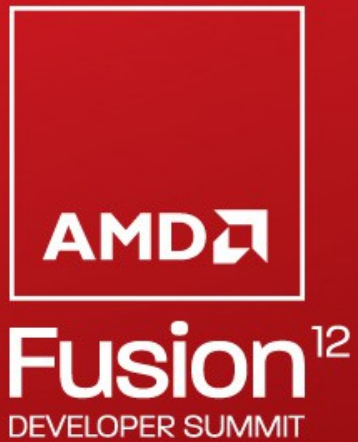
AMD

Fusion 12

DEVELOPER SUMMIT



Can gpgpu programming be liberated from the data-parallel bottleneck?

Lee Howes
AMD
Member of Technical Staff, Heterogeneous System Software

realism

Redirecting from mythology

last year…

- I talked about similarities between GPU and CPU

- AMD is pushing this story further
  - You saw the launch of the first graphics core next-based GPUs this year
  - The HSA devices, software infrastructure and associated specifications are progressing

- Today I want to get people thinking about reality
  - Get away from the mythology of GPU execution
  - Consider how we can streamline programming to make good use of hardware

First of all: GPU myths

- Most people outside of marketing view GPU architecture claims as slightly exaggerated
  - It **is** valid to say that a GPU has hundreds of cores
  - It is **not** valid to compare that definition of “core” with that of an x86 CPU
  - As such, if a GPU has hundreds of cores, a CPU does not have 8

- The same is true of threads
  - We **can** say that a thread is the programmer visible construct that we see in OpenCL™, CUDA, PTX and so on
  - That does not really mean that a CPU cannot run more than one of those per core or that multiple “threads” cannot run in a single hardware thread

First of all: GPU myths

- 100x improvements
  - We can say that a GPU program gets 100x performance gain over a single threaded CPU implementation
  - We must not say that a GPU is 100x faster than the CPU as a result
  - We should not quote numbers in a paper abstract without at least saying what we are trying to prove

- So what is the lesson to consider here?
  - Define your terms first
  - Quote results based on those terms
  - Be able to defend what you mean
  - Do not fall into the trap of comparing algorithm performance and calling it device performance

Branchy code: a concept to consider

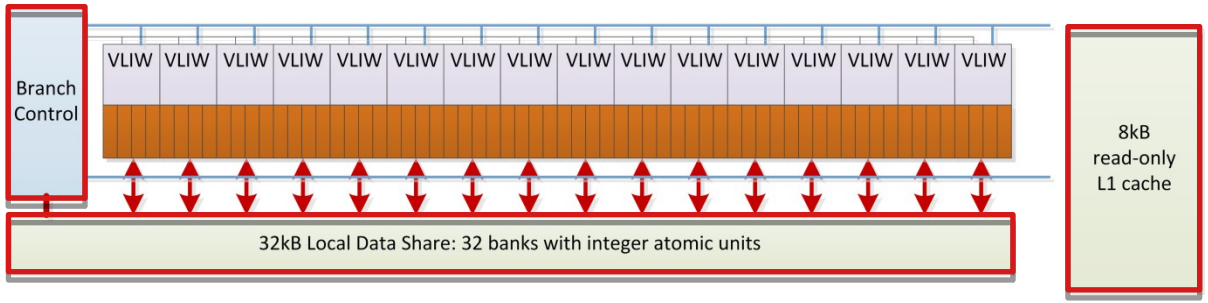
- GPUs are bad at executing branchy code
  - This is a claim we hear repeatedly
  - What does it really mean, and given a definition, is it actually true?

- The usual clarification is:
  - GPUs are bad at executing divergent branches

- Let me tell you a little story of evolution…

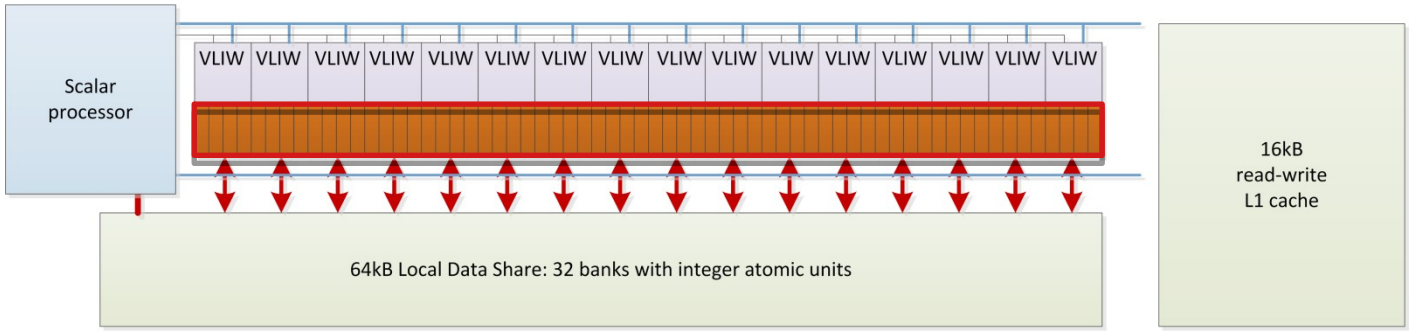
The SIMD core

- The SIMD unit on the AMD Radeon™ HD 6970 architecture (and related designs) had a branch control but full scalar execution was performed globally



The SIMD core

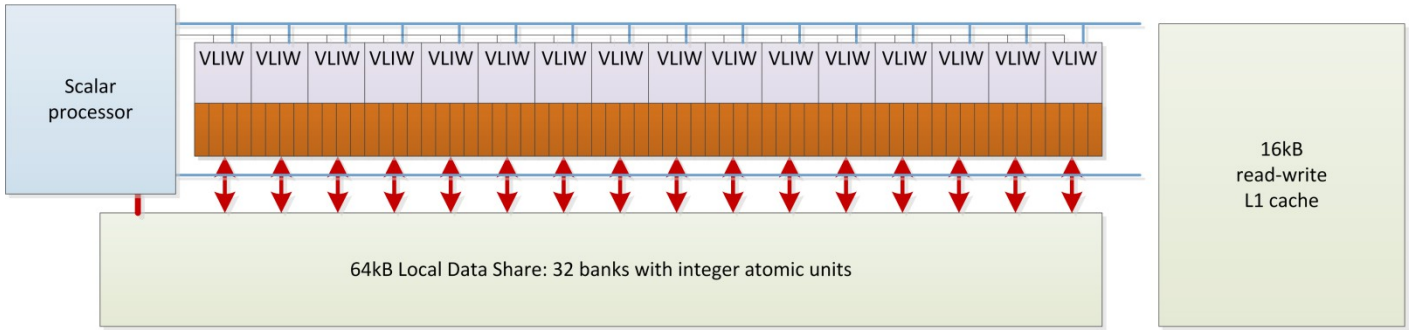
- On the AMD Radeon™ HD 7970 (and other chips; better known as “Graphics Core Next”) we have a full scalar processor and the L1 cache and LDS have been doubled in size
- Then let us consider the VLIW ALUs



- Notice that this already doesn’t seem quite so different from a CPU core with vector units?

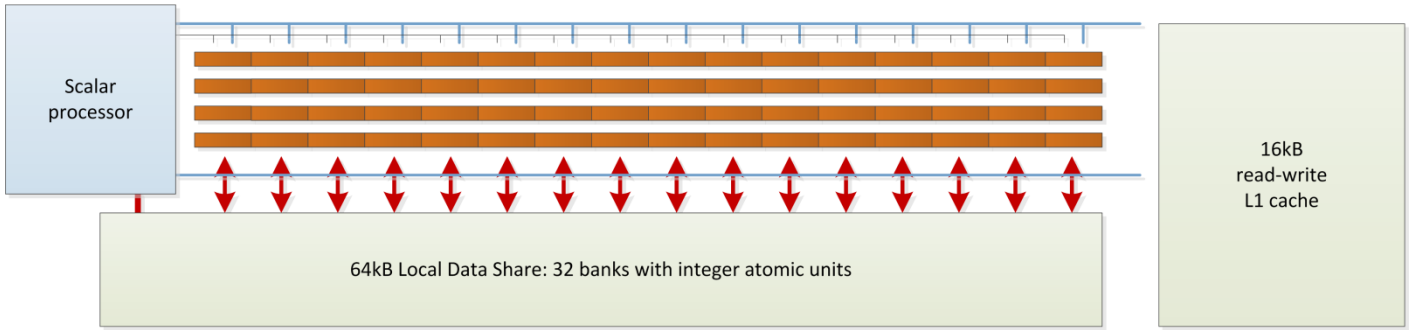
The SIMD core

- 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

Branching and the new ISA

```
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

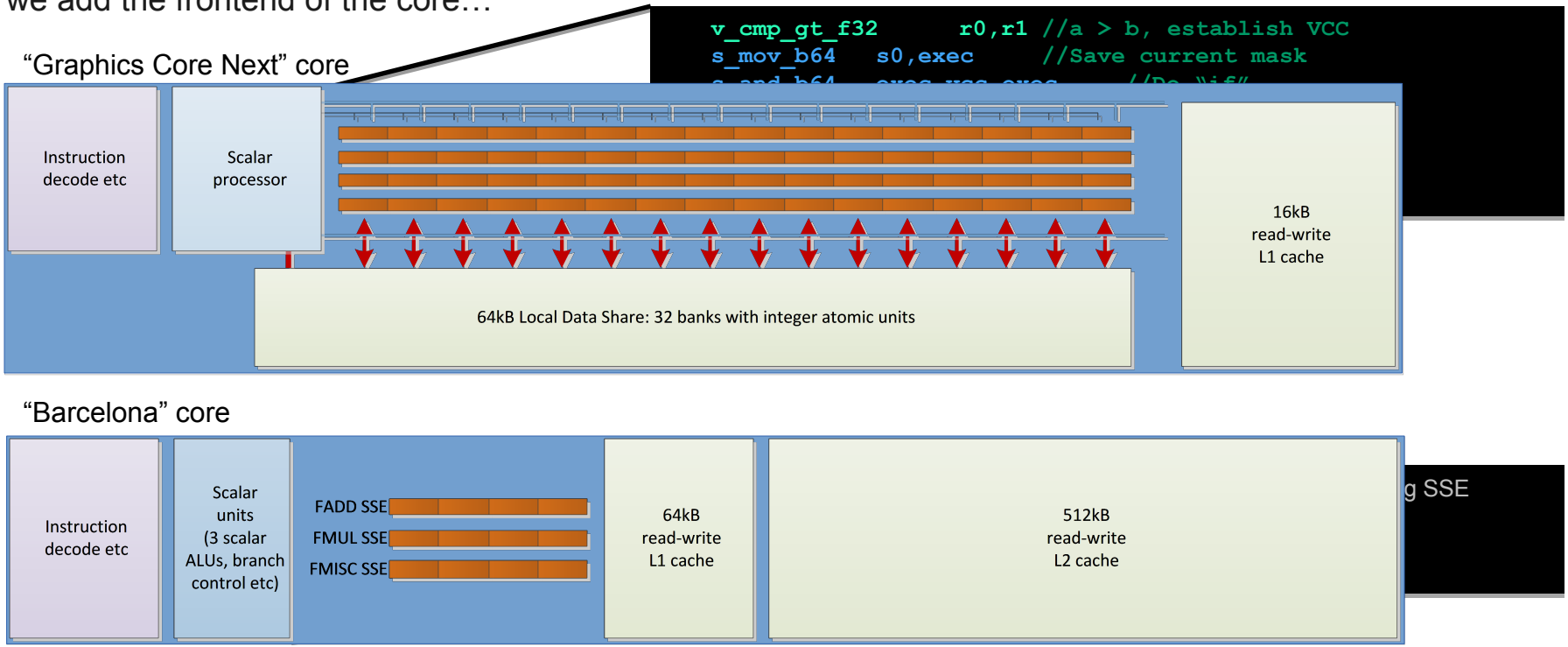
```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

  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

label0:
  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
label1:
  s_mov_b64 exec,s0   //Restore exec mask
```

Familiar?

- If we add the frontend of the core…



So returning to the topic of branches

- Are GPUs bad at executing branchy code?

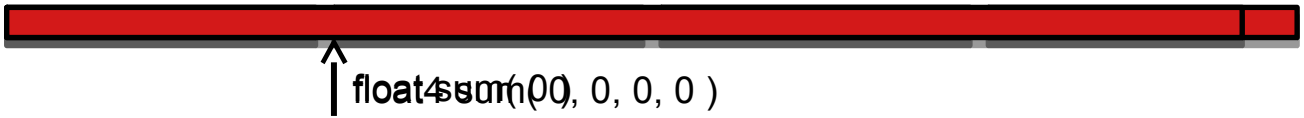
- Aren't CPUs just as bad at this?
  - Try running a divergent branch on an AVX unit.
  - Actually, CPUs are **worse** at this because they don't have the same flexibility in their vector units.
  - Which is easier to compile a pre-vectorized input like OpenCL to: CPU or GPU?

- So, what did we really mean?
  - GPUs are bad at executing branchy code IF we assume that the input was mapped in SPMD-on-SIMD fashion AND we do not assume the same about the CPU
  - Somehow that's less exciting…

- Instead, how **SHOULD** we be thinking about programming these devices?

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?
  - Take an input array
  - Block it based on the number of threads (8 or so per core up to 32 or so 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!

- For simple cases this is true
  - Vector issues for GPUs are not a programming problem
  - The differences are more about lazy CPU programming than difficulties of GPU programming

- However
  - GPUs really have been hard to program for
  - So let's look at the real problems and how we can address them

Improving the programming model

Fixing the composition problem

Function COMPOSITION

$$algorithm = f \bullet g$$

- The ability of decomposing a larger problem into sub-components
- The ability to replace one sub-component with another sub-component with the same external semantics

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
  - We need to pass data structures cleanly between threads

address spaces and communication

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
}
```

OpenCL™ C++ addresses memory spaces to a degree

- 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

- This works particularly well with the Khronos C++ API
  - #include <CL/cl.hpp>
  - Kernel functors:

```
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;
    }");
```

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;
}
```

- 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.

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

Memory consistency

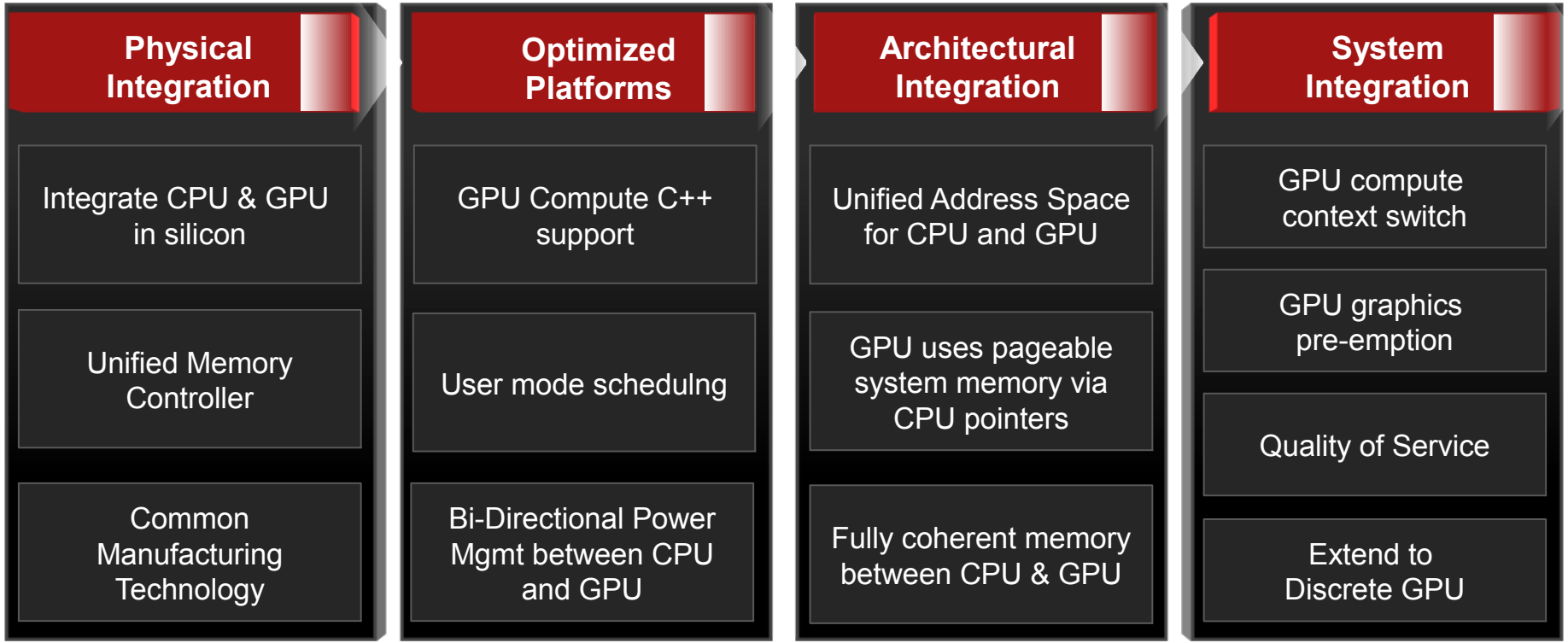
- We can also add a “flat” memory address space to improve the situation
  - Memory regions computed based on address, not instruction
- Even in global memory, communication is problematic
- We need to guarantee correct memory consistency
  - “at end of kernel execution” is simply not fine-grained enough for a lot of workloads
- Solution!
  - Define a clear memory consistency model

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
- With platform consistency, we can now pass and use pointer-based data structures correctly across the device boundary

*sequential consistent ordering

heterogeneous system architecture



Synchronization

Barrier synchronization IS not COMPOSaBLE

- Producer/consumer patterns in SPMD-on-SIMD model must take SIMD execution into account

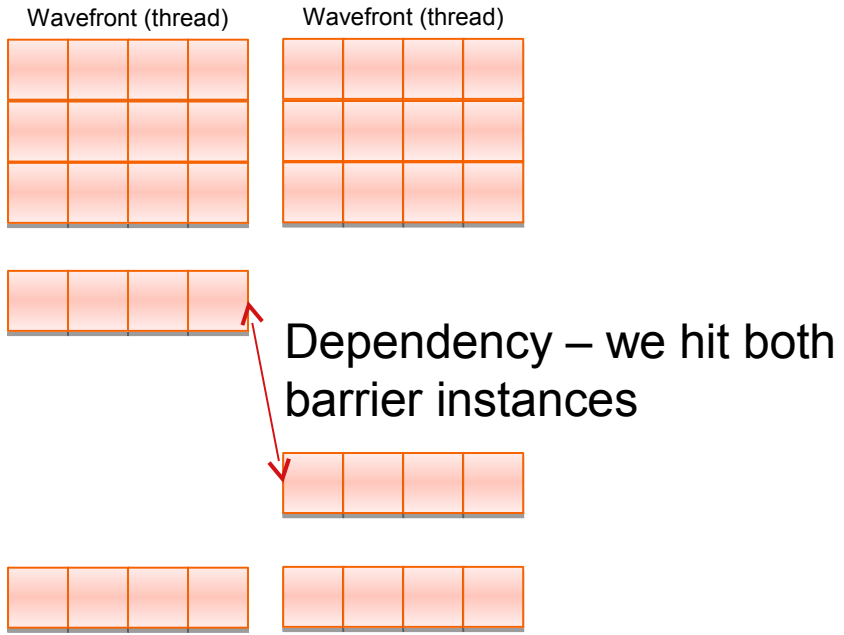
```
foreach work item i in buffer range
    for( some loop ) {
        if( work item is producer ) {
            // communicate
            barrier();
        } else {
            // communicate
            barrier();
        }
    }
}
```

Even if the barriers could be treated correctly, the SIMD width matters.

Barrier synchronization IS not COMPOSaBLE

- Producer/consumer patterns in SPMD-on-SIMD model must take SIMD execution into account

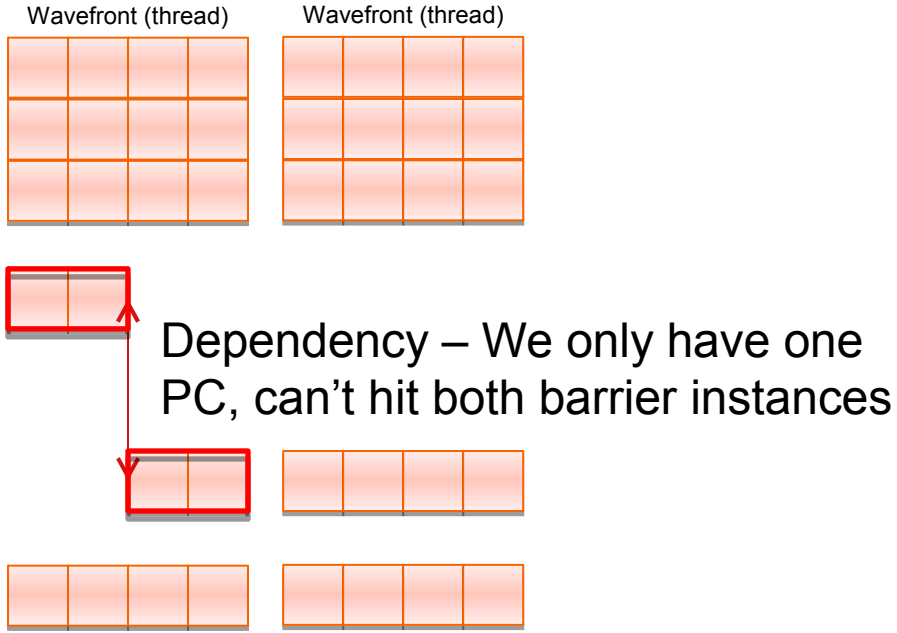
```
foreach work item i in buffer range
    for( some loop ) {
        if( work item is producer ) {
            // communicate
            barrier();
        } else {
            // communicate
            barrier();
        }
    }
}
```



Barrier synchronization IS not COMPOSaBLE

- Producer/consumer patterns in SPMD-on-SIMD model must take SIMD execution into account

```
foreach work item i in buffer range
    for( some loop ) {
        if( work item is producer ) {
            // communicate
            barrier();
        } else {
            // communicate
            barrier();
        }
    }
}
```



Barrier synchronization IS not COMPOSaBLE

- What if we use libraries?

```
foreach work item i in buffer range
    for( some loop ) {
        if( work item is producer ) {
            // communicate
            send();
        } else {
            // communicate
            receive();
        }
    }
}
```

Do send and receive contain barriers?

In the presence of libraries,wavefront-structured barriers are very hard to enforce.

Composability is at risk.

Making Barriers first class

- 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;
    }});
```

We can arrive or skip here, meaning that we acknowledge the barrier but do not wait on it if i >= 4

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;
  }});
```

Inside the loop we need to synchronize repeatedly

Early exit or skipping the loop entirely should never again affect synchronization within the loop.

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

Barrier used outside the library function and passed in

The **same barrier** used from within the library function

Barriers and threads

- Think about what this means about our definition of a thread

- Without first class barriers
  - A single work item can only be treated as a thread in the absence of synchronization
  - With synchronization the SIMD nature of execution must be accounted for

- With first class barriers
  - A single work item can be treated as a thread
  - It will behave, from a correctness point of view, like a thread
  - Only performance suffers from the SPMD-on-SIMD execution

What if?
Channels

CHANNELS - Persistent Control Processor Threading Model

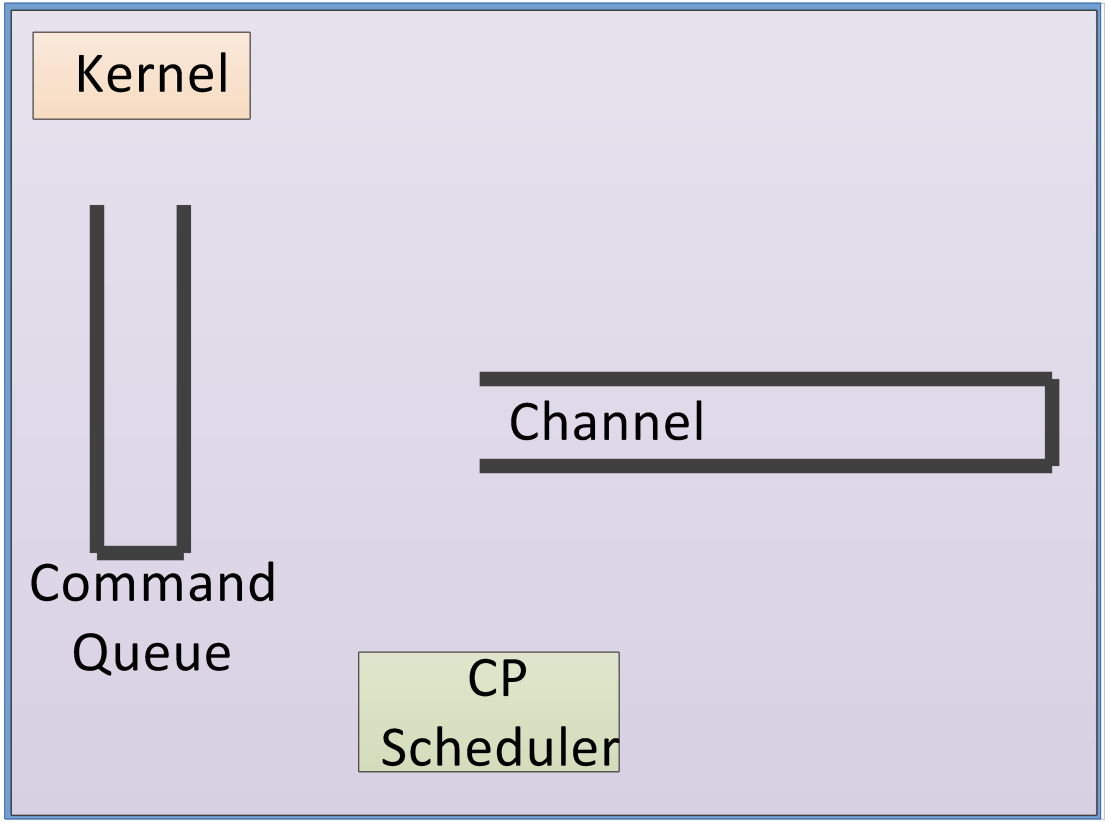
- Add data-flow support to GPGPU

- 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

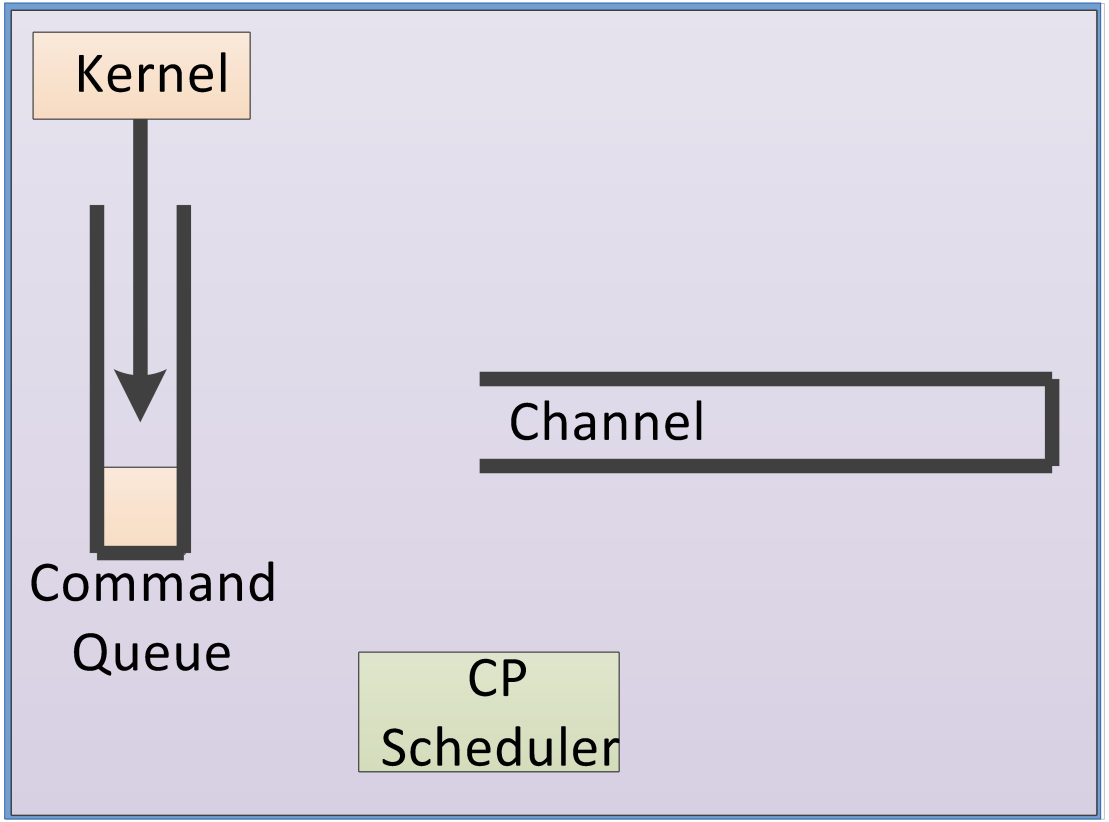
- Instead we assume that kernel entities produce/consume but consumer work-items are launched on-demand

- An alternative to the point to point data flow using of persistent threads, avoiding the uber-kernel

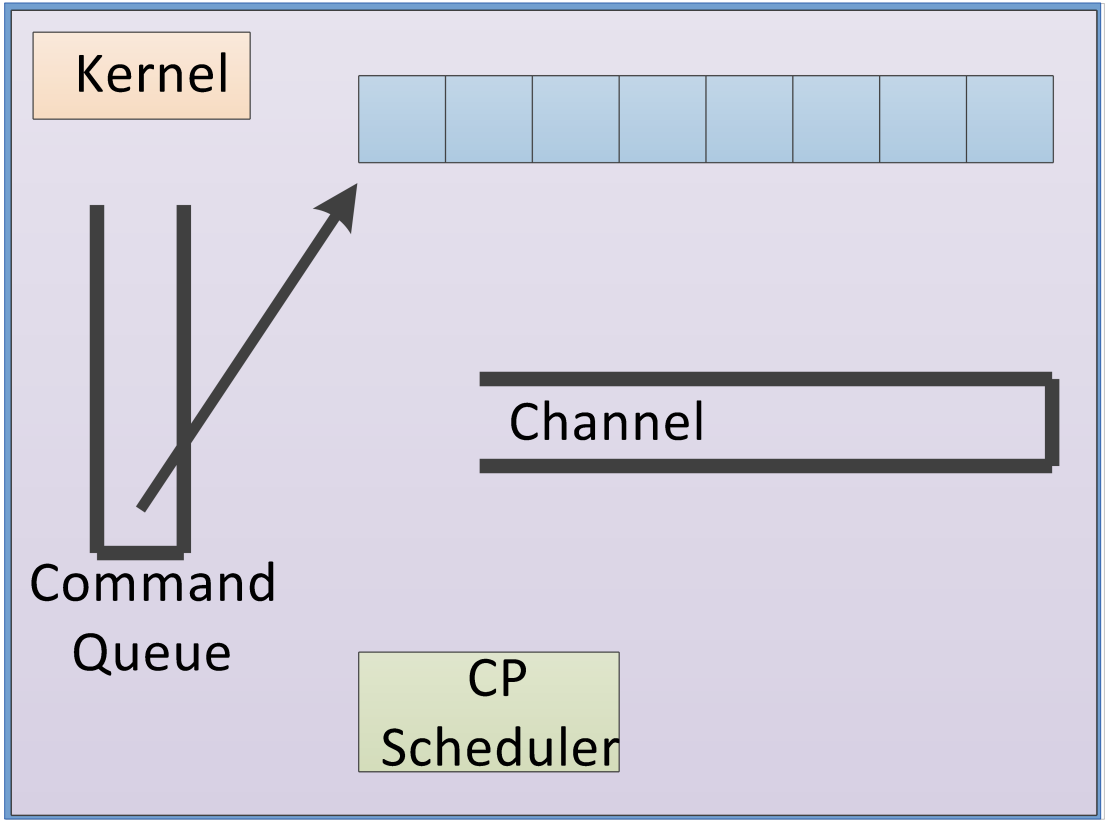
Operational Flow



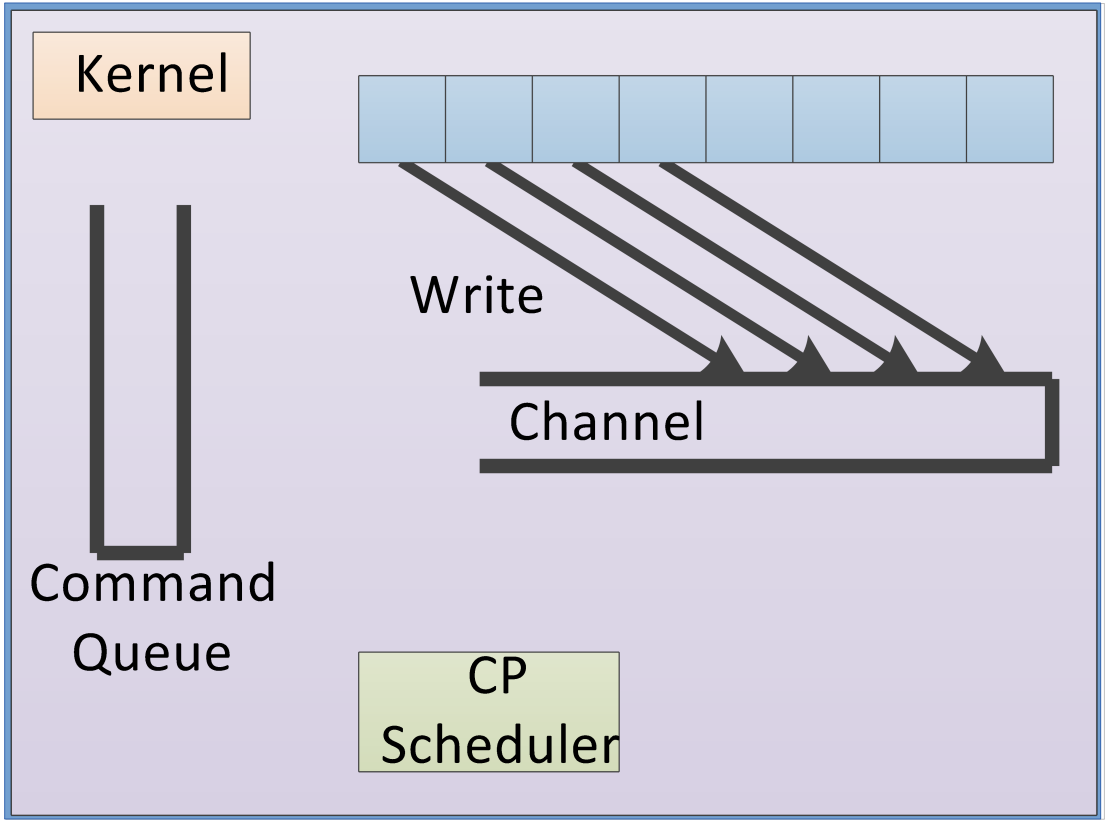
Operational Flow



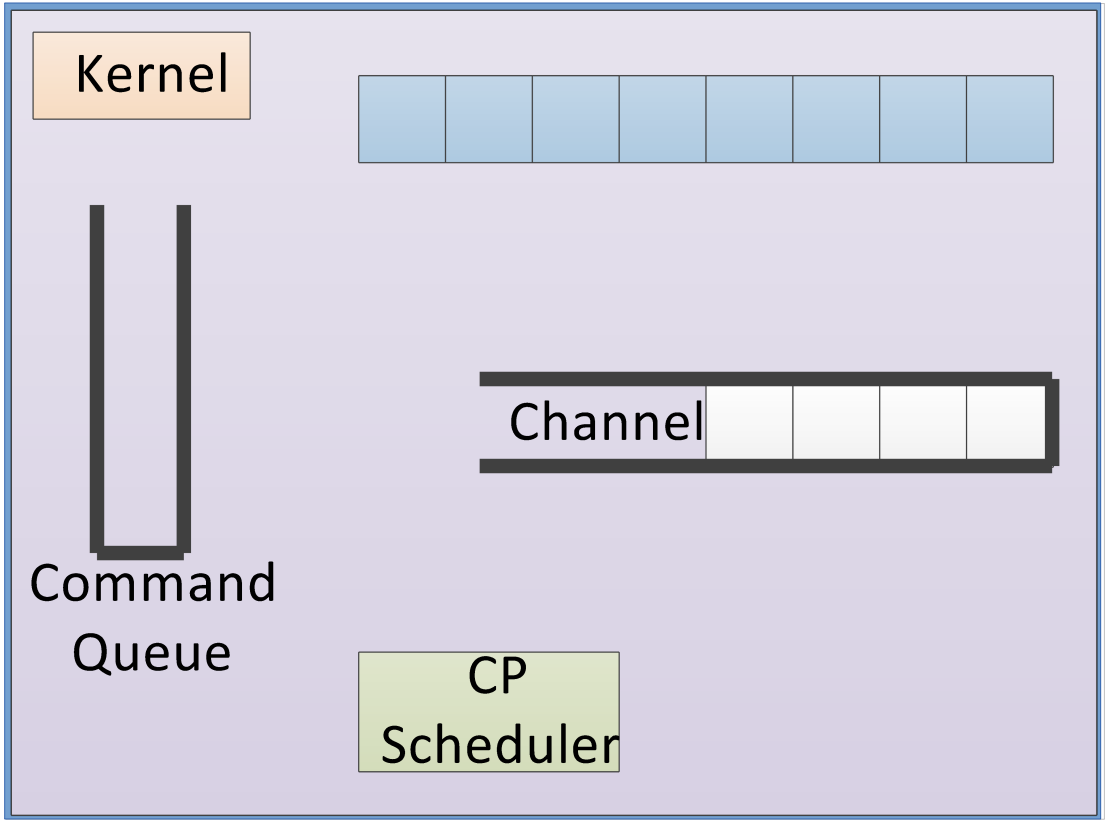
Operational Flow



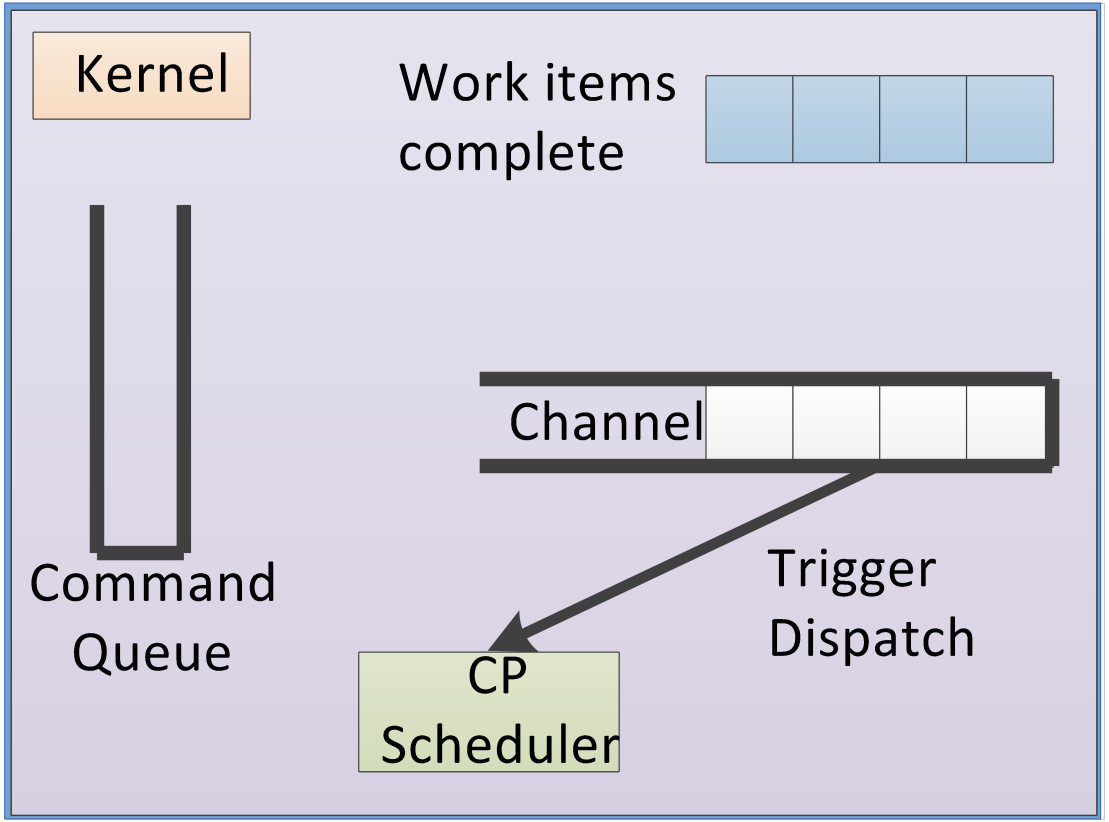
Operational Flow



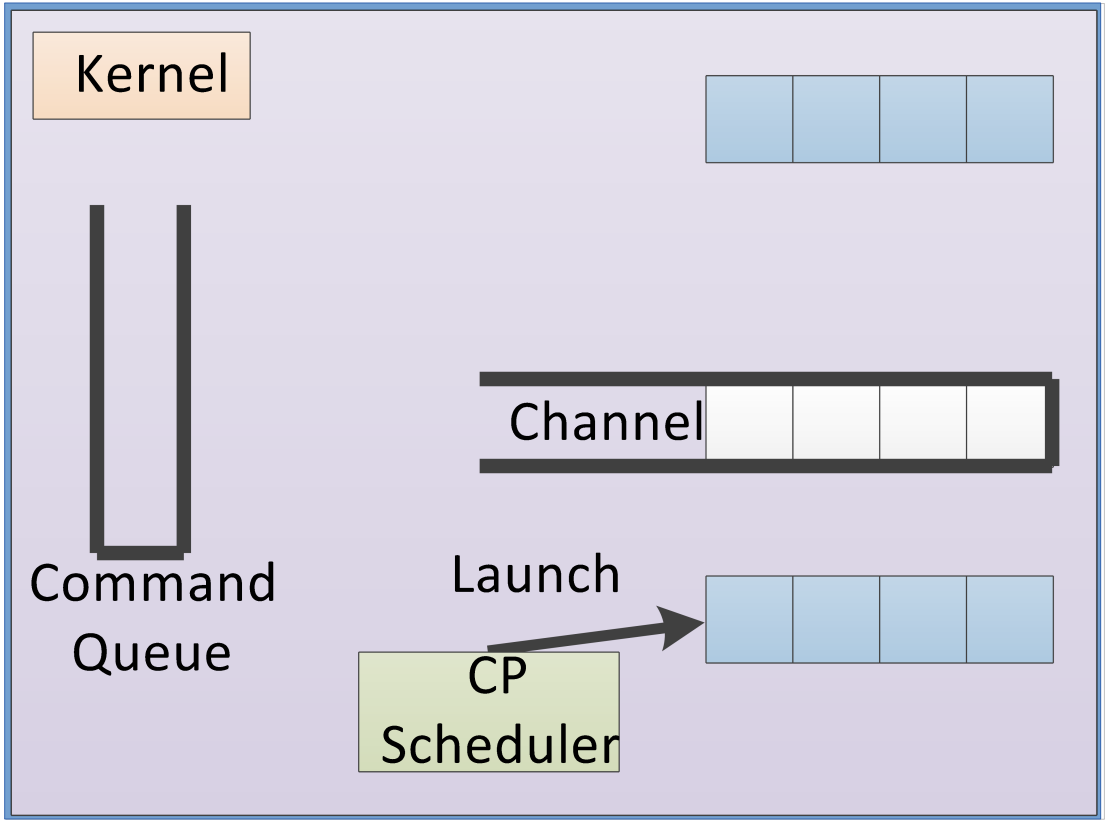
Operational Flow



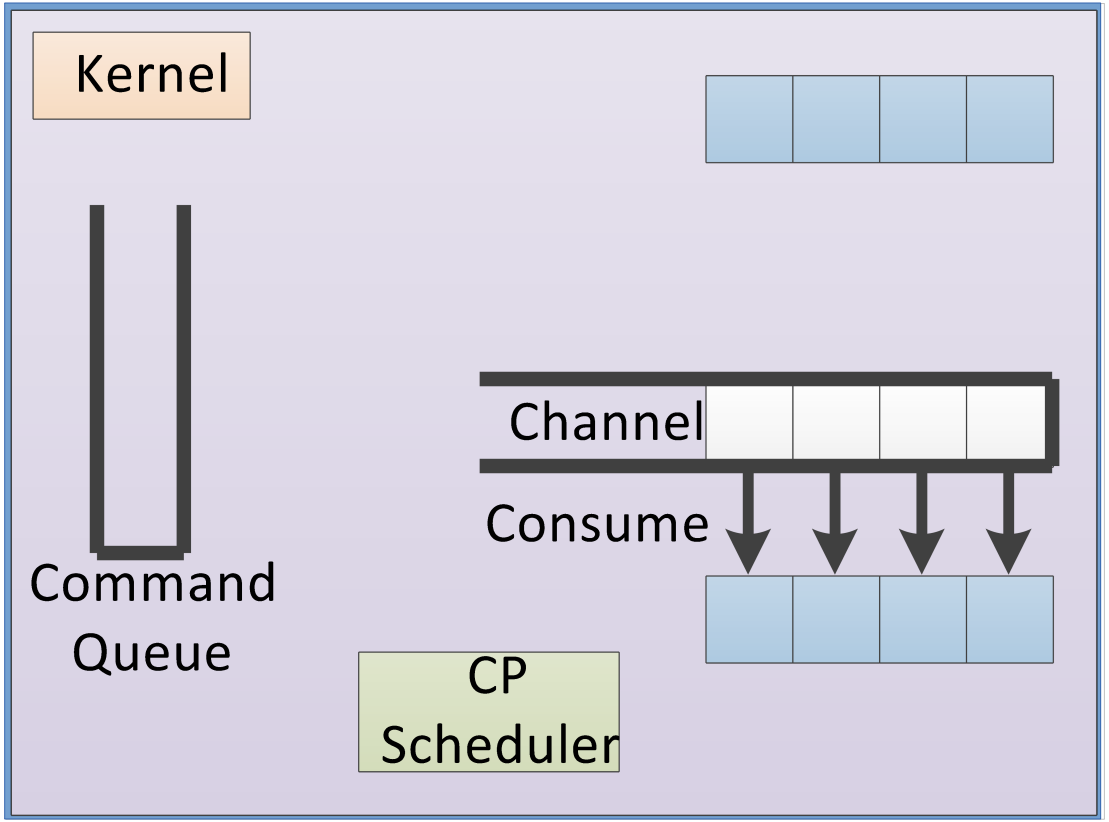
Operational Flow



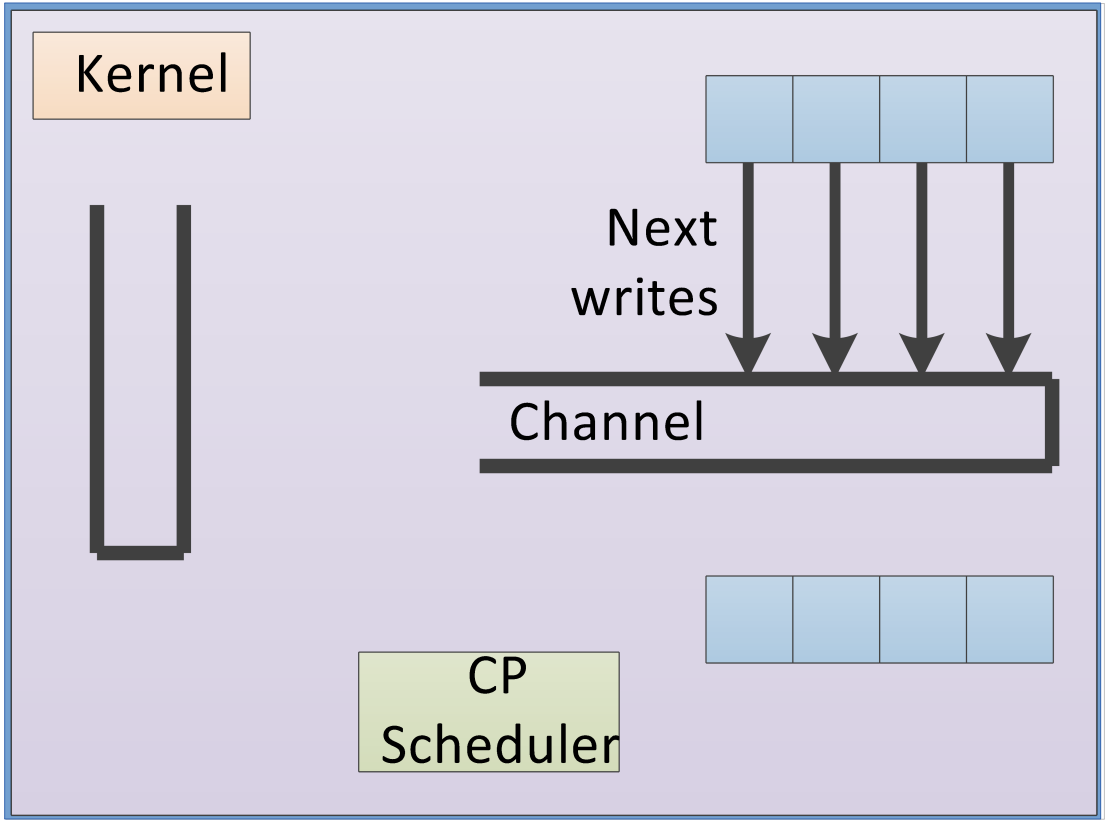
Operational Flow



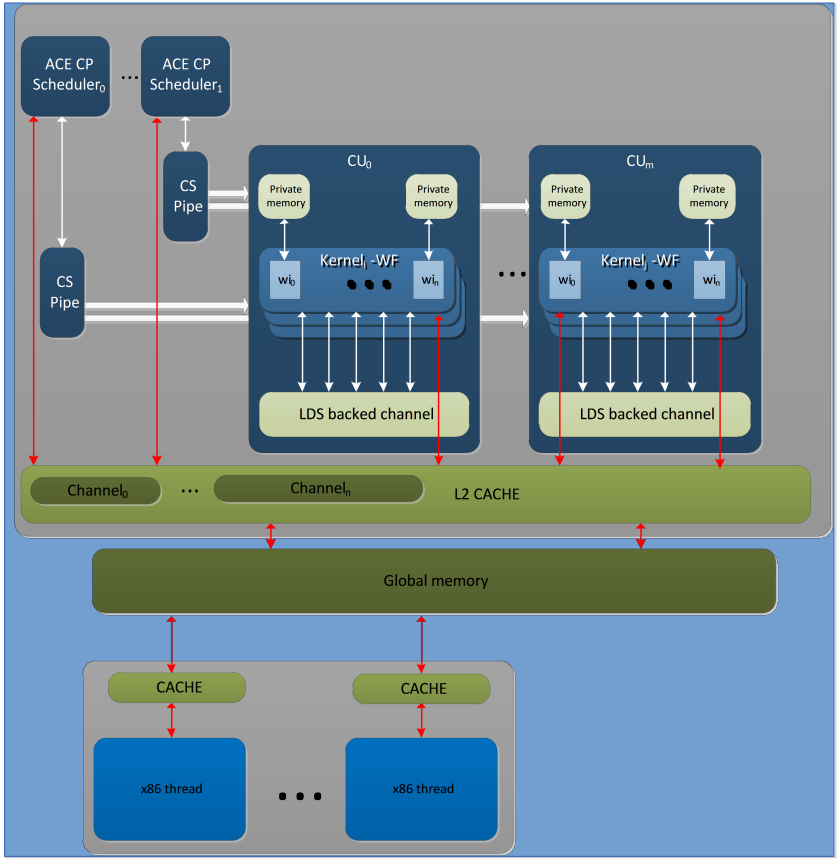
Operational Flow



Operational Flow



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);
        }
    });
```

Back to basics

Maybe we are going about it wrong?

- Do we really want to be doing these big data-parallel clustered dispatches?
  - What happens if we step a little back and rethink?

- What do people really want to do?
  - What about a motion vector search:

```
foreach block:
  while(found closest match):
    foreach element in comparison
```

- There is serial code mixed in with two levels of parallel code
  - So what if instead of trying to do a single blocked dispatch we explicitly layer the launch for the programmer?

Layered dispatch

```
parallelFor(int numThreads [](int index){ // "scalar" functor
  // do scalar stuff
  // for example, to do a motion estimation we might do
  X = int2(0,0);
  sumSq = 0.f;
  while( sumSq still being minimized ){
    X = new position from X'
    Accumulator<int2> acc;
    localParallelFor(int2(-5, -5) to int2(5, 5), [=X](int2
      auto diff = currentFrame(X + index) - previousFrame(X
      acc += diff;
    }); // end localParallelFor

    If sumSq < acc // See if it's small enough or continu
    X' = X
  }});
```

A parallel thread launch:
One wavefront/workgroup launched for each index

A serial loop that searches for the best match. This is best written as a serial loop and is clean scalar

Parallel loop:
Covers each pixel in the block being compared. Easily vectorisable and with no launch overhead.

Research directions

- There is a need to look at more source languages targeting heterogeneous and vector architectures
  - C++AMP and similar models fill a certain niche, but they target simplification the tool chain rather than expanding the scope of the programming model

- HSA will help
  - Multi-vendor
  - Architected low-level APIs
  - Well defined low-level intermediate language
    - Vitally: with largely preserved optimizations. Finalizer will not interfere too much with code generation.
    - Fast linking.

Conclusions

- HSA takes some steps towards improving the execution model and giving a programming model target
  - HSA does not directly improve the programming model
  - It offers an excellent low-level framework

- It is time to start enthusiastically looking at the programming model
  - Let's program the system in a single clean manner
  - Let's forget nonsense about how GPUs are vastly different beasts from CPUs – it's just not true
  - Let's get away from 400x speedups, and magical efficiency gains and look at the system realistically
  - But let's not forget: performance portability without abstraction is a pipe dream

- Please use HSA as a starting point
  - Go wild and use your imaginations

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, AMD Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used with permission by Khronos. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.