

# AMD and OpenCL

Mike Houston  
Senior System Architect  
Advanced Micro Devices, Inc.



# Overview

Brief overview of ATI Radeon™ HD 4870 architecture and operation

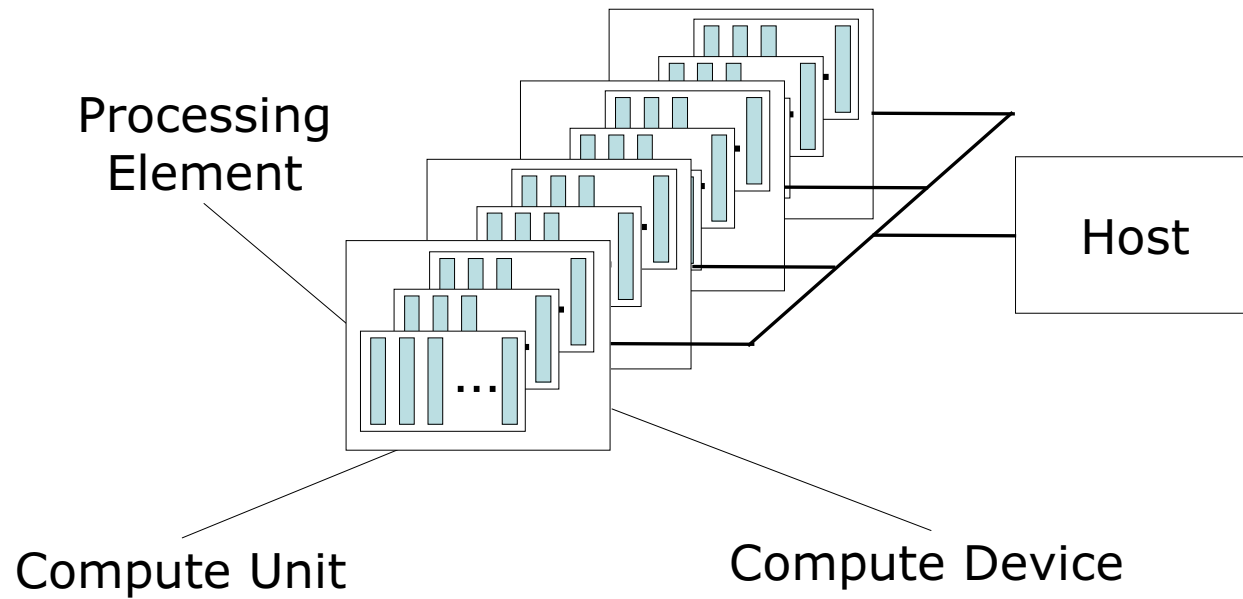
“Spreadsheet” performance analysis

Tips & tricks

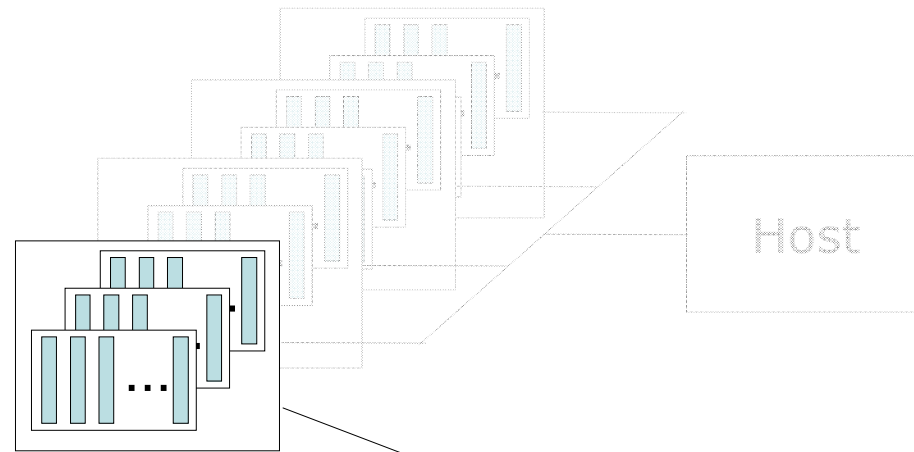
Demo



# Review: OpenCL View



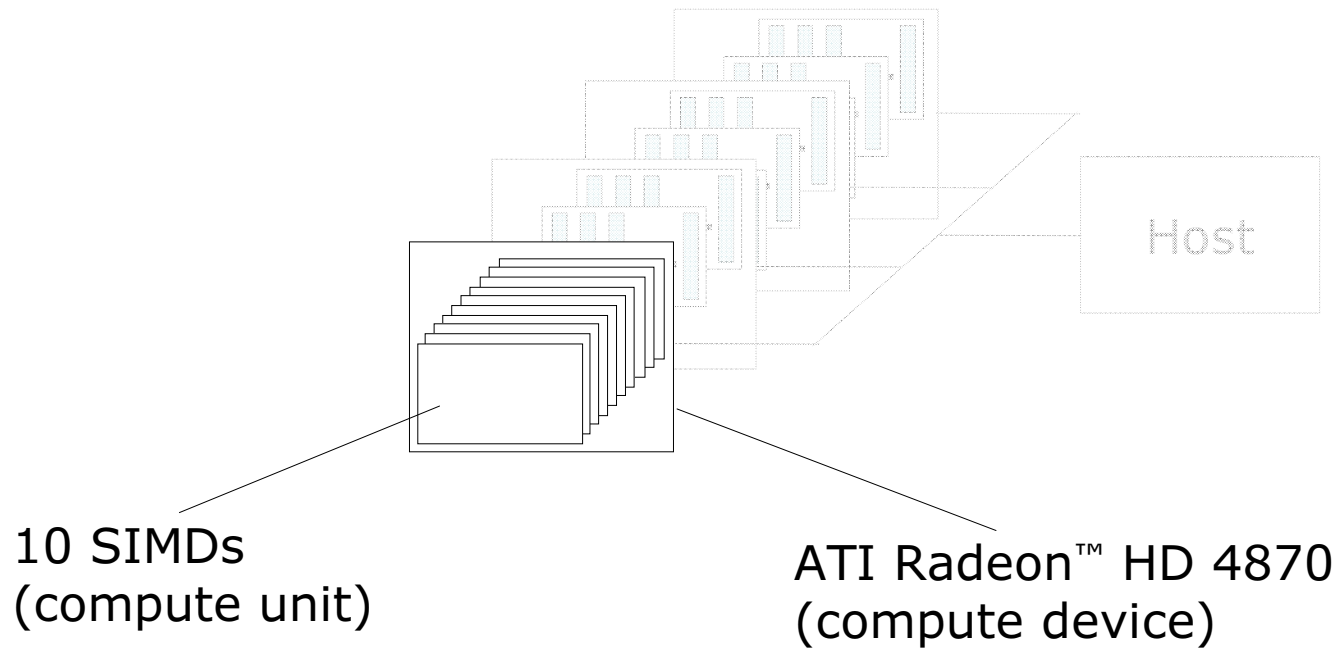
# Review: OpenCL View



ATI Radeon™ HD 4870  
(compute device)



# Review: OpenCL View

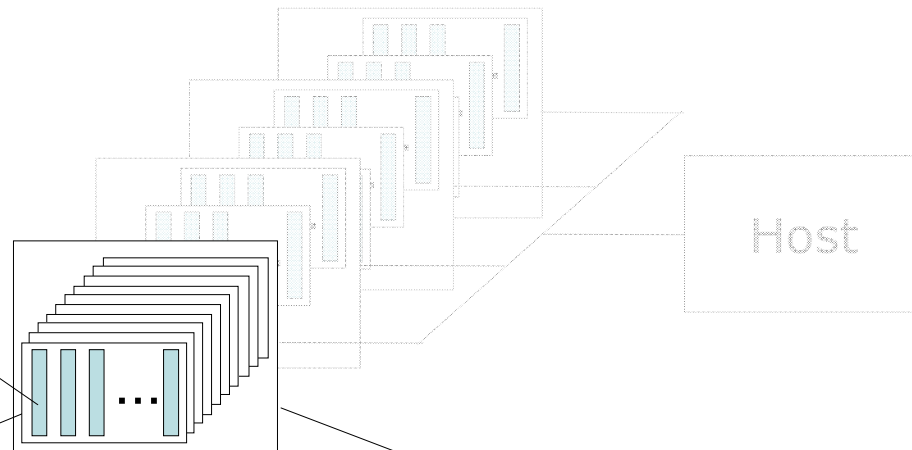


# Review: OpenCL View

64 Element Wide  
"wavefront"  
(Processing elements)

10 SIMDs  
(compute unit)

ATI Radeon™ HD 4870  
(compute device)



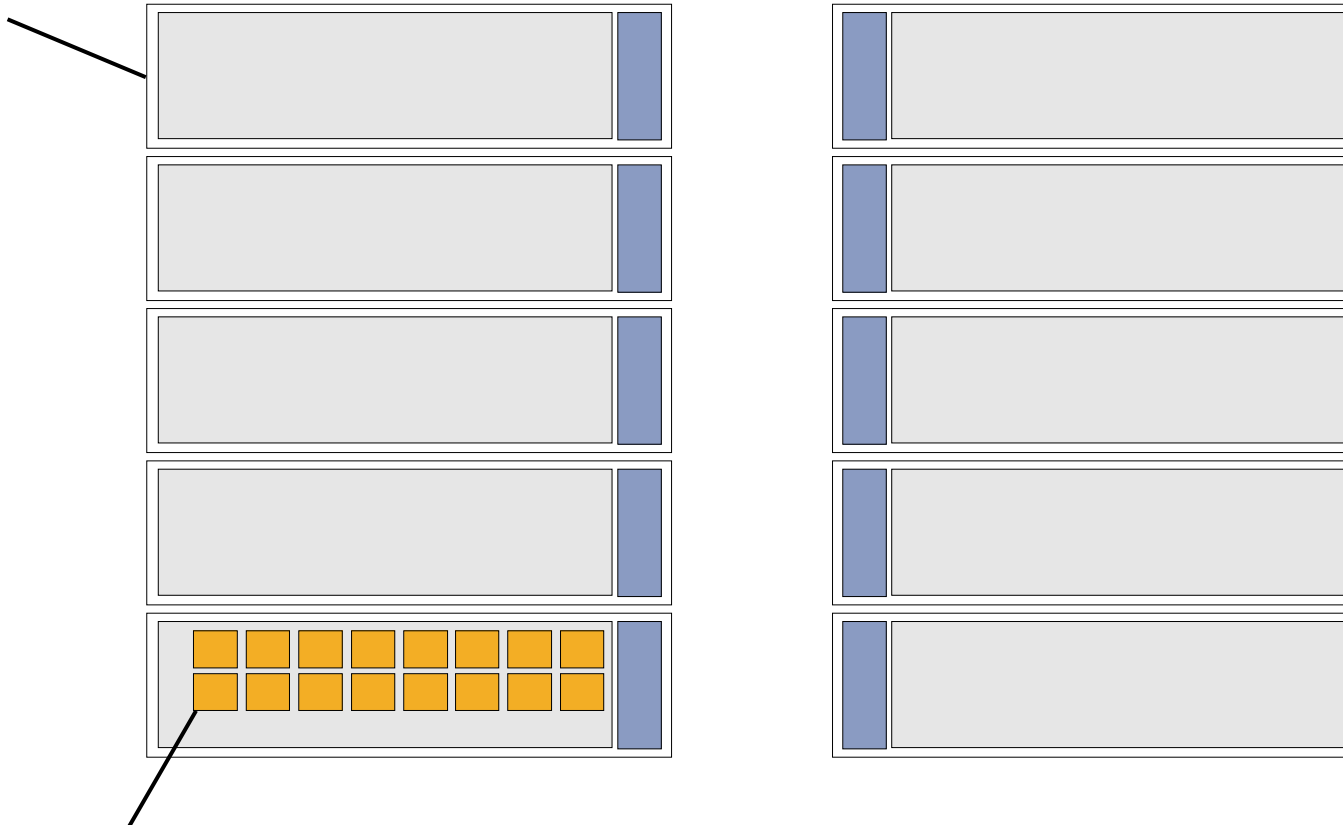
# ATI Radeon™ HD 4870 : *Reality*

**10** SIMDs



# ATI Radeon™ HD 4870 : Reality

**10** SIMDs



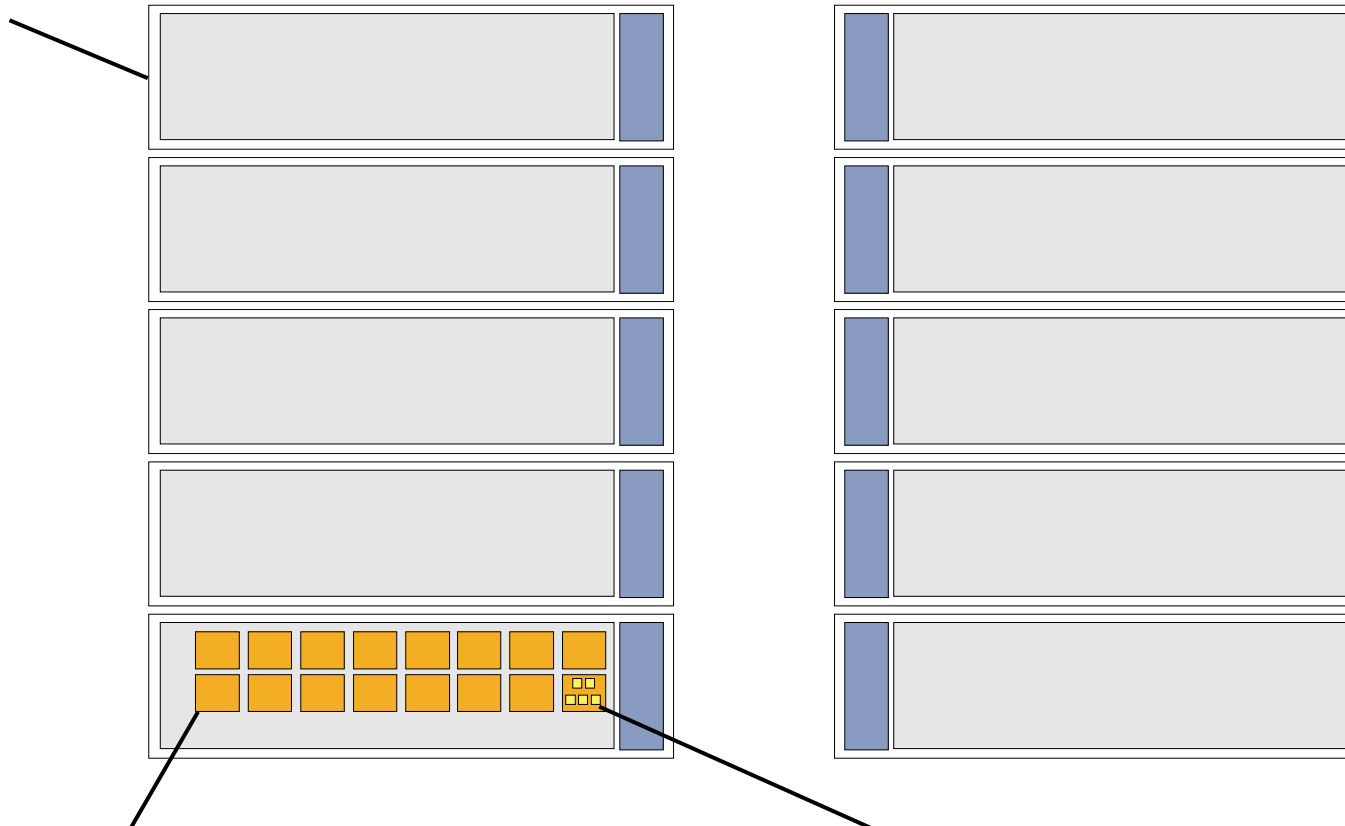
**16** Processing "cores" per SIMD  
(**64** Elements over 4 cycles)





# ATI Radeon™ HD 4870 : Reality

**10** SIMDs

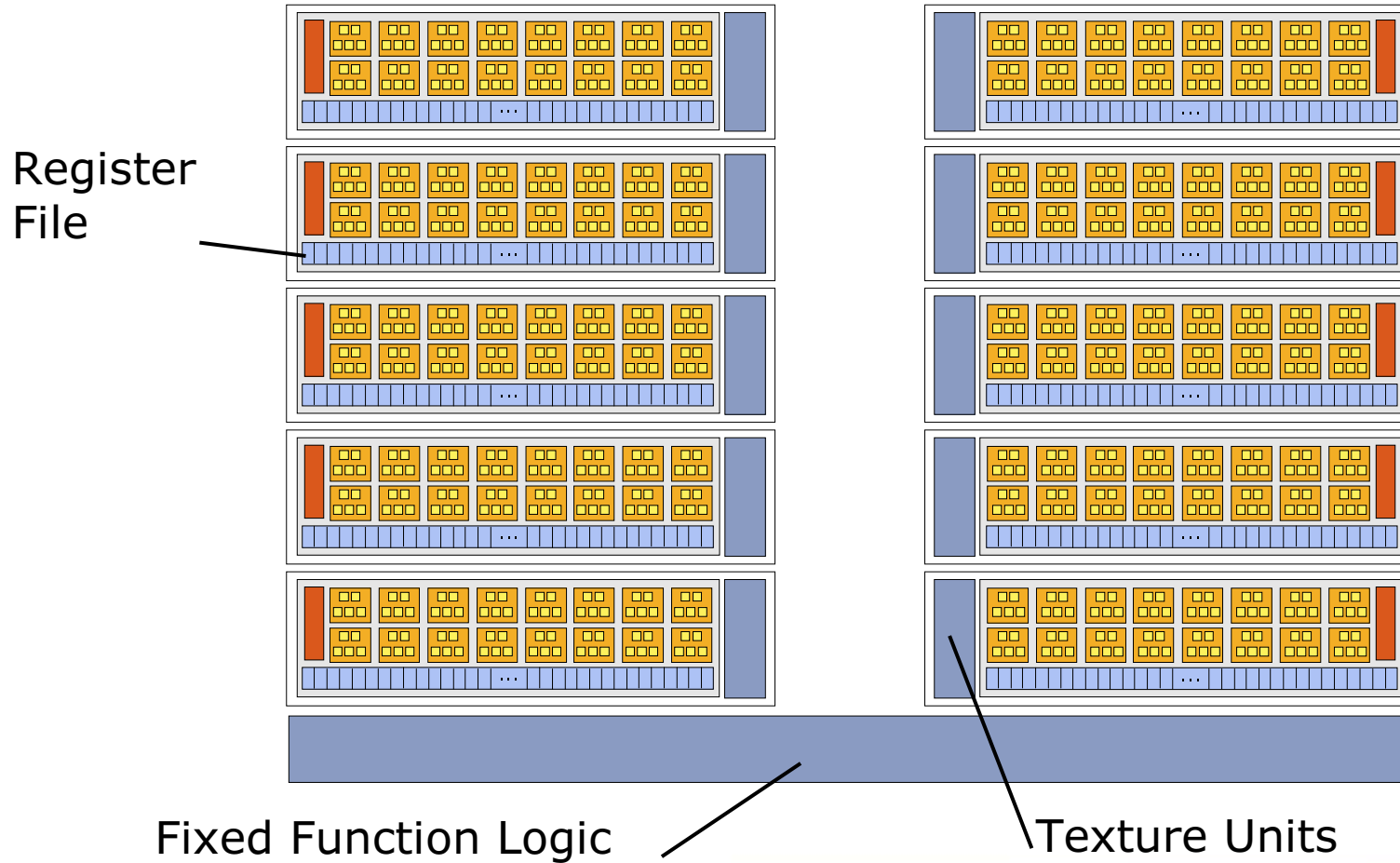


**16** Processing "cores" per SIMD  
(**64** Elements over 4 cycles)

**5** ALUs per "core"  
(VLIW Processors)



# ATI Radeon™ HD 4870 : Reality



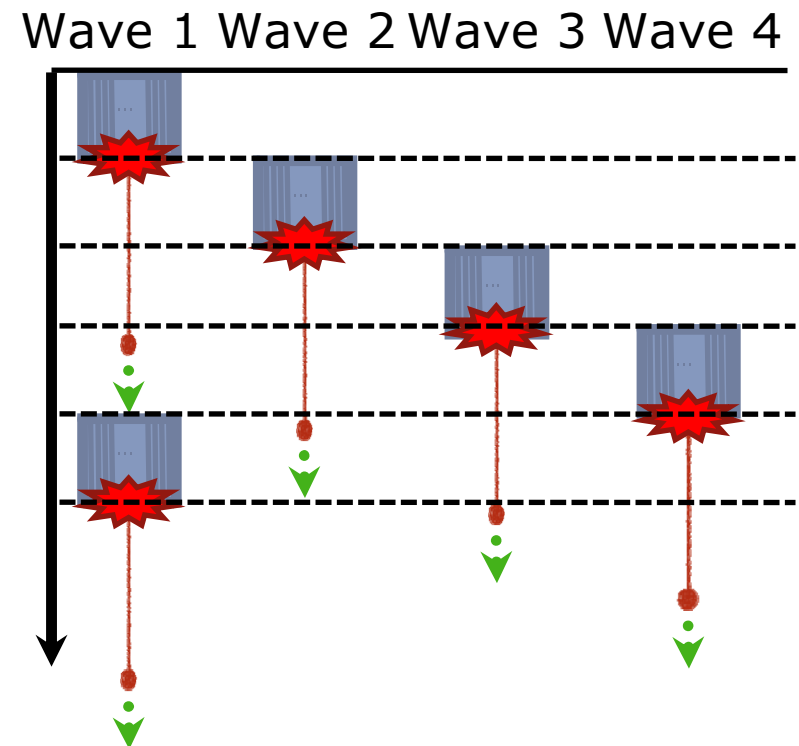
# Latency Hiding

GPUs eschew large caches for large register files

Ideally launch more “work” than available ALUs

Register file partitioned amongst active wavefronts

Fast switching on long latency operations



# “Spreadsheet Analysis”

Valuable to estimate performance of kernels

- Helps identify when something is “wrong”

First-order bottlenecks: ALU, TEX, or MEM

- $T_{\text{kernel}} = \max(\text{ALU}, \text{TEX}, \text{MEM})$

$$T_{\text{alu}} = \text{global\_size} * \text{VLIW instructions} / (\text{\# SIMDs} * \text{\# cores per SIMD}) / \text{Engine Clock}$$

The diagram shows the equation  $T_{\text{alu}} = \text{global\_size} * \text{VLIW instructions} / (\text{\# SIMDs} * \text{\# cores per SIMD}) / \text{Engine Clock}$  with colored lines connecting terms to their definitions:   
- **global\_size** (green) connects to **#elements** (green) in the equation.   
- **VLIW instructions** (red) connects to **#ALU** (red) in the equation.   
- **# SIMDs** (yellow) connects to **(10\*16)** (yellow) in the equation.   
- **"cores" per SIMD** (teal) connects to **16** (teal) in the equation.   
- **Engine Clock** (purple) connects to **750 Mhz** (purple) in the equation.



# Practical Implications on ATI Radeon™ HD 4870

Workgroup size should be a multiple of 64

- Remember: *Wavefront* is 64 elements
- Smaller workgroups SIMDs will be underutilized

SIMDs operate on pairs of wavefronts



# Minimum Global Size on ATI Radeon™ HD 4870

**10** SIMDs \* **2** waves \* **64** elements = **1280** elements

- Minimum global size to utilize GPU with one kernel
- Does not allow for any latency hiding!

For minimum latency hiding: **2560 elements**



# Register Usage

Recall GPUs hide latency by switching between large number of wavefronts

Register usage determines maximum number of wavefronts in flight

More wavefronts better latency hiding

Fewer wavefronts worse latency hiding

Long runs of ALU instructions can compensate for low number of wavefronts



# Kernel Guidelines on ATI Radeon™ HD 4870

Prefer **int4** / **float4** when possible

- Processor “cores” are 5-wide VLIW machines
- Memory system prefers 128-bit load/stores

Consider access patterns - e.g. access along rows

AMD GPUs have large register files

- Perform “more work per element” - example later





# Median Filtering

Non-linear filter for removing “one-shot” noise in images

Simple Algorithm:

1. Load neighborhood
2. Sort pixels - example uses an efficient method
3. Output median value



# Simple Kernel

```
__kernel void medianfilter_rgba( __global uint *id, __global uint *od, int width, int h, int r )  
{  
    Compute my index  
    const int posx = get_global_id(0);  
    const int posy = get_global_id(1);  
    const int idx = posy*width + posx;  
  
    Handle edge cases  
    if( posx == 0 || posy == 0 || posx == width-1 || posy == h-1 )  
    {  
        od[ idx ] = id[ idx ];  
    }  
    else  
    {  
  
    Load neighborhood  
        uint row00, row01, row02, row10, row11, row12, row20, row21, row22;  
        row00 = id[ idx - 1 - width ]; row01 = id[ idx - width ]; row02 = id[ idx + 1 - width ];  
        row10 = id[ idx - 1 ]; row11 = id[ idx ]; row12 = id[ idx + 1 ];  
        row20 = id[ idx - 1 + width ]; row21 = id[ idx + width ]; row22 = id[ idx + 1 + width ];  
  
    Sort along the rows  
        // sort the rows  
        sort3( &(row00), &(row01), &(row02) );  
        sort3( &(row10), &(row11), &(row12) );  
        sort3( &(row20), &(row21), &(row22) );  
  
    Sort along the cols  
        // sort the cols  
        sort3( &(row00), &(row10), &(row20) );  
        sort3( &(row01), &(row11), &(row21) );  
        sort3( &(row02), &(row12), &(row22) );  
  
    Sort along the diagonal  
        // sort the diagonal  
        sort3( &(row00), &(row11), &(row22) );  
  
    Output median  
        // median is the middle value of the diagonal  
        od[ idx ] = row11;  
    }  
}
```



# Auxiliary Functions

Convert to luminance

```
float rgbToLum( uint a )  
{  
    return (0.3f*(a & 0xff) + 0.59*((a>>8) & 0xff) + 0.11*((a>>16) & 0xff));  
}
```

Swap A and B

```
void swap( uint *a, uint *b )  
{  
    uint tmp = *b;  
    *b = *a;  
    *a = tmp;  
}
```

Comparison sort

```
void sort3( uint *a, uint *b, uint *c )  
{  
    if( rgbToLum( *a ) > rgbToLum( *b ) )  
        swap( a, b );  
    if( rgbToLum( *b ) > rgbToLum( *c ) )  
        swap( b, c );  
    if( rgbToLum( *a ) > rgbToLum( *b ) )  
        swap( a, b );  
}
```



# More Work per Work-item

Prefer read/write 128-bit values

Compute more than one output per work-item

Better Algorithm (further optimizations possible):

1. Load neighborhood 8x3 via six 128-bit loads
2. Sort pixels for each of four pixels
3. Output median values via 128-bit write



# Better Kernel

```
__kernel void medianfilter_x4( __global uint *id, __global uint *od, int width, int h, int r )  
{  
    Compute my index  
    const int posx = get_global_id(0); // global width is 1/4 image width  
    const int posy = get_global_id(1); // global height is image height  
    const int width_d4 = width >> 2; // divide width by 4  
    const int idx_4 = posy*(width_d4) + posx;  
  
    Load row above  
    uint4 left0, right0, left1, right1, left2, right2, output;  
    // ignoring edge cases for simplicity  
    left0 = ((__global uint4*)id)[ idx_4 - width_d4 ];  
    right0 = ((__global uint4*)id)[ idx_4 - width_d4 + 1];  
  
    Load my row  
    left1 = ((__global uint4*)id)[ idx_4 ];  
    right1 = ((__global uint4*)id)[ idx_4 + 1];  
  
    Load row below  
    left2 = ((__global uint4*)id)[ idx_4 + width_d4 ];  
    right2 = ((__global uint4*)id)[ idx_4 + width_d4 + 1];  
  
    Find four medians  
    // now compute four median values  
    output.x = find_median( left0.x, left0.y, left0.z,  
                            left1.x, left1.y, left1.z, left2.x, left2.y, left2.z );  
    output.y = find_median( left0.y, left0.z, left0.w,  
                            left1.y, left1.z, left1.w, left2.y, left2.z, left2.w );  
    output.z = find_median( left0.z, left0.w, right0.x,  
                            left1.z, left1.w, right1.x, left2.z, left2.w, right2.x );  
    output.w = find_median( left0.w, right0.x, right0.y,  
                            left1.w, right1.x, right1.y, left2.w, right2.x, right2.y );  
  
    Output medians  
    ((__global uint4*)od)[ idx_4 ] = output;  
}
```



# Demo



# Conclusions

Optimization is a balancing act

Things to consider:

- Register usage / number of wavefronts in flight
- ALU to memory access ratio
  - Sometimes better re-compute something
- Workgroup size a multiple of 64
- Global size at least 2560 for a single kernel



# Multi-Core x86 CPU implementation available now!

<http://developer.amd.com/streambeta>

Submitted for conformance during SIGGRAPH for Microsoft® Windows® XP 32-bit, Windows Vista® 32-bit, Windows® 7 32-bit, Linux® 32-bit, Linux® 64-bit

Performance advice (similar to GPU):

- Use 128-bit types (i.e. float4)
  - Will map well to SSE
- Use reasonably large work groups
  - L1/L2 cache locality
- Do a lot of work per work item

Start writing code and playing with OpenCL now on  
any x86 machine





## DISCLAIMER

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

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## ATTRIBUTION

© 2009 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, ATI, the ATI logo, Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Windows, and Windows Vista are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. OpenCL is trademark of Apple Inc. used under license to the Khronos Group Inc. Other names are for informational purposes only and may be trademarks of their respective owners.

