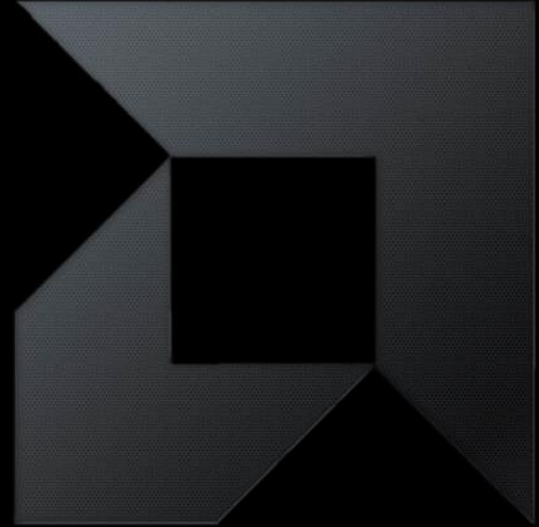


FUTURE COMMAND AND DISPATCH CPU <-> GPU



IMPROVING THE PROGRAMMING MODEL

Baby steps... the host API and kernel language



VECTOR ADDITION - HOST PROGRAM

```
// create the OpenCL context on a GPU device
```

```
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
```

```
// get the list of GPU devices associated with context
```

```
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
```

```
devices = malloc(cb);
```

```
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
```

```
// create a command-queue
```

```
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
```

```
// allocate the buffer memory objects
```

```
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
```

```
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB, NULL);
```

```
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);
```

```
// create the program
```

```
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
```



VECTOR ADDITION - HOST PROGRAM

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                    NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2], sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```



VECTOR ADDITION - HOST PROGRAM

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL,
    NULL, NULL);
```

```
// get the list of GPU devices associated with context
```

```
cl_
Define platform and queues
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
```

```
// create a command-queue
```

```
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
```

```
// allocate the buffer memory objects
```

```
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY,
    sizeof(cl_float)*n, NULL, NULL);
memobj
C:
Define memory objects
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(cl_float)*n, NULL, NULL);
```

```
// create the program
```

```
program =
    NULL)
Create the program
source, NULL,
```

Build the program

Create and setup kernel

Execute the kernel

Read results on the host

It's complicated, but most of this is “boilerplate” and not as bad as it looks.

HERE'S THE SAME PROGRAM IN CUDA

```
int main(int argc, char** argv) {  
    int N = 50000;  
    size_t size = N * sizeof(float);  
  
    h_A = (float*)malloc(size);  
    h_B = (float*)malloc(size);  
    h_C = (float*)malloc(size);  
  
    RandomInit(h_A, N);  
    RandomInit(h_B, N);  
  
    cudaMalloc((void**)&d_A, size);  
    cudaMalloc((void**)&d_B, size);  
    cudaMalloc((void**)&d_C, size);
```

```
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);  
  
    int blocksPerGrid = (N + 256 - 1) / 256;  
    vecAdd<<<blocksPerGrid, 256>>>( d_A, d_B, d_C, N);  
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```



CUDA EXAMPLE

- Clearly simpler than the OpenCL API code!
 - However, still a based C API that is not type safe!
 - Requires a non-standard host compiler, not just a device compiler!
- Can we not do something different?

CUDA EXAMPLE

- Clearly simpler than the OpenCL API code!
 - However, still a based C API that is not type safe!
 - Requires a non-standard host compiler, not just a device compiler!
- Can we not do something different?
- In the words of my colleague:
 - “Don’t use C it is a STUPID language, designed in the dark ages of computing!”

CUDA EXAMPLE

- Clearly simpler than the OpenCL API code!
 - However, still a based C API that is not type safe!
 - Requires a non-standard host compiler, not just a device compiler!
- Can we not do something different?
- In the words of my colleague:
 - “Don’t use C it is a STUPID language, designed in the dark ages of computing!”
- In the context, I think he has a point...

CUDA EXAMPLE

- Clearly simpler than the OpenCL API code!
 - However, still a based C API that is not type safe!
 - Requires a non-standard host compiler, not just a device compiler!
- Can we not do something different?
- In the words of my colleague:
 - “Don’t use C it is a STUPID language, designed in the dark ages of computing!”
- In the context, I think he has a point...
- So how about C++?

THE C++ INTERFACE

- Khronos has defined a common C++ header file containing a high level interface to OpenCL.
- Key features:
 - Uses common defaults for the platform and command-queue ... saving the programmer from extra coding for the most common use cases.
 - Simplifies basic API by bundling key parameters with the objects rather than verbose and repetitive argument lists.
 - Reference counting
 - Kernel functors
 - Statically checked information routines

C++ KERNELS

- OpenCL C++ Kernel language
 - Develop device code using full C++
 - Extending OpenCL address spaces in the C++ type system
 - Automatic inference of address spaces, handle this pointer address space deduction
- Combine C++ Host API and C++ Kernel language:
 - Shared pointer representation, allow support for pointer based data-structures shared between CPU and GPU



C++ INTERFACE: SETTING UP THE HOST PROGRAM

Single header files ... both standard

```
#include <CL/cl.hpp>          // Khronos C++ Wrapper API
```

Setup the functional interface (used by the C++ OpenCL API)

```
#include <functional>
```

Everything defined with single namespace:

```
using namespace cl;
```



POINTER AND FUNCTOR EXAMPLE

```
cl::Pointer<int> x = cl::malloc<int>(N);  
for (int i = 0; i < N; i++) { *(x+i)= rand(); }
```

```
std::function<Event (const cl::EnqueueArgs&, cl::Pointer<int>) plus =  
    make_kernel<cl::Pointer<int>, int>(  
        "kernel void plus(global Pointer<int> io)  
        {  
            int I = get_global_id(0);  
            *(io+i) = *(io+i) * 2;  
        }");  
plus(EnqueueArgs(NDRange(N)), x);  
  
for(int i = 0; i < N; i++) { cout << *(x+i) << endl; }
```



POINTER AND FUNCTOR EXAMPLE

```
cl::Pointer<int> x = cl::malloc<int>(N);  
for (int i = 0; i < N; i++) { *(x+i)= rand(); }
```

```
std::function<Event (const cl::EnqueueArgs&, cl::Pointer<int>) plus =  
    make_kernel<cl::Pointer<int>, int>(  
        "kernel void plus(global Pointer<int> io)  
        {  
            int I = get_global_id(0);  
            *(io+i) = *(io+i) * 2;  
        }");  
plus(EnqueueArgs(NDRange(N)), x);
```

- **Defaults - no need to reference context, command queue.**
- **Program automatically created and compiled too.**

```
for(int i = 0; i < N; i++) { cout << *(x+i) << endl; }
```

OPENCL KERNEL LANGUAGE

```
template<address-space aspace_>
struct Shape {
    int foo(aspace_ Colour&) global + local;
    int foo(aspace_ Colour&) private;
    int bar(void);
};

operator (const decltype(this)& rhs) -> decltype(this)&
{
    if (this == &rhs) { return *this; }
    ...
    return *this;
}
```



OPENCL KERNEL LANGUAGE

```
template<address-space aspace_>  
struct Shape {  
    int foo(aspace_ Colour&) global + local;  
    int foo(aspace_ Colour&) private;  
    int bar(void);  
};
```

- Abstract over address space qualifiers.
- Methods can be annotated with address spaces, controls “this” pointer location. Extended to overloading.
- Default address space for “this” is deduced automatically. Support default constructors.

```
operator (const decltype(this)& rhs) -> decltype(this)&  
{  
    if (this == &rhs) { return *this; }  
    ...  
    return *this;  
}
```

OPENCL KERNEL LANGUAGE

```
template<address-space aspace_>
struct Shape {
    int foo(aspace_ Colour&) global + local;
    int foo(aspace_ Colour&) private;
    int bar(void);
};
```

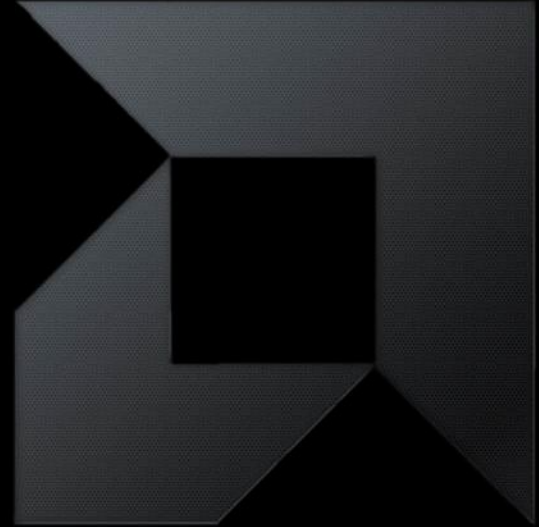
- Abstract over address space qualifiers.
- Methods can be annotated with address spaces, controls “this” pointer location. Extended to overloading.
- Default address space for “this” is deduced automatically. Support default constructors.

```
operator (const decltype(this)& rhs) -> decltype(this)&
{
    if (this == &rhs) { return *this; }
    ...
    return *this;
}
```

- C+11 features used to handle cases when type of “this” needs to be written down by the developer.

IMPROVING THE PROGRAMMING MODEL

The future: fixing the composition problem



COMPOSABILITY IN CURRENT MODELS

- Current GPU programming models suffer from composability limitations
- The data-parallel model works in simple cases. Its fixed, limited nature breaks down when:
 - We need to use long-running threads to more efficiently perform reductions
 - We want to synchronize inside and outside library calls
 - We want to pass memory spaces into libraries
- Among others...

MEMORY ADDRESS SPACES ARE NOT COMPOSABLE

```
void foo(global int *)  
{  
    ...  
}
```

```
void bar(global int * x)  
{  
    foo(x); // works fine  
  
    local int a[1];  
    a[0] = *x;  
    foo(a); // will now not work  
}
```

BARRIER ELISION IS NOT COMPOSABLE

- Parallel prefix sum (in this case taken from a RadixSort)

for level = 0 to n

 foreach work item i in buffer range

 if($i > 2^{\text{level}}$)

 temp = buffer[i- $2^{(n-1)}$] + buffer[i];

 barrier(); // barrier is not needed within a wavefront or warp

 // adds overhead so often dropped for optimization reasons

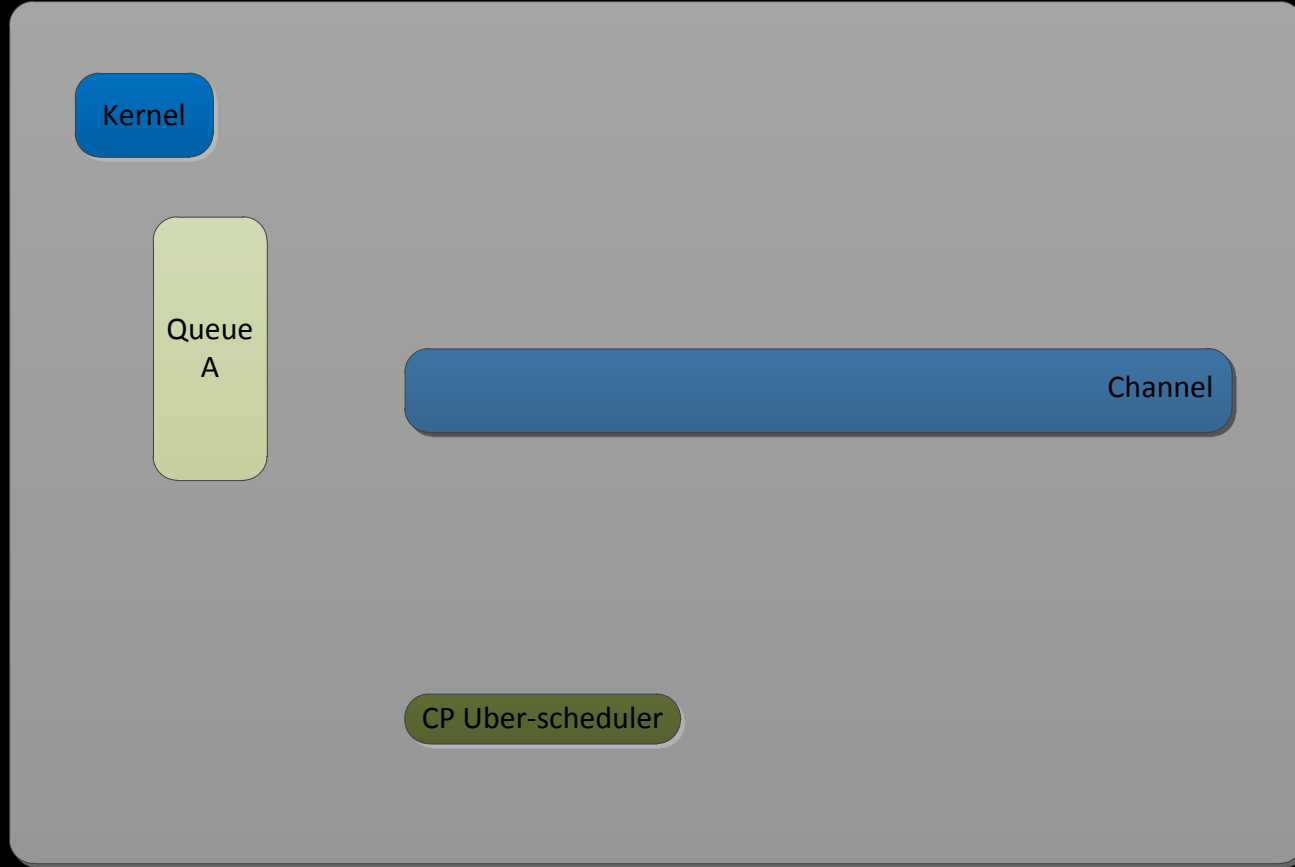
 buffer[i] = temp;

CHANNELS - PERSISTENT CONTROL PROCESSOR THREADING MODEL

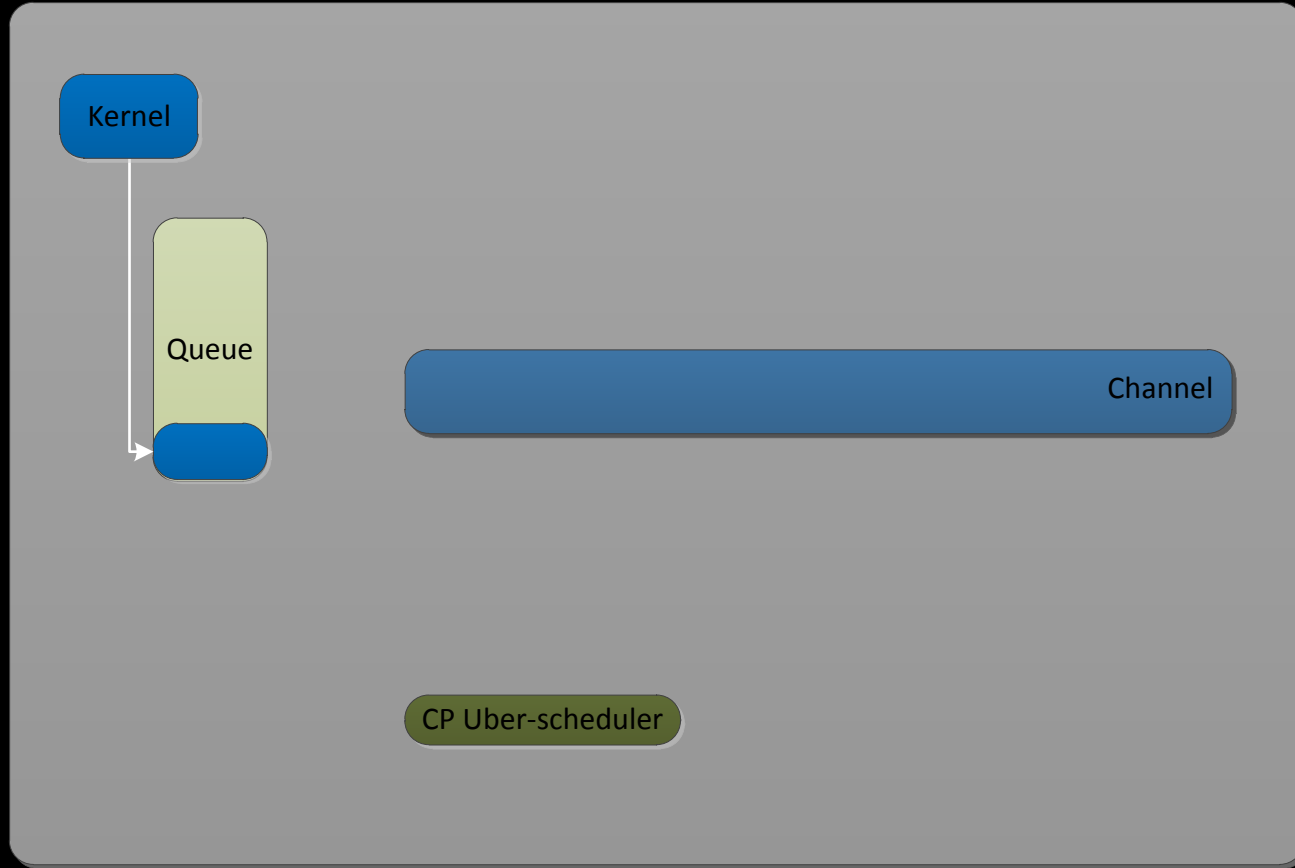
- Add data-flow support to GPGPU
- We are not primarily notating this as producer/consumer kernel bodies
 - That is that we are not promoting a method where one kernel loops producing values and another loops to consume them
 - That has the negative behavior of promoting long-running kernels
 - We've tried to avoid this elsewhere by basing in-kernel launches around continuations rather than waiting on children
- Instead we assume that kernel entities produce/consume but consumer work-items are launched on-demand
- An alternative to the point to point data flow using of persistent threads, avoiding the uber-kernel



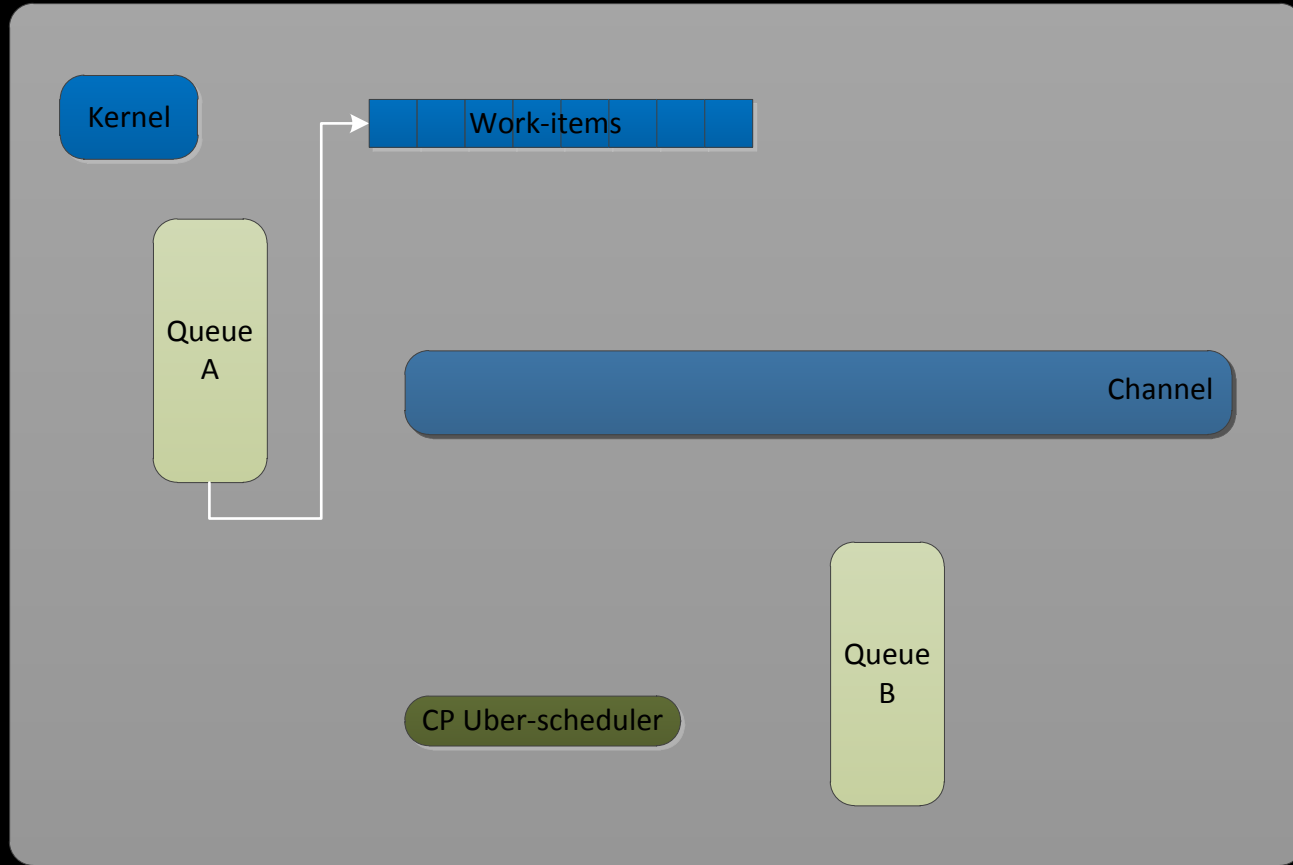
OPERATIONAL FLOW



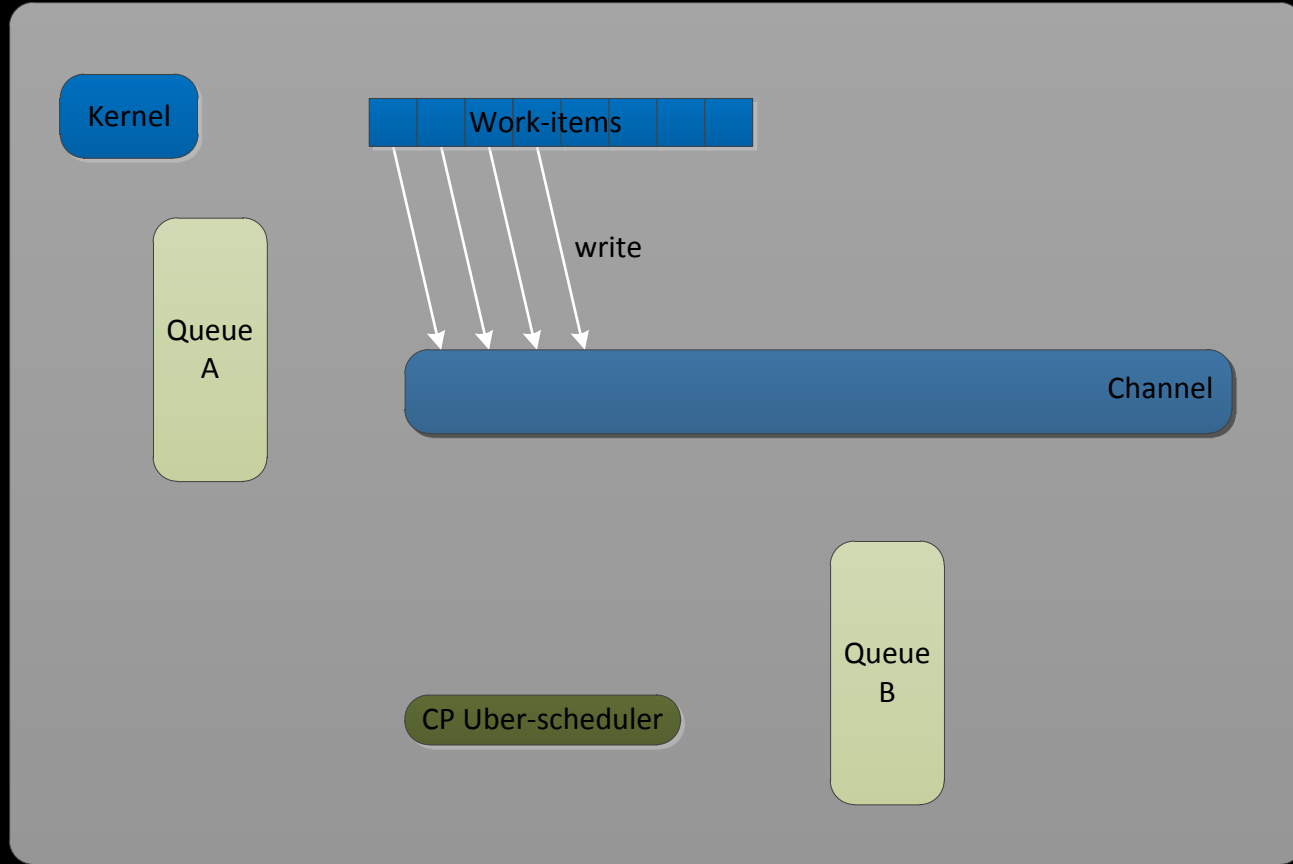
OPERATIONAL FLOW



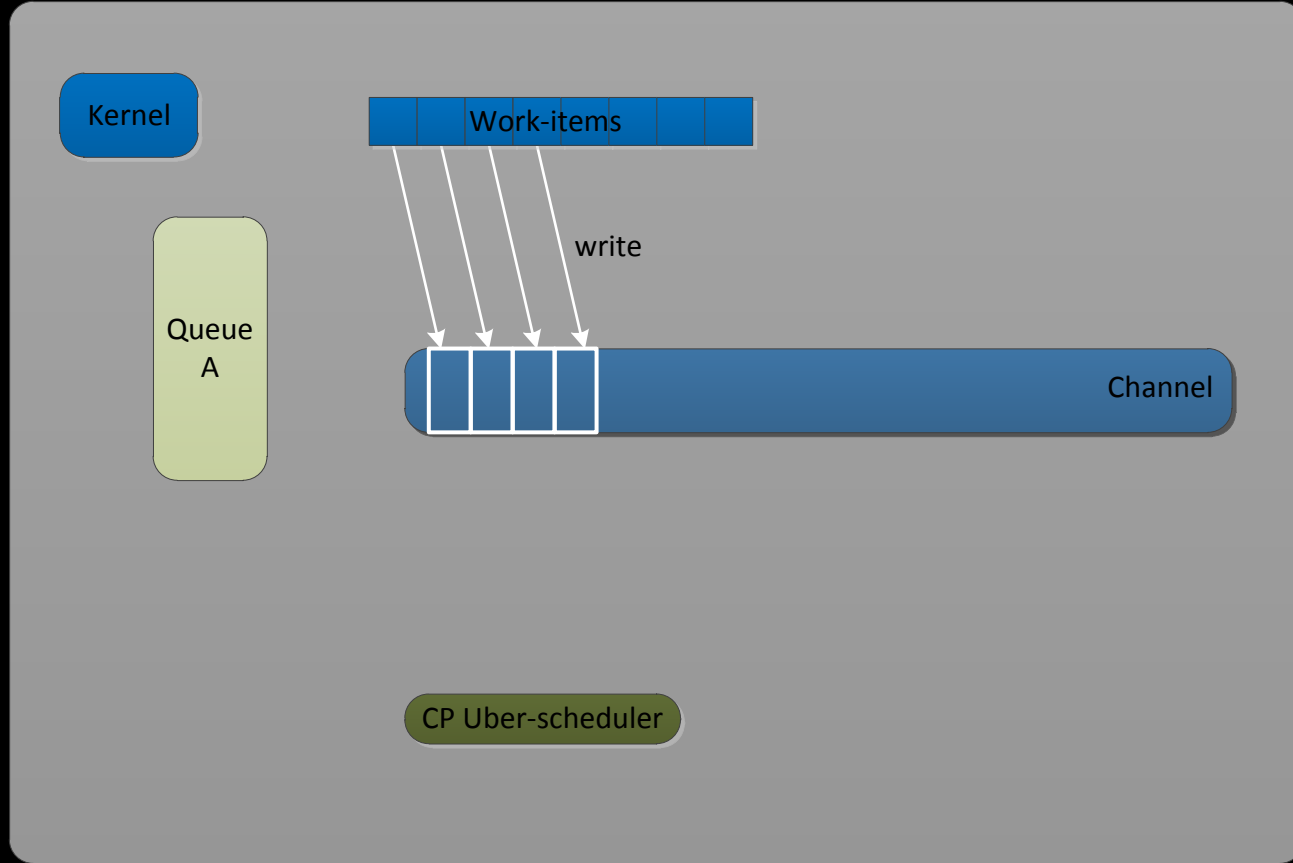
OPERATIONAL FLOW



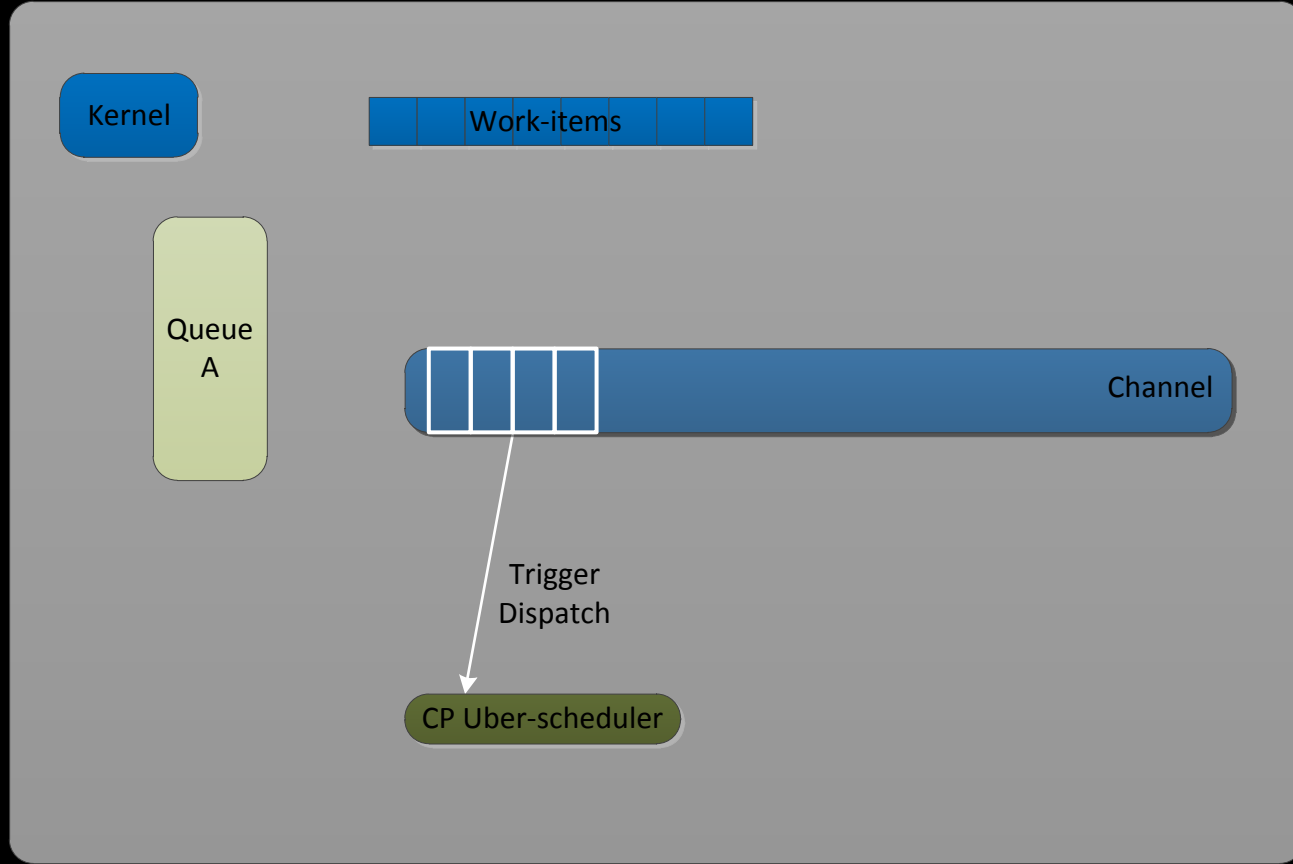
OPERATIONAL FLOW



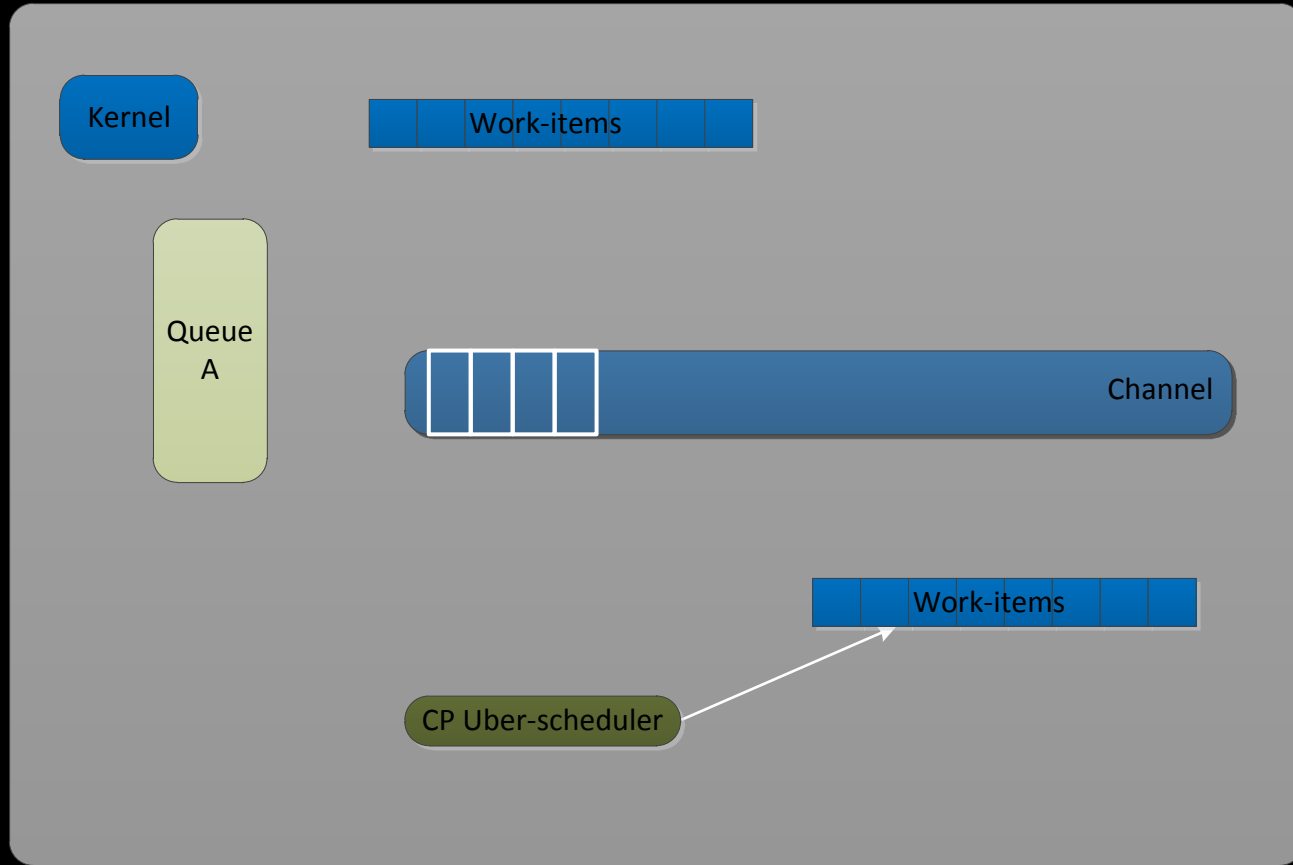
OPERATIONAL FLOW



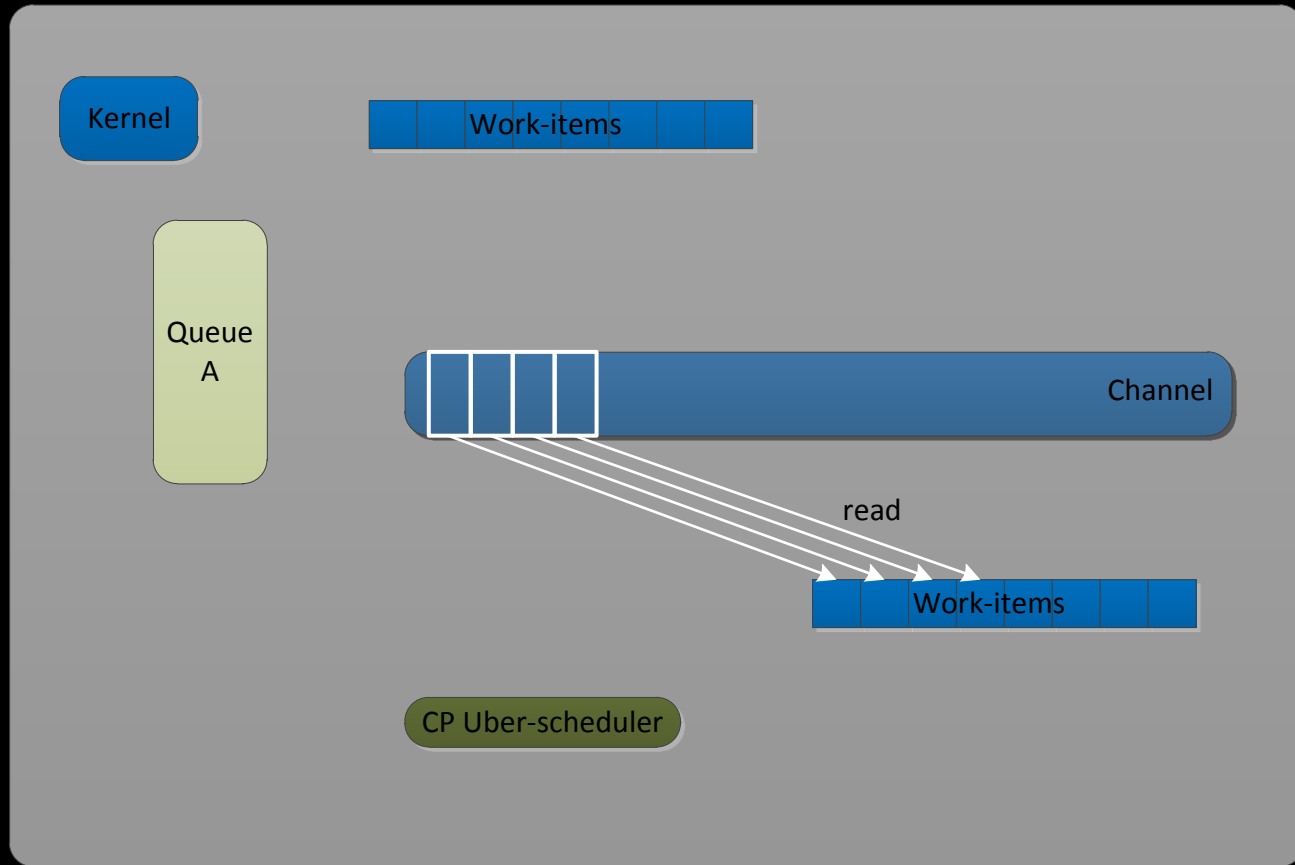
OPERATIONAL FLOW



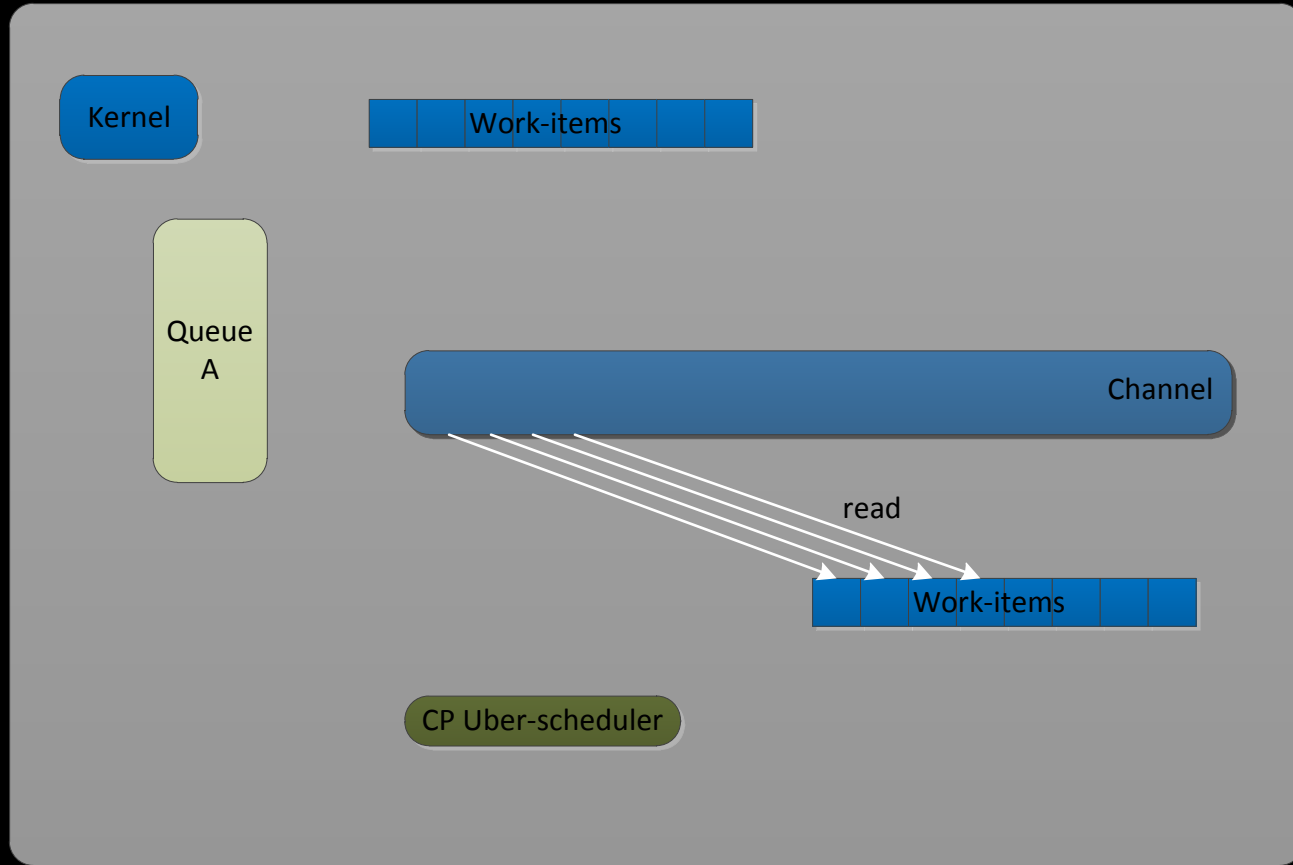
OPERATIONAL FLOW



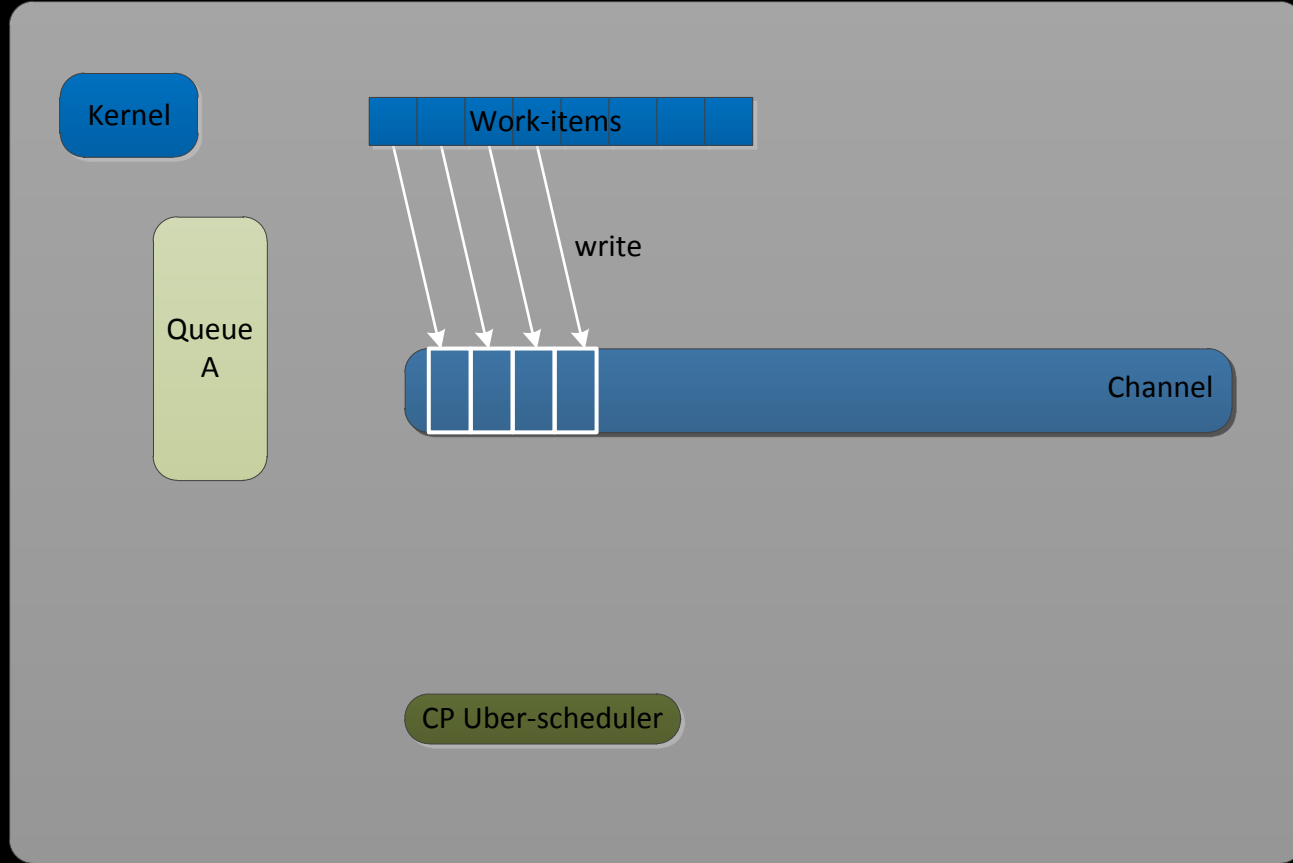
OPERATIONAL FLOW



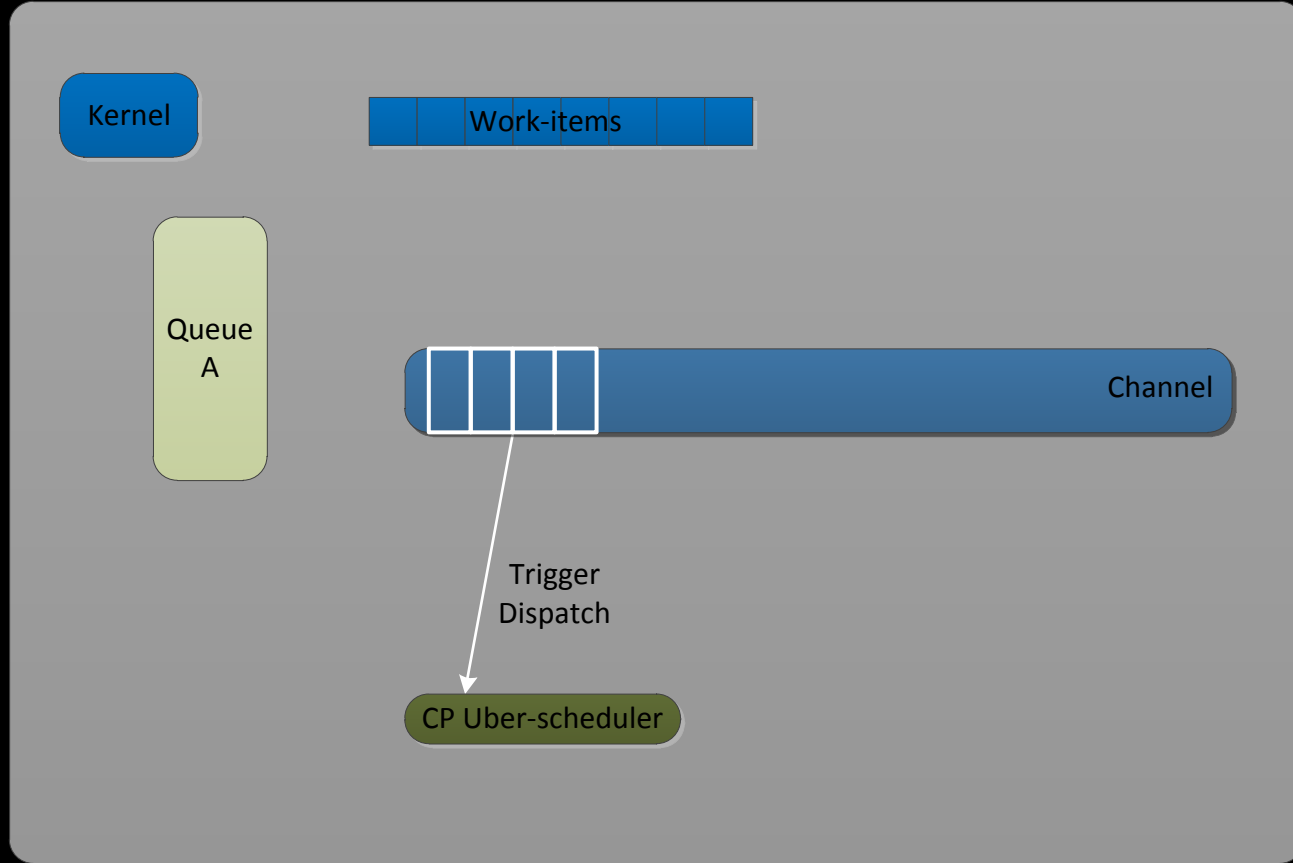
OPERATIONAL FLOW



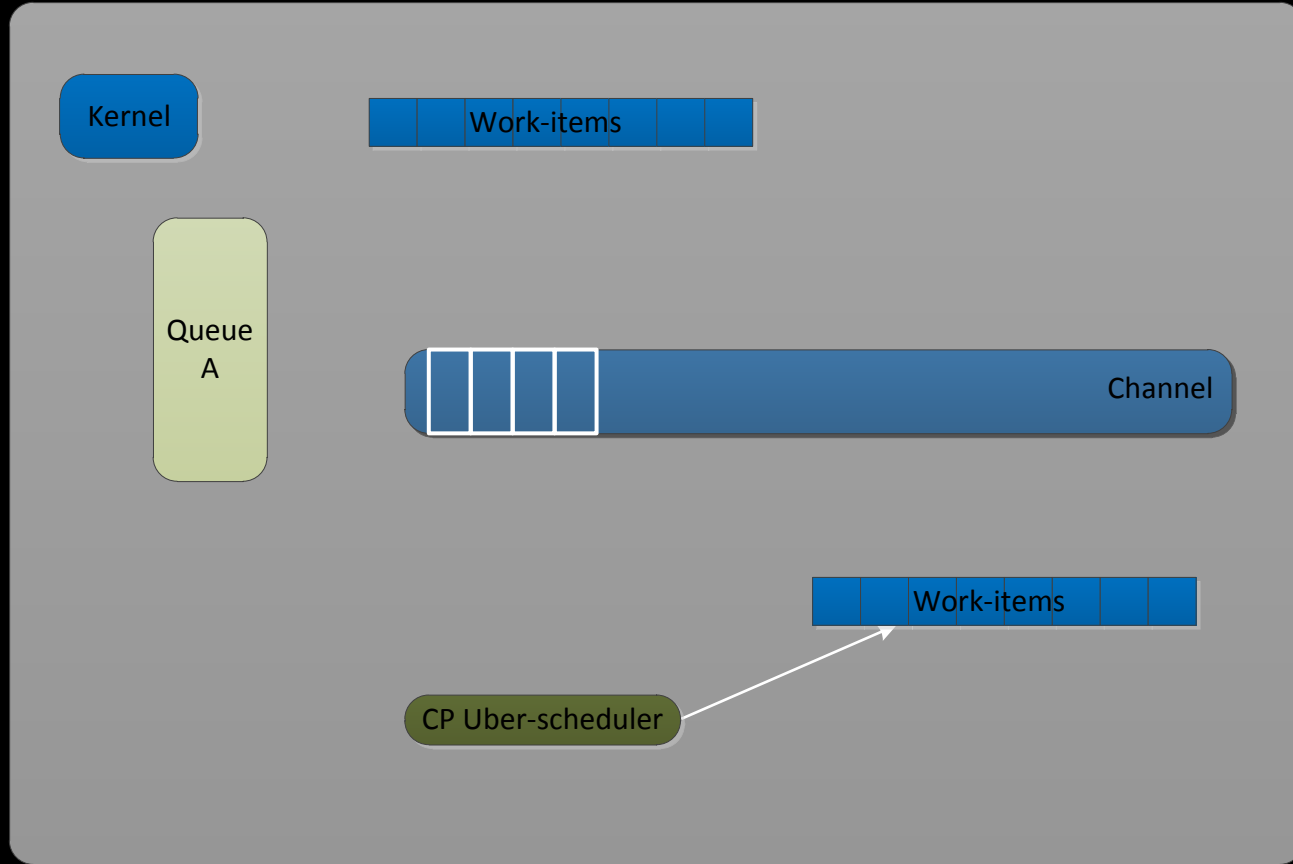
OPERATIONAL FLOW



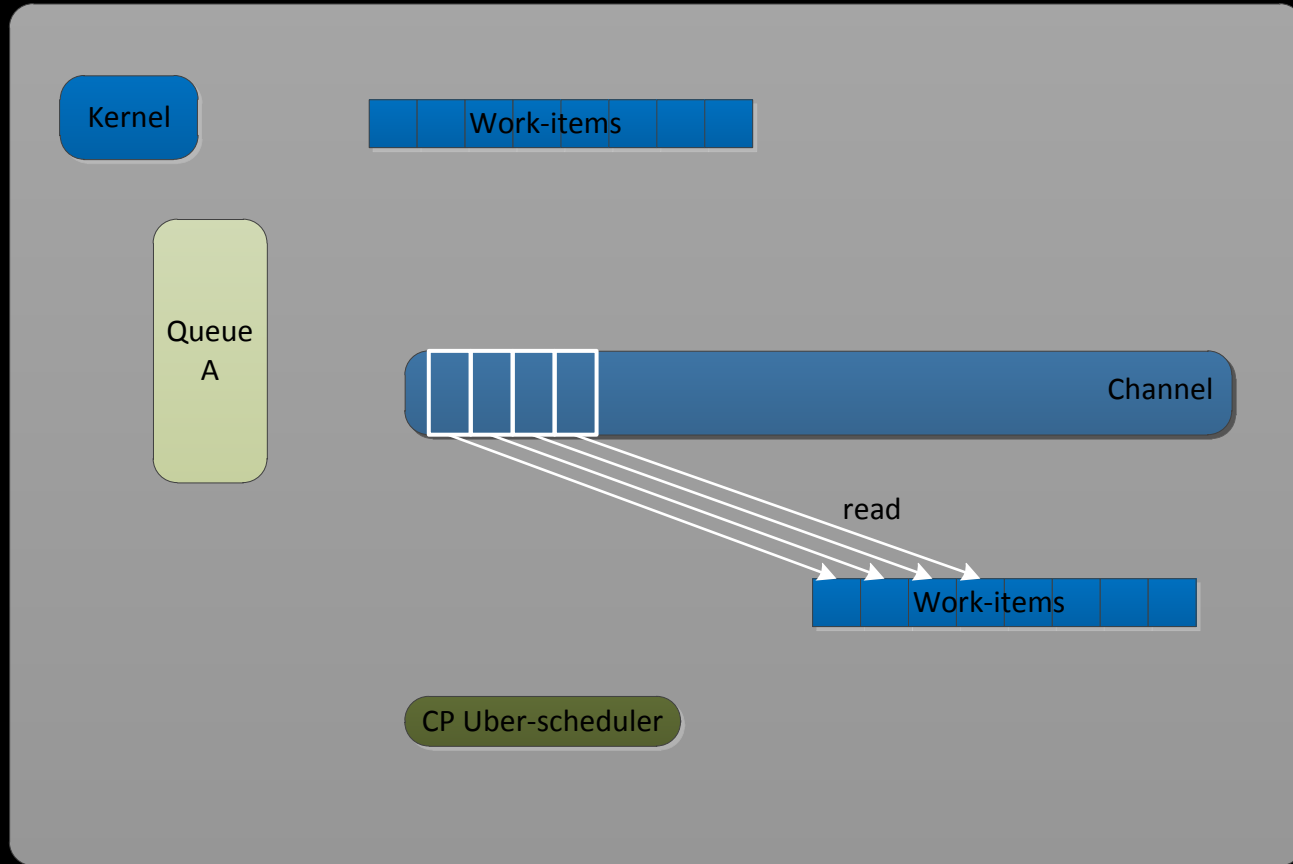
OPERATIONAL FLOW



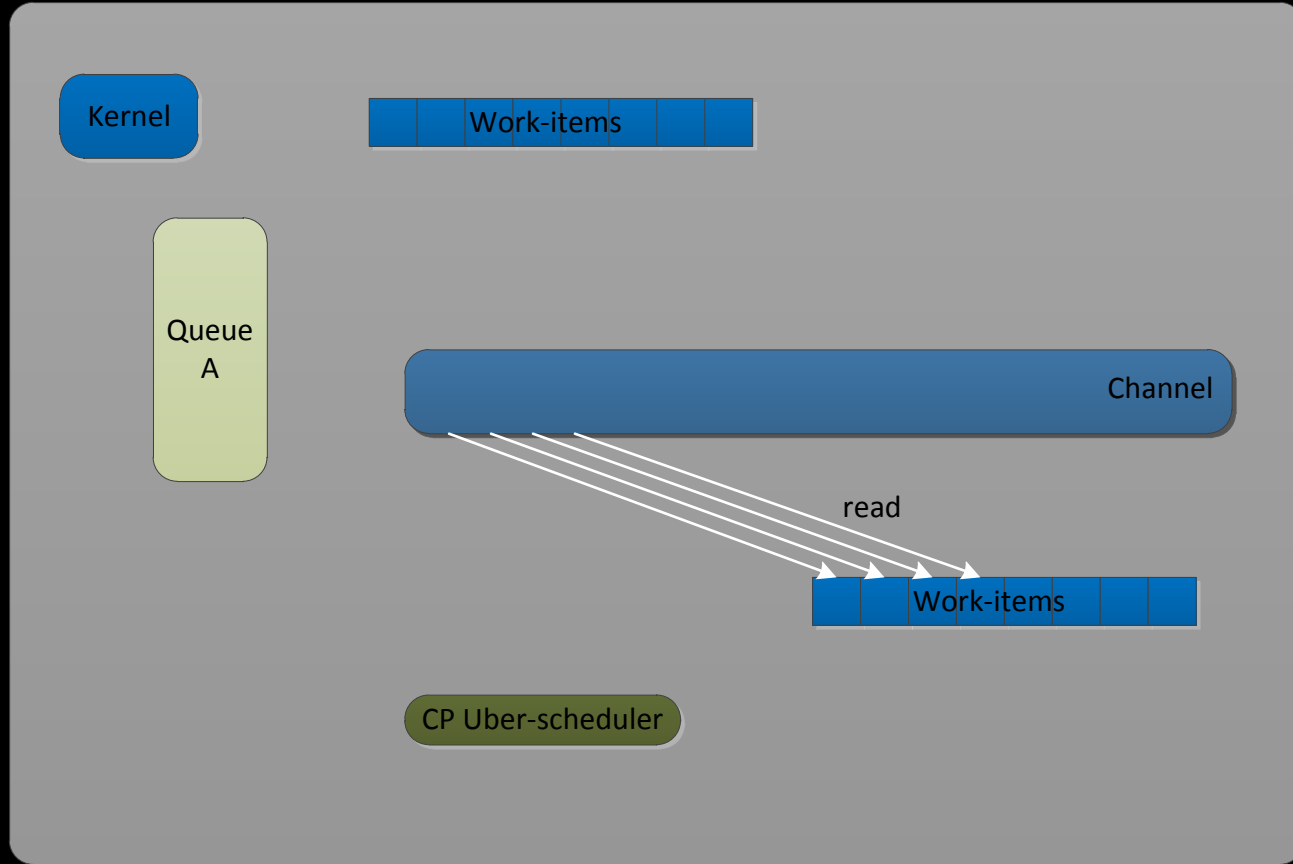
OPERATIONAL FLOW



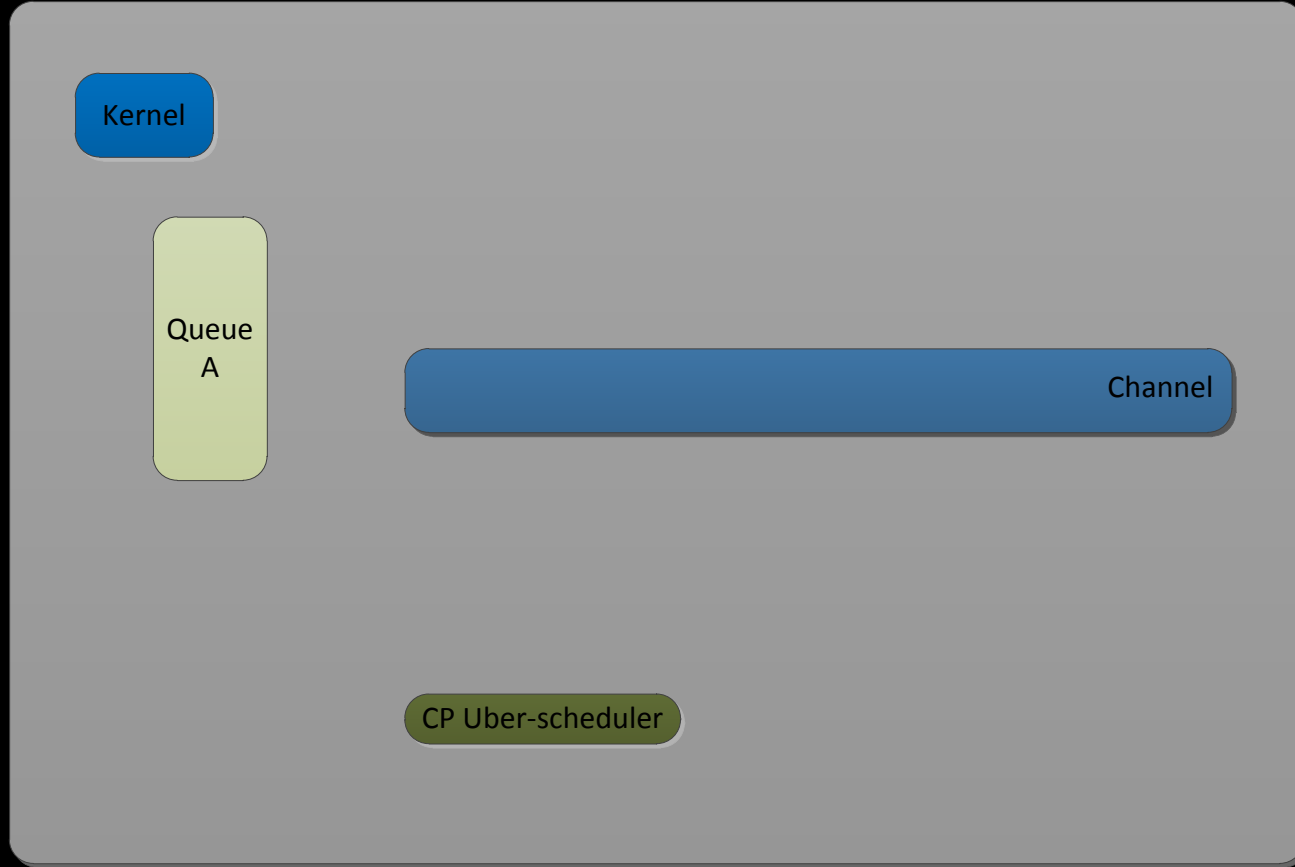
OPERATIONAL FLOW



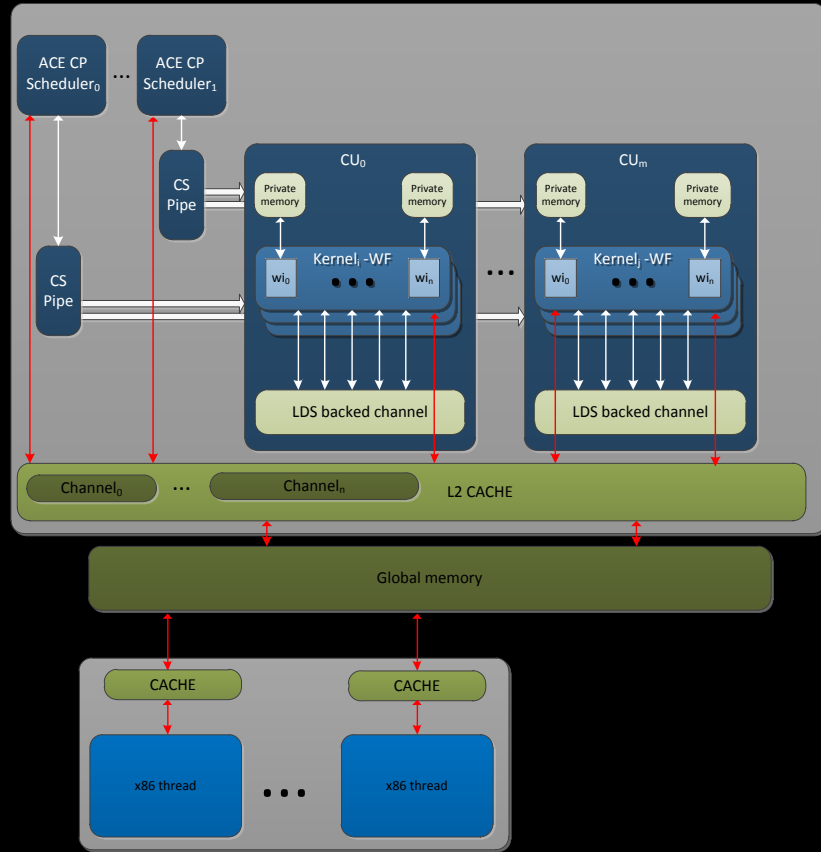
OPERATIONAL FLOW



OPERATIONAL FLOW



PERSISTENT CONTROL PROCESSOR THREADING MODEL



CHANNEL EXAMPLE

```
std::function<bool (opp::Channel<int>*)> predicate =  
    [] (opp::Channel<int>* c) -> bool __device(fql) {  
        return c->size() % PACKET_SIZE == 0;  
    };
```

```
opp::Channel<int> b(N);
```

```
b.executeWith(  
    predicate,  
    opp::Range<1>(CHANNEL_SIZE),  
    [&sumB] (opp::Index<1>) __device(opp) {  
        sumB++;  
    });
```

```
opp::Channel<int> c(N);
```

```
c.executeWith(  
    predicate,  
    opp::Range<1>(CHANNEL_SIZE),  
    [&sumC] (opp::Index<1>, const int v) __device(opp) {  
        sumC += v;  
    });
```

```
opp::parallelFor(  
    opp::Range<1>(N),  
    [a, &b, &c] (opp::Index<1> index) __device(opp) {  
        unsigned int n = *(a+index.getX());  
        if (n > 5) {  
            b.write(n);  
        }  
        else {  
            c.write(n);  
        }  
    });
```


MAKING BARRIERS FIRST CLASS

- We looked at two approaches to solving the barrier composibility problem:
 - Implicit barriers, simply extend the current barrier
 - Problem with this approach is it has surprisingly limited application, for example think a set of waves producing data for another set of waves within the same wavefront. It is not possible to express this relationship in a way that allows the producer to progress for multiple clients and multiple data sets.
 - Barrier objects, introduce barriers as first class values with a set of well define operations:
 - Construction – initialize a barrier for some sub-set of work-items within a work-group or across work-groups
 - Arrive – work-item marks the barrier as satisfied
 - Skip – work-item marks the barrier as satisfied and note that it will no longer take part in barrier operations. Allows early exit from a loop or divergent control where work-item never intends to hit take part in barrier
 - Wait – work-item marks the barrier as satisfied and waits for all other work-items to arrive, wait, or skip.



BARRIER OBJECT EXAMPLE – SIMPLE DIVERGENT CONTROL FLOW

```
barrier b(8);  
parallelFor(Range<1>(8), [&b] (Index<1> i) {  
    int val = i.getX();  
    scratch[i] = val;  
    if( i < 4 ) {  
        b.wait();  
        x[i] = scratch[i+1];  
    } else {  
        b.skip();  
        x[i] = 17;  
    }  
});
```

BARRIER OBJECT EXAMPLE – LOOP FLOW

```
barrier b(8);
parallelFor(Range<1>(8) [&b] (Index<1> i) {
    scratch[i.getX()] = i.getX();
    if( i.getX() < 4 ) {
        for( int j = 0; j < i; ++j ) {
            b.wait();
            x[i] += scratch[i+1];
        }
        b.skip();
    } else {
        b.skip();
        x[i.getX()] = 17;
    }
});
```

BARRIER OBJECT EXAMPLE – CALLING A FUNCTION FROM WITHIN CONTROL FLOW

```
barrier b(8);  
parallelFor(Range<1>(8), [&b] (Index<1> i) {  
    scratch[i] = i.getX();  
    if( i.getX() < 4 ) {  
        someOpaqueLibraryFunction(i.getX(), b);  
    } else {  
        b.skip();  
        x[i.getX()] = 17;  
    }  
});
```

```
void someOpaqueLibraryFunction(  
    const int i, barrier &b)  
{  
    for( int j = 0; j < i; ++j ) {  
        b.wait();  
        x[j] += scratch[i+1];  
    }  
    b.skip();  
}
```

TO SUMMARIZE

- Architectures
 - Current CPU and GPU architectures and how they differ
 - How AMD is trying to improve the architectures to make programming easier and more flexible
- Programming models
 - The present state of the art for heterogeneous/GPU programming
 - Collaborative efforts to improve the models we use to develop for them
 - Future programming model enhancements to address the composition problem

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.

© 2011 Advanced Micro Devices, Inc.

