Adopting OpenCL in Your Application

Session 522

Anna Tikhonova
OpenCL Engineer

What Is OpenCL?

- C-based language
- Run same code on CPUs and GPUs

Agenda

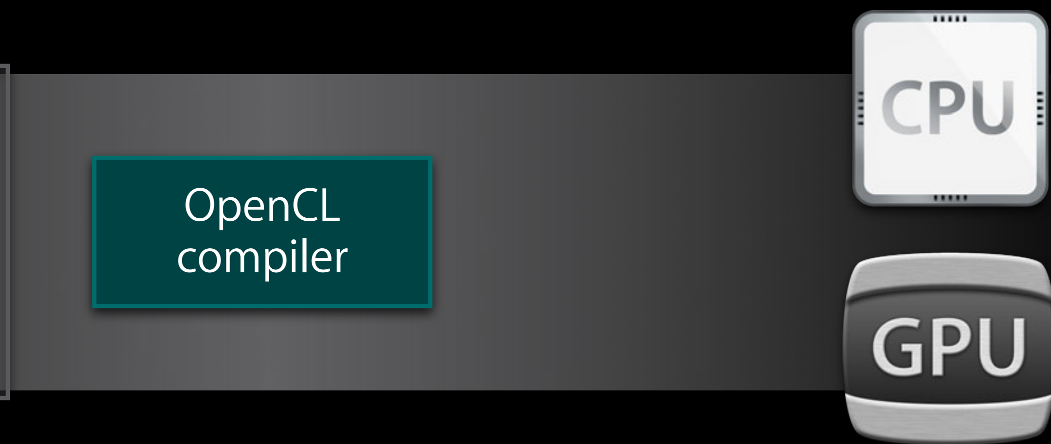
- What’s new in OpenCL in Mountain Lion
  - OpenCL 1.2
  - Improvements to the Intel Auto-Vectorizer
- From C code to optimized OpenCL code
- Leveraging the power of OpenCL in Adobe CS6

OpenCL 1.2

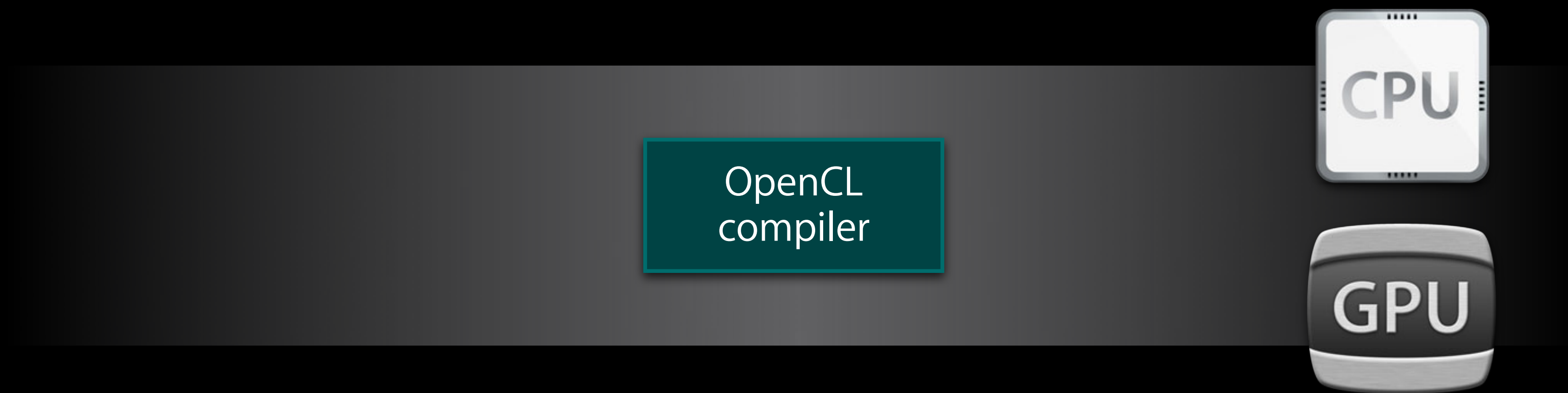
Program Compilation

Online

```
kernel void sum(
global float *a,
global float *b,
global float *c) {
  int id = get_global_id(0);
  c[id] = a[id] + b[id];
}
```

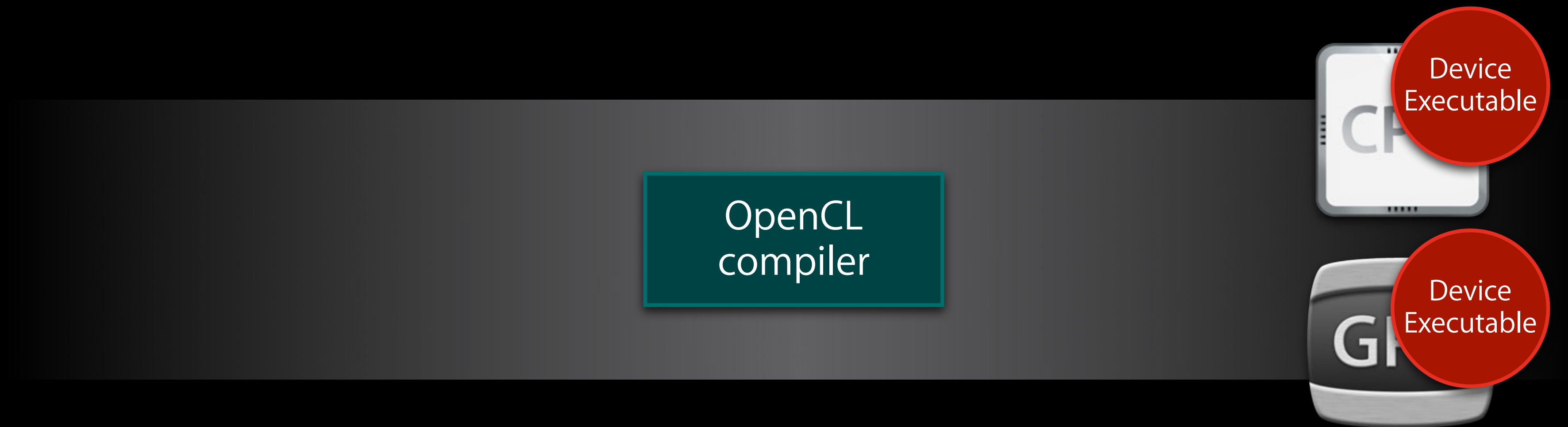


Program Compilation
Online



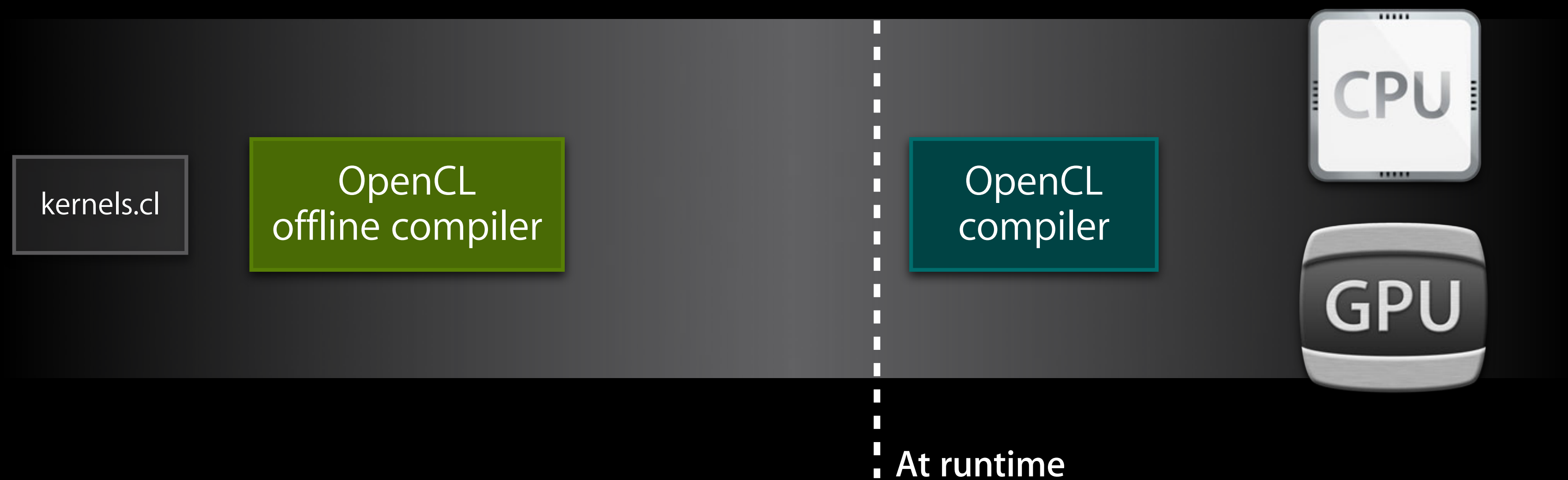
Program Compilation

Online



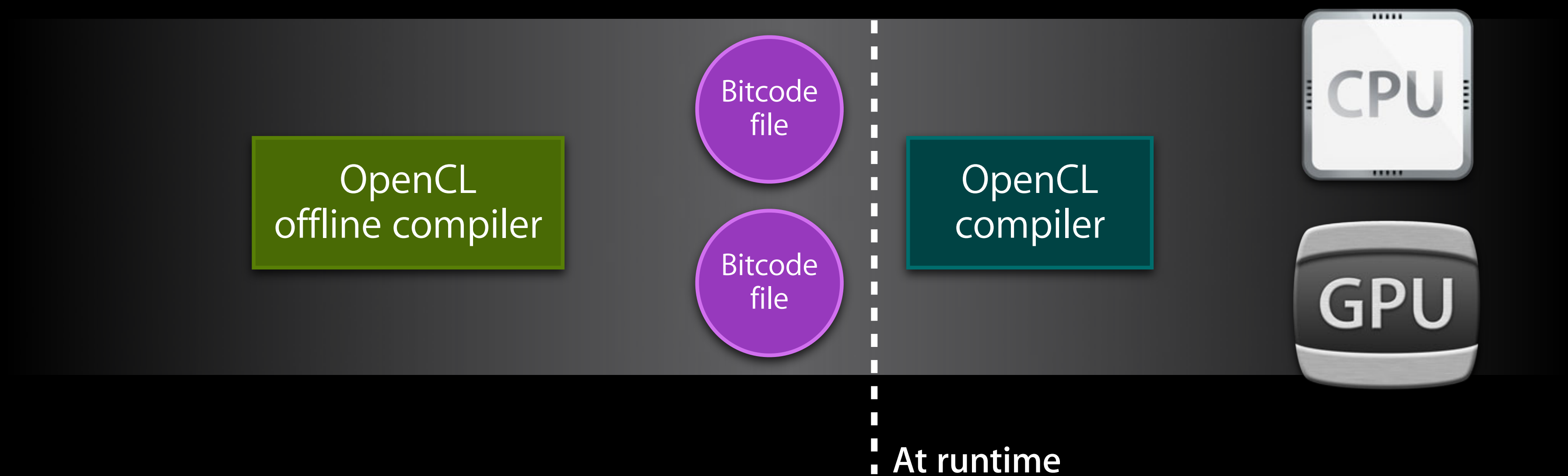
Program Compilation

Offline



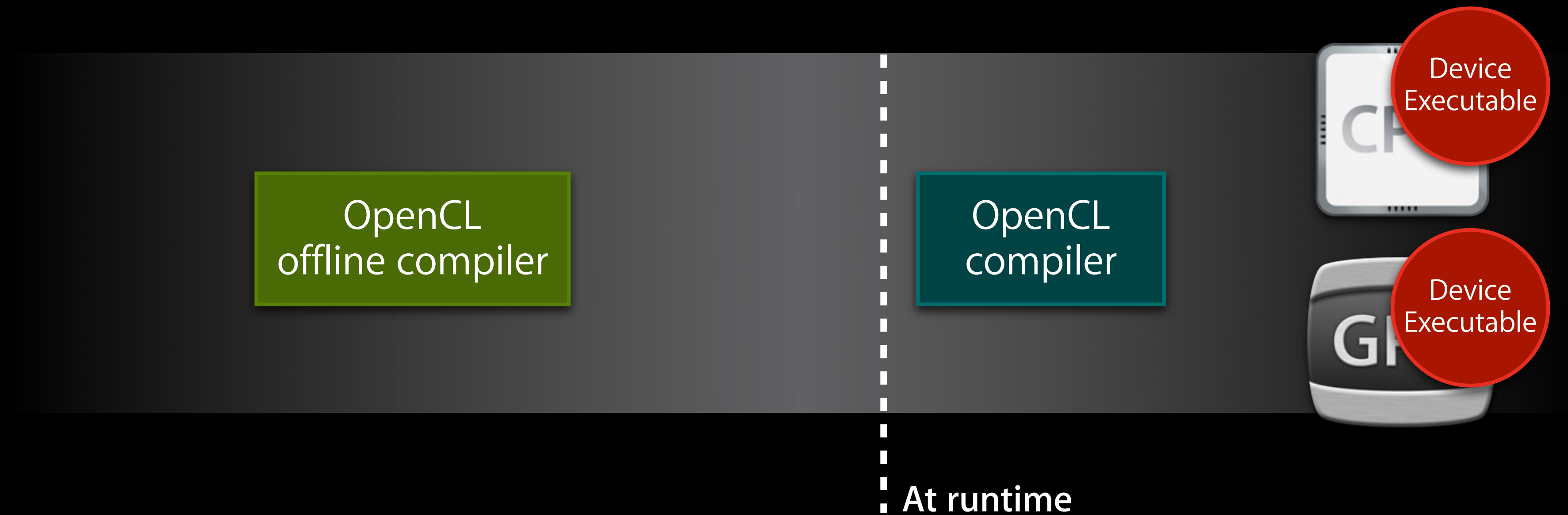
Program Compilation

Offline



Program Compilation

Offline



Compiling to Bitcode

/System/Library/Frameworks/OpenCL.framework/Libraries/openclc

Compiling to Bitcode

/System/Library/Frameworks/OpenCL.framework/Libraries/openclc

```
$ openclc -x cl -arch gpu_32 -emit-llvm-bc file.cl -o file.cl.gpu_32.bc
                -arch i386                            file.cl.i386.bc
                -arch x86_64                          file.cl.x86_64.bc
```

Compiling to Bitcode

/System/Library/Frameworks/OpenCL.framework/Libraries/openclc

```
$ openclc -x cl -arch gpu_32 -emit-llvm-bc file.cl -o file.cl.gpu_32.bc
                -arch i386                            file.cl.i386.bc
                -arch x86_64                          file.cl.x86_64.bc
```

Libraries of OpenCL Kernels

Traditional compile and link model

New

Compile

Link

Libraries of OpenCL Kernels

Traditional compile and link model

blur_filters.cl

noise_reduction_filters.cl

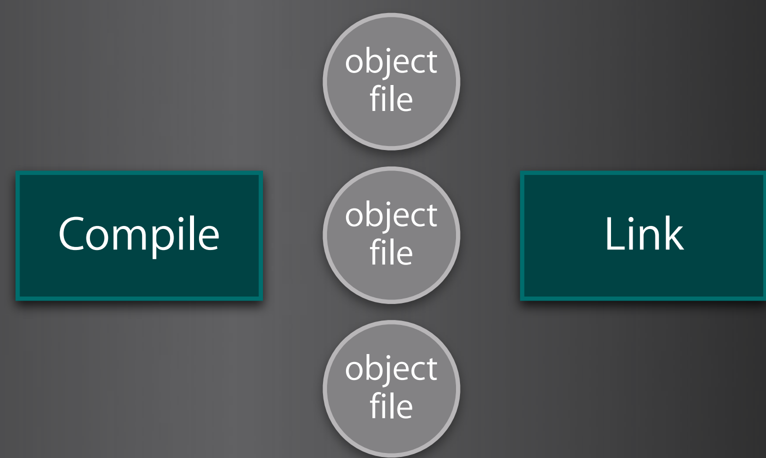
edge_enhancement_filters.cl

Compile

Link

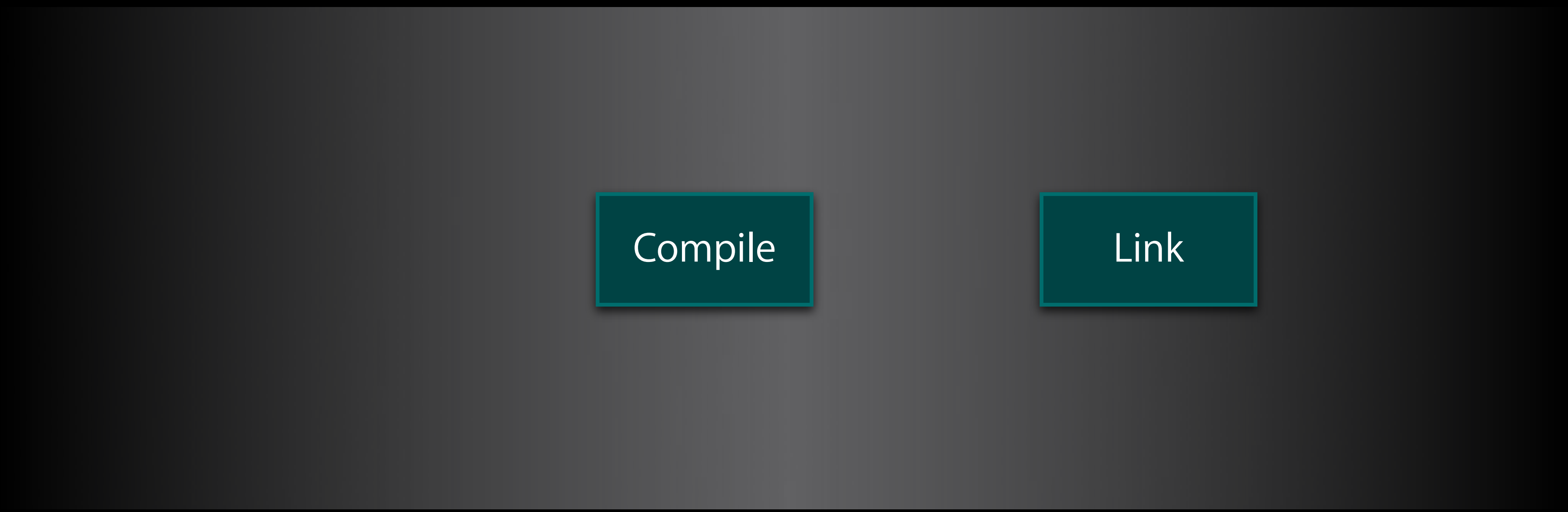
Libraries of OpenCL Kernels

Traditional compile and link model



Libraries of OpenCL Kernels

Traditional compile and link model

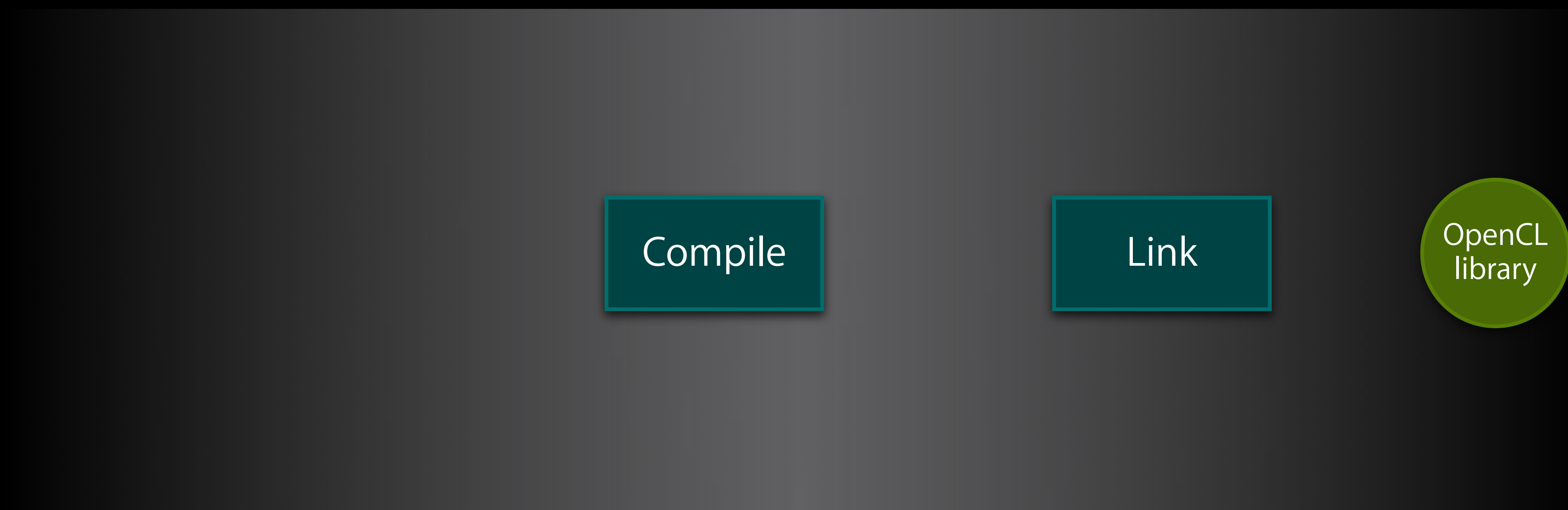


Libraries of OpenCL Kernels

Traditional compile and link model

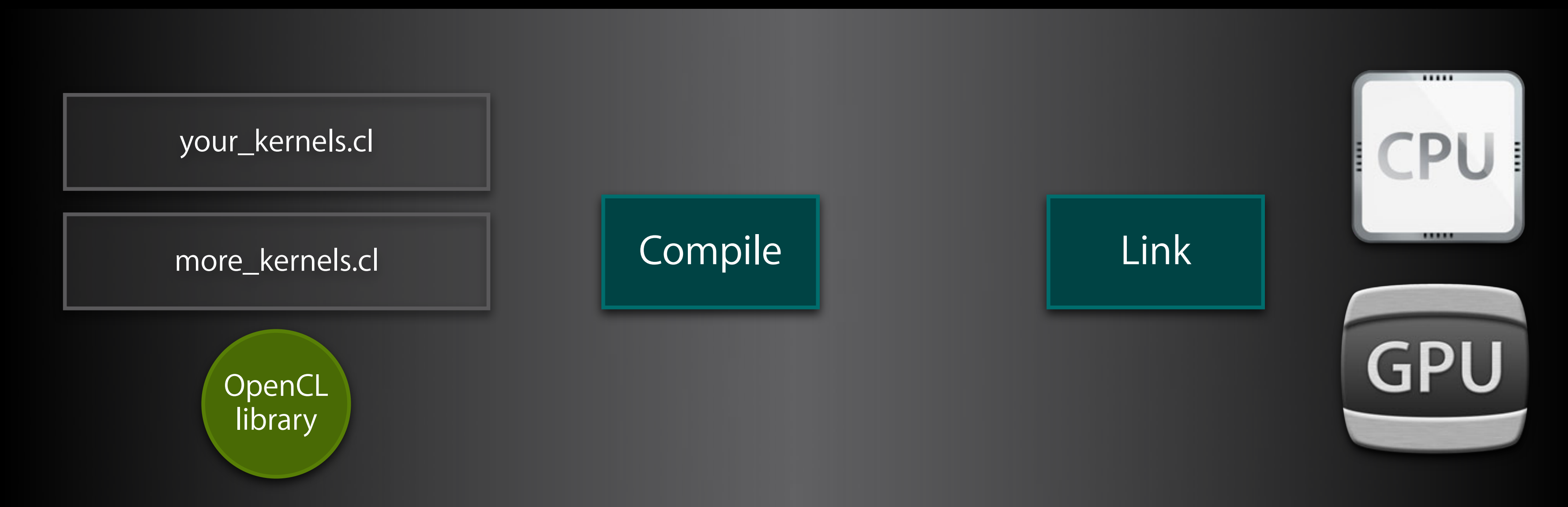
Libraries of OpenCL Kernels

Traditional compile and link model



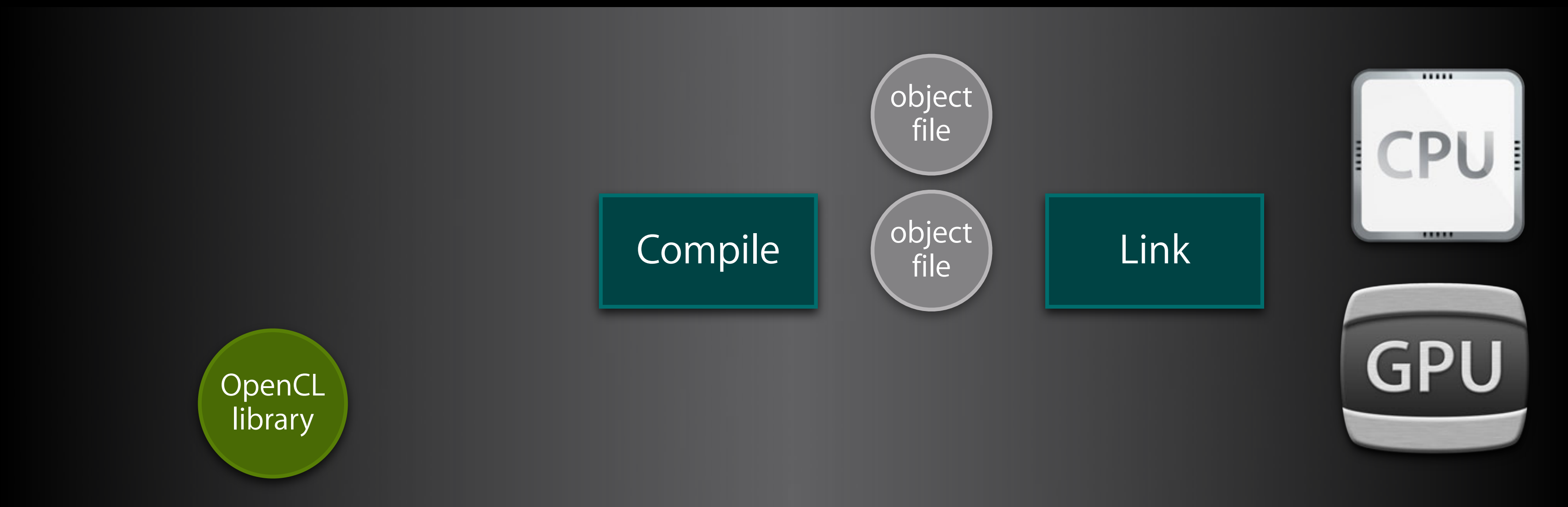
Libraries of OpenCL Kernels

Traditional compile and link model



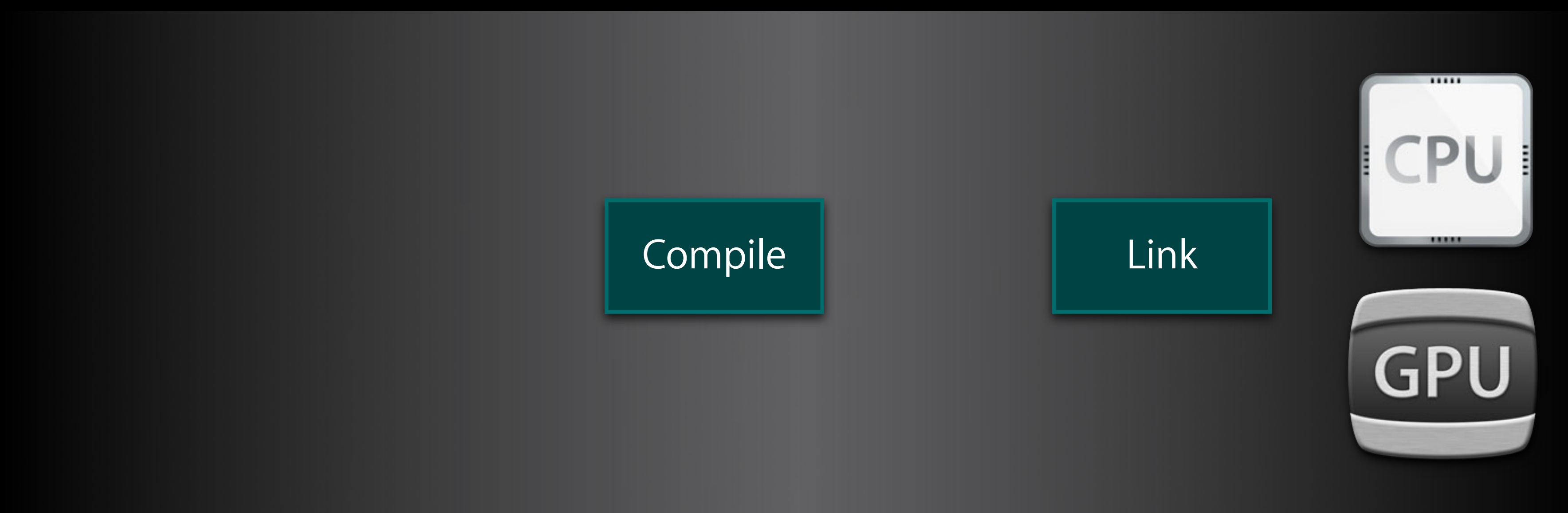
Libraries of OpenCL Kernels

Traditional compile and link model



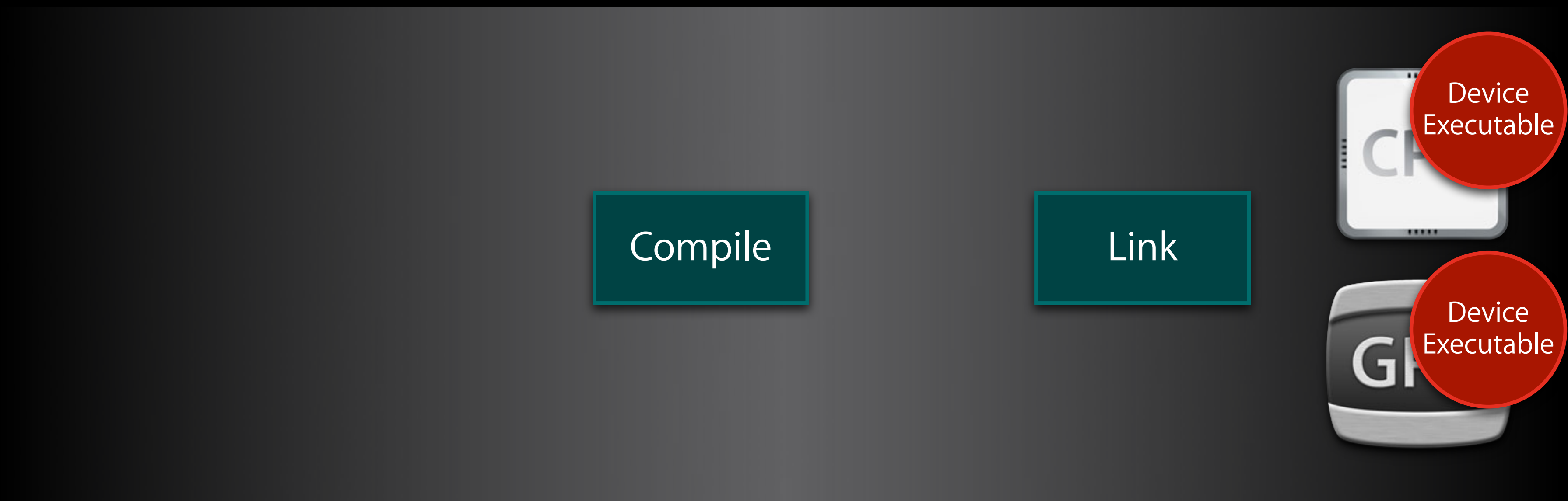
Libraries of OpenCL Kernels

Traditional compile and link model



Libraries of OpenCL Kernels

Traditional compile and link model



printf

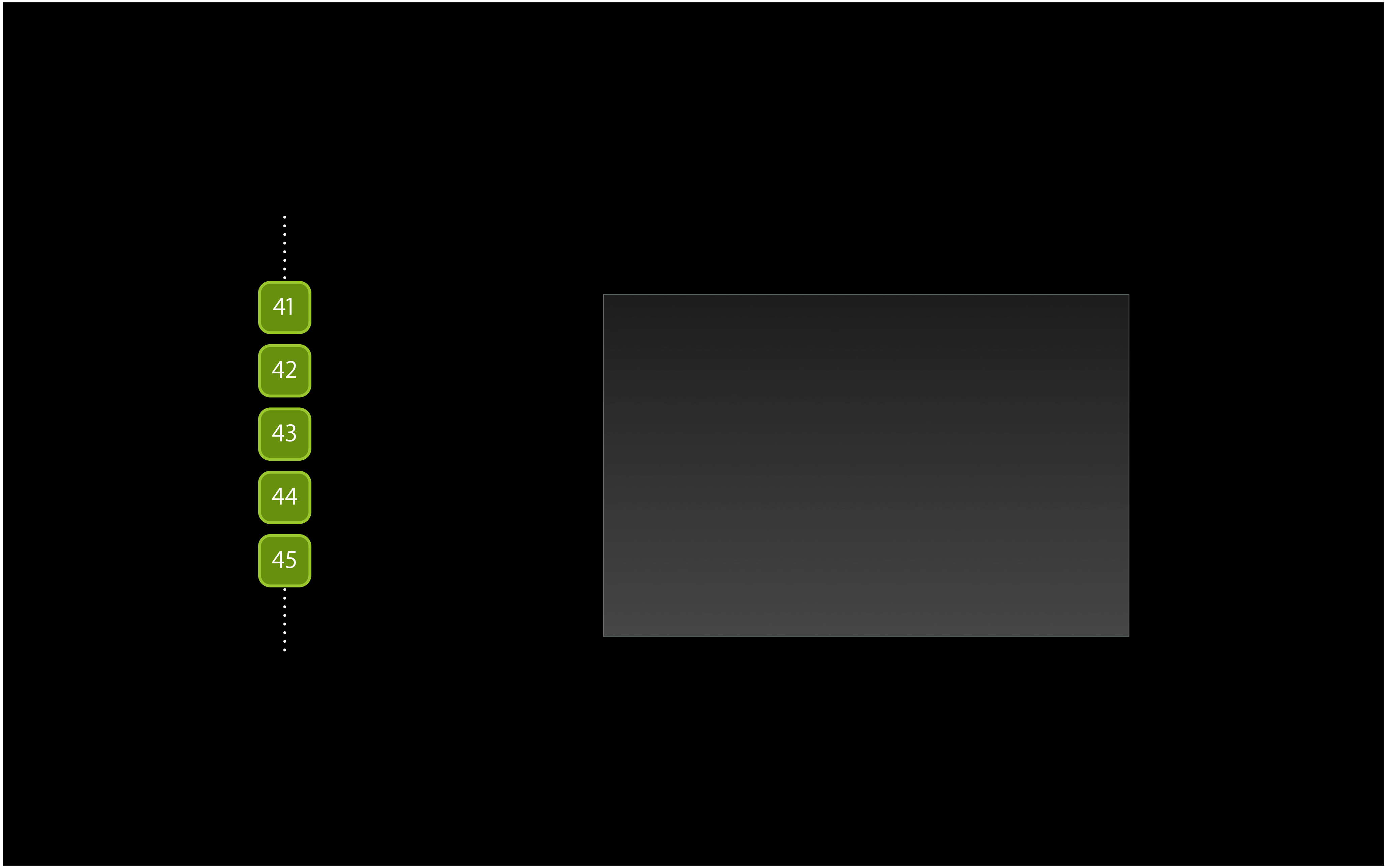
Printing within OpenCL kernel

- Print to stdout
- Output for concurrently executing work-items may not be serial

[redacted]


41
42
43
44
45


```
I index element 113
I index element 111
I index element 112
I index element 115
```


41
42
43
44
45


```
id 43: I index element 113
id 41: I index element 111
id 42: I index element 112
id 45: I index element 115
```


```
id 43: I index element 113
id 41: I index element 111
id 42: I index element 112
id 45: I index element 115
```

printf

Printing within OpenCL kernel

- Supports vector types
  - Examples

```
float4  f = (float4)(100.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);

printf(“f4 = %.2v4f\n”, f);
printf(“uc = %v4hhx\n”, uc);
```

printf

Printing within OpenCL kernel

- Supports vector types
    - Examples

```
float4  f = (float4)(100.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);

printf(“f4 %.2v4f\n”, f);
printf(“uc %v4hhx\n”, uc);
```

printf

Printing within OpenCL kernel

- Supports vector types
  - Examples

```
float4  f = (float4)(100.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);

printf(“f4 = %.2v4f\n”, f);
printf(“uc = %v4hhx\n”, uc);
```

```
f4 = 1.00,2.00,3.00,4.00

uc = 0xfa,0xfb,0xfc,0xfd
```

Overloaded User Functions

New

- Use `__OVERLOAD__` attribute to overload user functions

Overloaded User Functions

- Use __OVERLOAD__ attribute to overload user functions

```
int   graph_int(int *data);

float graph_float(float *data);
```

Overloaded User Functions

New

- Use __OVERLOAD__ attribute to overload user functions

```
int   graph_int(int *data);         int   __OVERLOAD__ graph(int *data);

float graph_float(float *data);     float __OVERLOAD__ graph(float *data);
```

Performance Hints

Help us help you

Performance Hints

Help us help you

- Filling memory

Performance Hints

Help us help you

- Filling memory
  - `clEnqueueFillBuffer`

To fill a buffer with a pattern

Performance Hints

Help us help you

- Filling memory
  - `clEnqueueFillBuffer`
  - `clEnqueueFillImage`

To fill an image with a color

Performance Hints

Help us help you

- Filling memory
  - `clEnqueueFillBuffer`
  - `clEnqueueFillImage`
- Memory access

New

Performance Hints

Help us help you

New

- Filling memory
  - `clEnqueueFillBuffer`
  - `clEnqueueFillImage`
- Memory access
  - `CL_MEM_HOST_WRITE_ONLY`

If you are **only writing** from host

Performance Hints

Help us help you

- Filling memory
  - `clEnqueueFillBuffer`
  - `clEnqueueFillImage`
- Memory access
  - `CL_MEM_HOST_WRITE_ONLY`
  - `CL_MAP_WRITE_INVALIDATE_REGION`

If **overwriting** a mapped region from host

Deprecated APIs

Use new APIs

New

Deprecated APIs	New APIs
clCreateImage2D, clCreateImage3D	clCreateImage
clCreateFromTexture2D, clCreateFromTexture3D	clCreateFromTexture
clEnqueueMarker, clEnqueueBarrier, clEnqueueWaitForEvents	clEnqueueMarkerWithWaitList, clEnqueueBarrierWithWaitList

Intel Auto-Vectorizer

Sion Berkowits
Senior SW Engineer
Intel Corporation

Agenda

- Recap: What is OpenCL Auto-Vectorizer
- What is new in Mountain Lion
- How the new Auto-Vectorizer works
- Tips for OpenCL programmers
- Demo

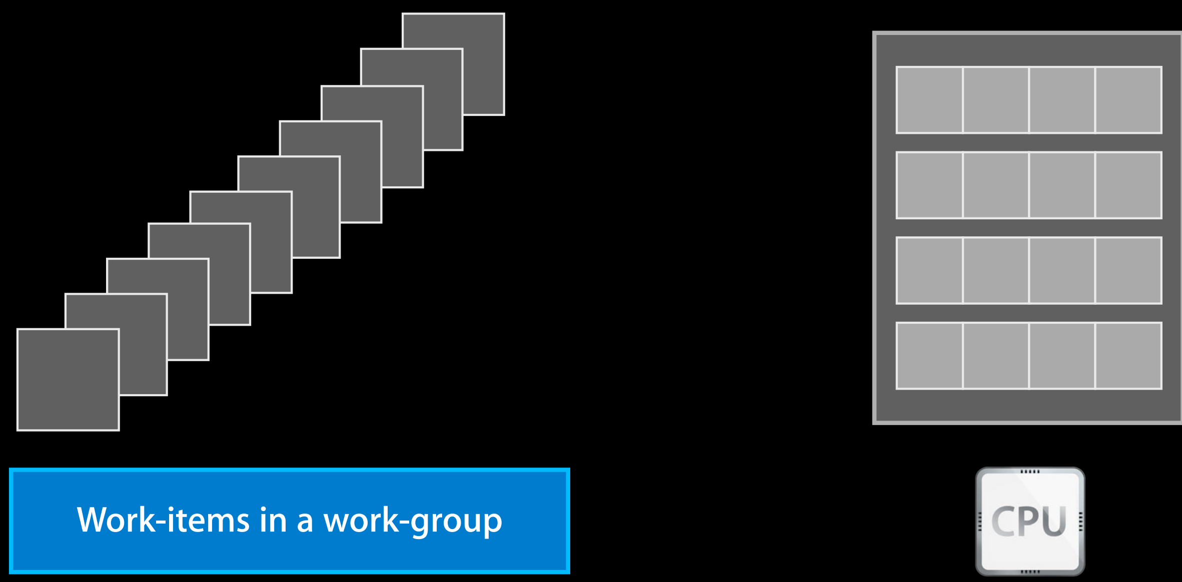
Developing OpenCL on the CPU

- Optimal performance on the CPU requires target-specific optimizations
- Code loses simplicity
- Code loses performance portability

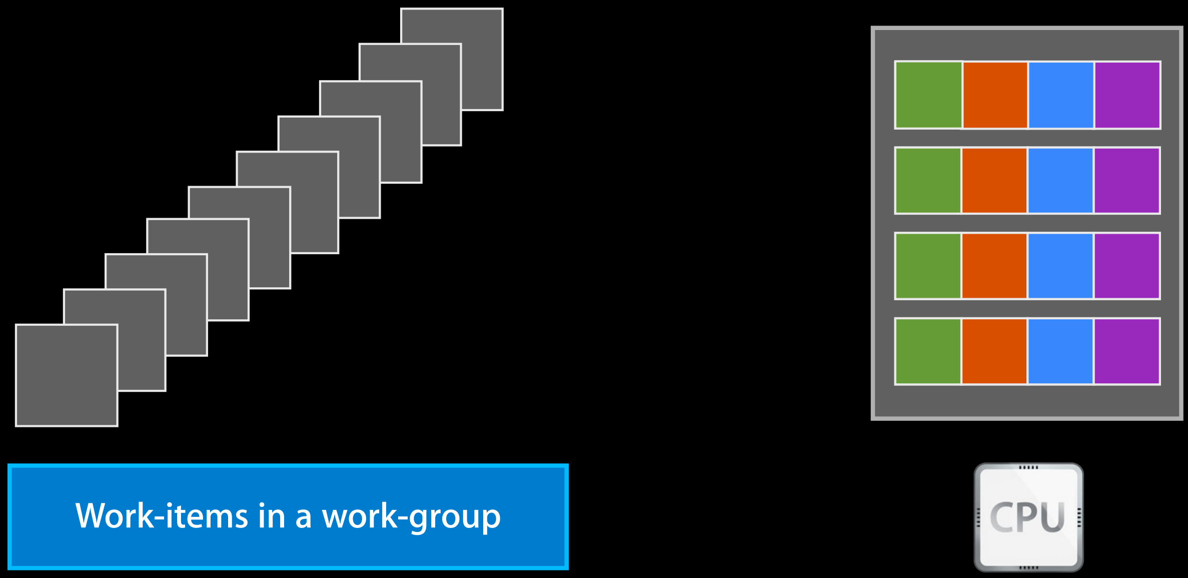
The OpenCL Auto-Vectorizer

- An OpenCL CPU compiler optimization
- Introduced in Lion
- Performance utilizes the vector registers (SIMD) in the CPU

What Is OpenCL Auto-Vectorizer



What Is OpenCL Auto-Vectorizer



Auto-Vectorizer Features

- Runs by default when compiling kernels for the CPU
- Does not require user modifications
- Works in the presence of scalar and vector operations
- Only works in the absence of control flow

The New Auto-Vectorizer

Introduced in Mountain Lion

New

- Added support for vectorizing kernels with control flow
- Automatically optimize code for the underlying CPU architecture
- Significant speedup on kernels, compared to non-vectorized code
  - Speedup may be reduced, due to control flow in kernel

How the Auto-Vectorizer Works

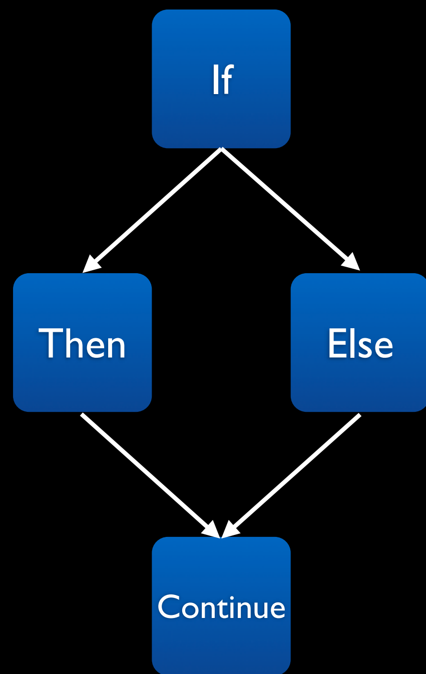
Dealing with control flow

If-Then-Else Blocks

```
...
...
if (condition)
{
    do_some_work();
}
else
{
    do_some_other_work();
}
...
...
```

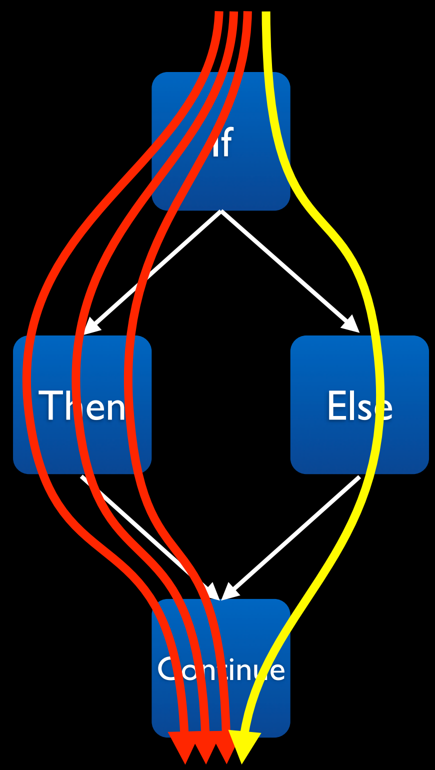
If-Then-Else Blocks

- The problem: Different work-items may choose different code paths
- How to pack instructions from several work-items into a single vector instruction?



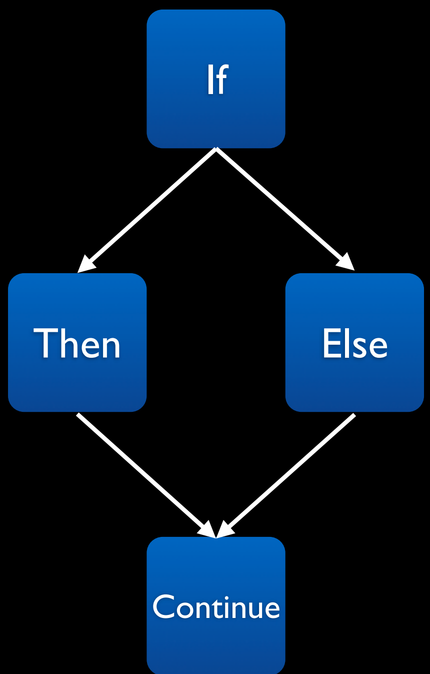
If-Then-Else Blocks

- The problem: Different work-items may choose different code paths
- How to pack instructions from several work-items into a single vector instruction?



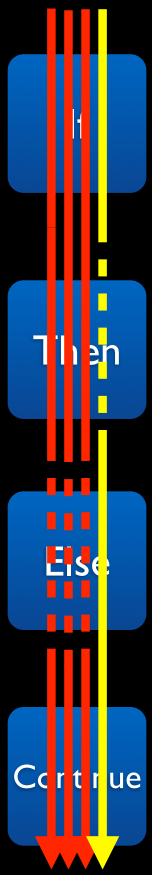
Vectorizing If-Then-Else Blocks

- Auto-Vectorized code should execute both sides of control flow statement
- Control flow is serialized by Auto-Vectorizer
- Packed work-items go through both **Then** and **Else** code
- Unneeded calculations are disposed
  - Side effects (such as stores) are masked



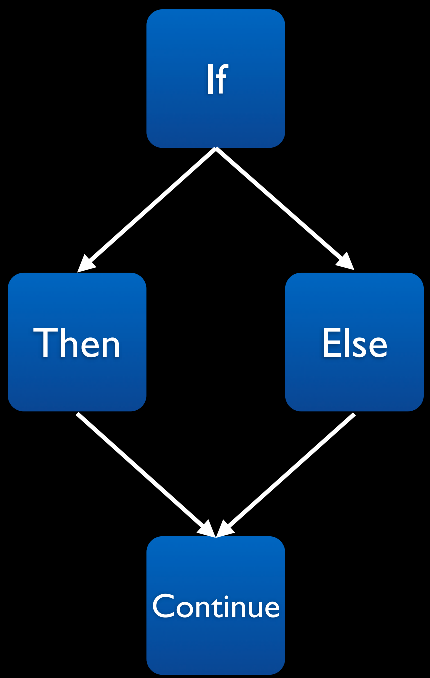
Vectorizing If-Then-Else Blocks

- Auto-Vectorized code should execute both sides of control flow statement
- Control flow is serialized by Auto-Vectorizer
- Packed work-items go through both **Then** and **Else** code
- Unneeded calculations are disposed
    - Side effects (such as stores) are masked



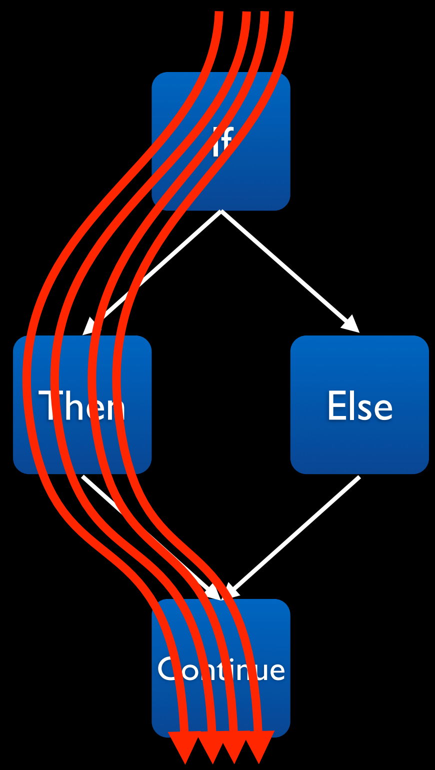
Uniform Control Flow

- In some cases, control flow is uniform
  - All work-items choose the same path in the control flow
  - For example, when **If** condition is a constant
- In such cases, no modifications are done to the control flow code



Uniform Control Flow

- In some cases, control flow is uniform
  - All work-items choose the same path in the control flow
  - For example, when **If** condition is a constant
- In such cases, no modifications are done to the control flow code



Loop Blocks

```
...
...
while (some_condition)
{
    do_work();
    ...
    ...
    do_more_work();
}
...
...
```

Loop Blocks

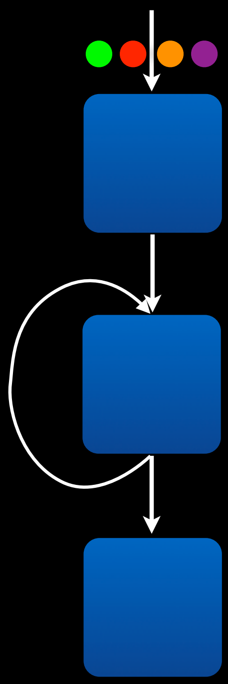
- The problem: Number of loop iterations may depend on work-item ID
- For example

```
for (int i=0 ; i<get_global_id(0) ; ++i)
{
    ...
    do_something();
}
```

- How to pack instructions, when every work-item may require a different amount of loop iterations?

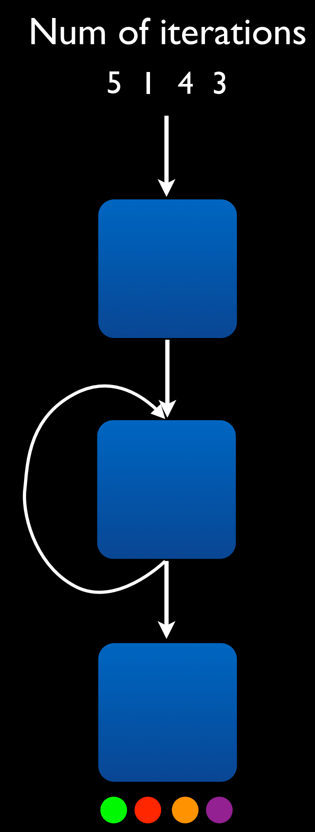
Vectorizing Loop Blocks

- Auto-Vectorized code iterates over loop for all packed work-items
- The loop is iterated until all participating work-items finish executing their respective iterations



Vectorizing Loop Blocks

- Auto-Vectorized code iterates over loop for all packed work-items
- The loop is iterated until all participating work-items finish executing their respective iterations



Tips for OpenCL Programming

Getting the most from the Auto-Vectorizer

Memory Access in Control Flow

- Memory accesses in control flow must be masked
  - Some work-items may need to avoid them
- Adds overhead for such memory accesses

Memory Access in Control Flow

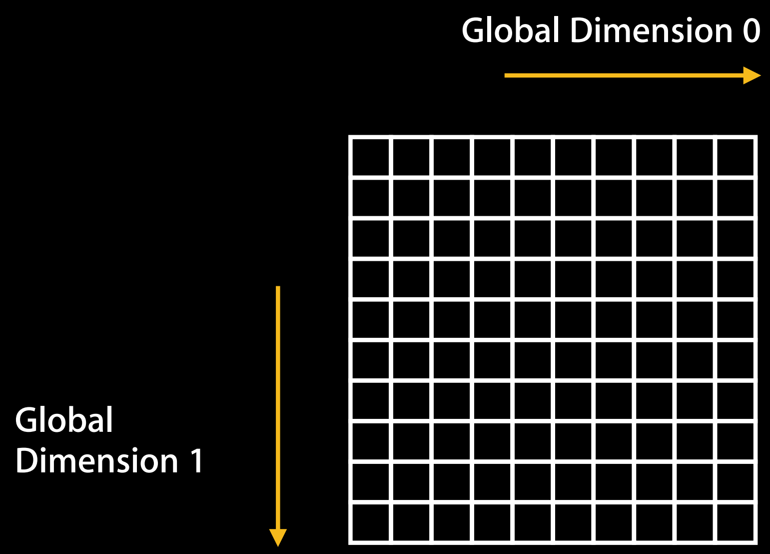
- Recommendation: Move memory access out of control flow, if possible

```
if (cond) {
    a[index] = 1;
} else {
    a[index] = 2;
}
```

```
if (cond) {
    temp = 1;
} else {
    temp = 2;
}
a[index] = temp;
```

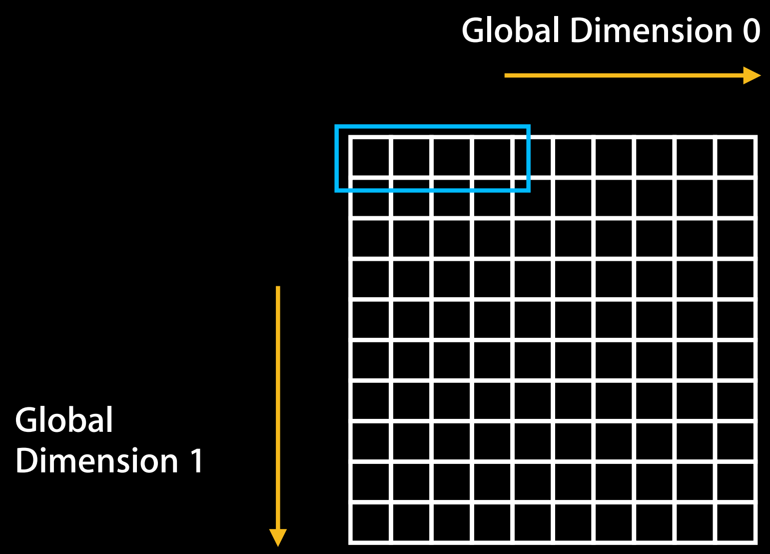
Row-Wise Data Access

- The Auto-Vectorizer packs together work-items which have consecutive indices in global dimension 0



Row-Wise Data Access

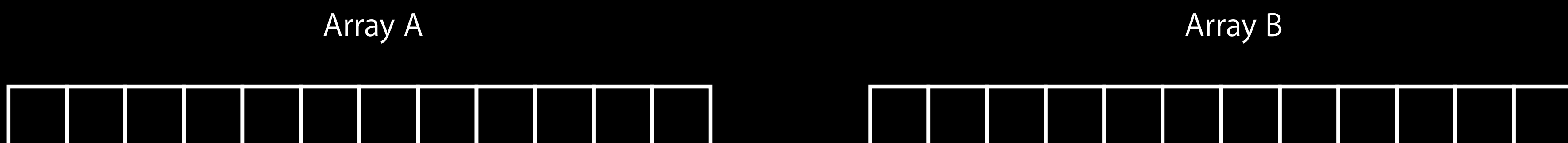
- The Auto-Vectorizer packs together work-items which have consecutive indices in global dimension 0



Row-Wise Data Access

- When accessing array elements, it is preferred to access consecutive array elements in consecutive work-items

```
int tid0 = get_global_id(0);
int arrayA_val = A[tid0];
int arrayB_val = B[tid0 * some_constant];
```



Row-Wise Data Access

- When accessing array elements, it is preferred to access consecutive array elements in consecutive work-items

```
int tid0 = get_global_id(0);
int arrayA_val = A[tid0];
int arrayB_val = B[tid0 * some_constant];
```

Demo

Summary

- The Auto-Vectorizer optimizes your OpenCL kernels on the CPU, providing significant speedup
- Works “behind the scenes”, requiring no user modifications to run
- In Mountain Lion, the Auto-Vectorizer supports kernels with complex control flow

OpenCL Kernel Tuning

From C code to optimized OpenCL code

Eric Bainville
OpenCL Engineer

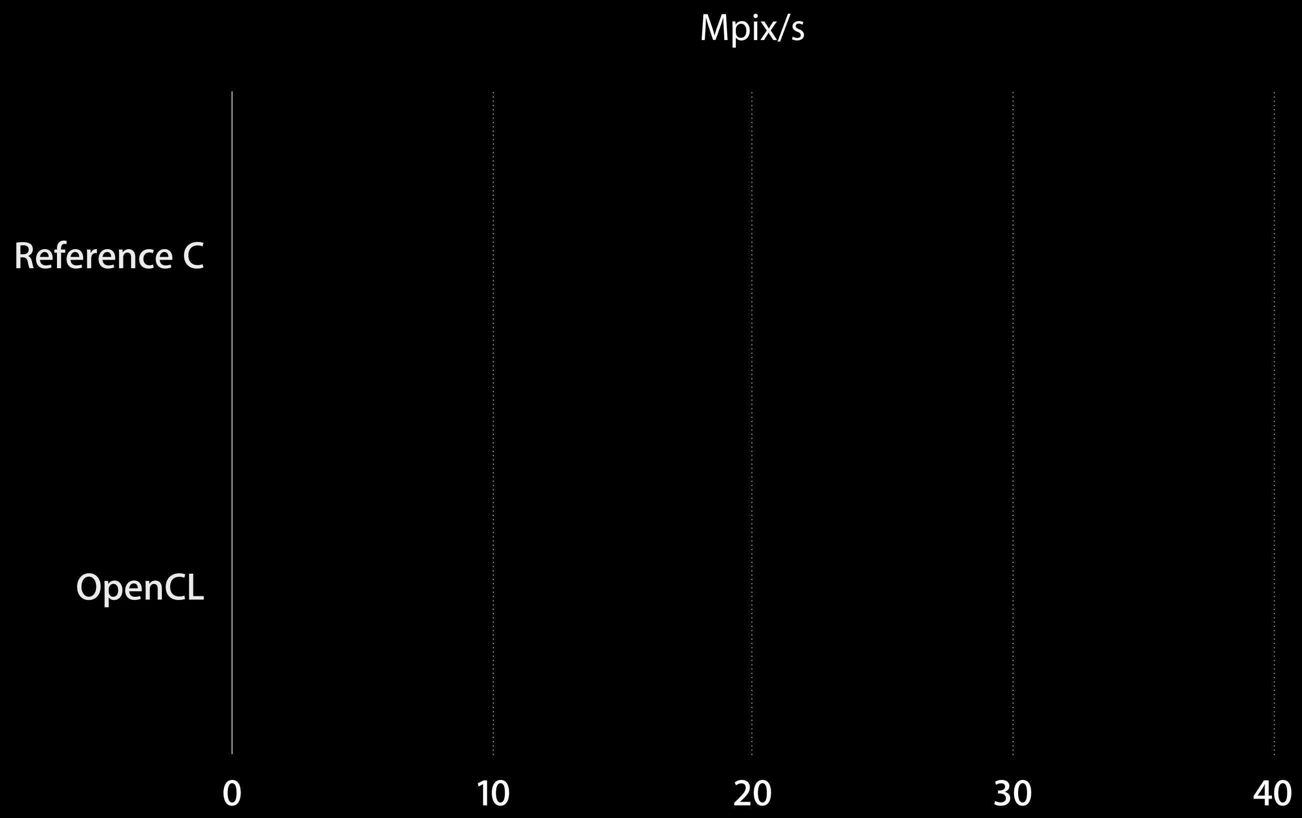
OpenCL Kernel Tuning

Why?

Gaussian Blur, 16 Mpix Image

OpenCL Kernel Tuning

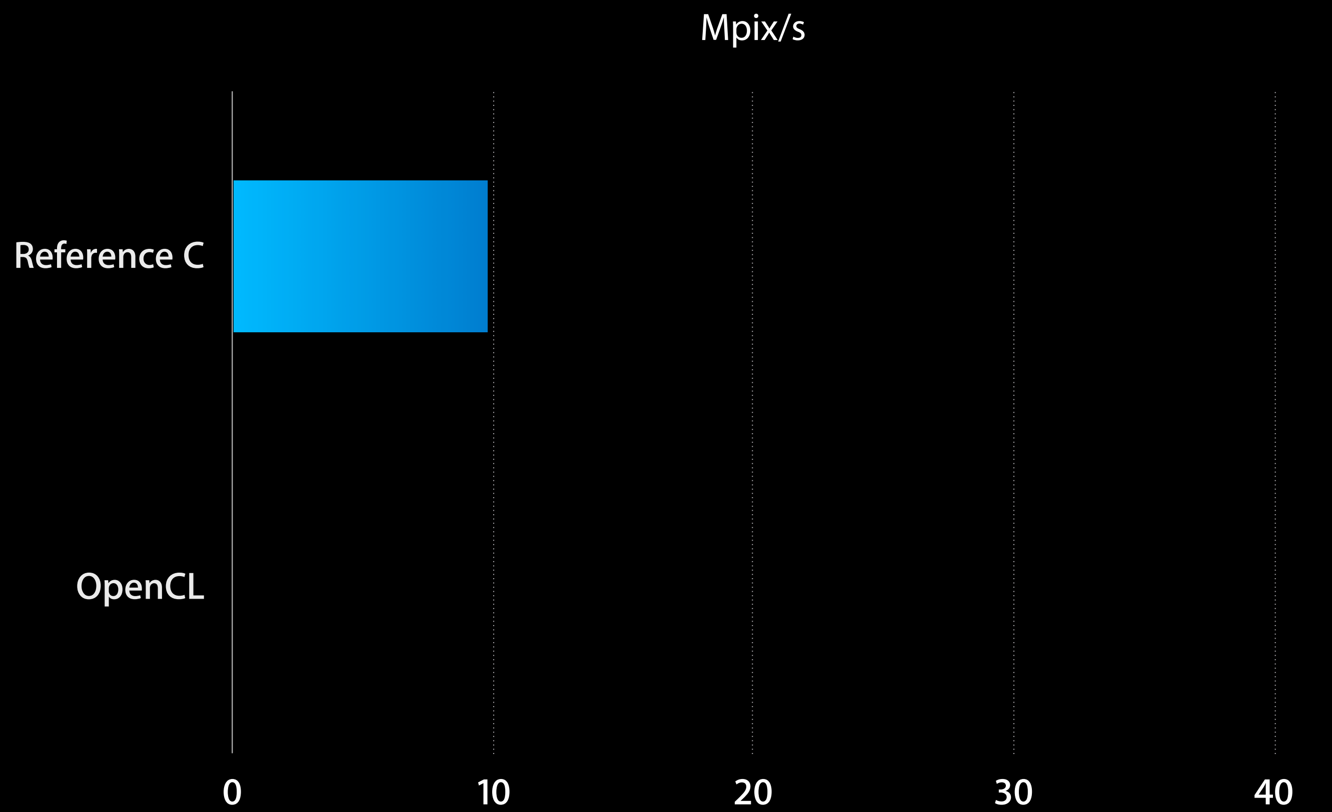
Why?



Gaussian Blur, 16 Mpix Image

OpenCL Kernel Tuning

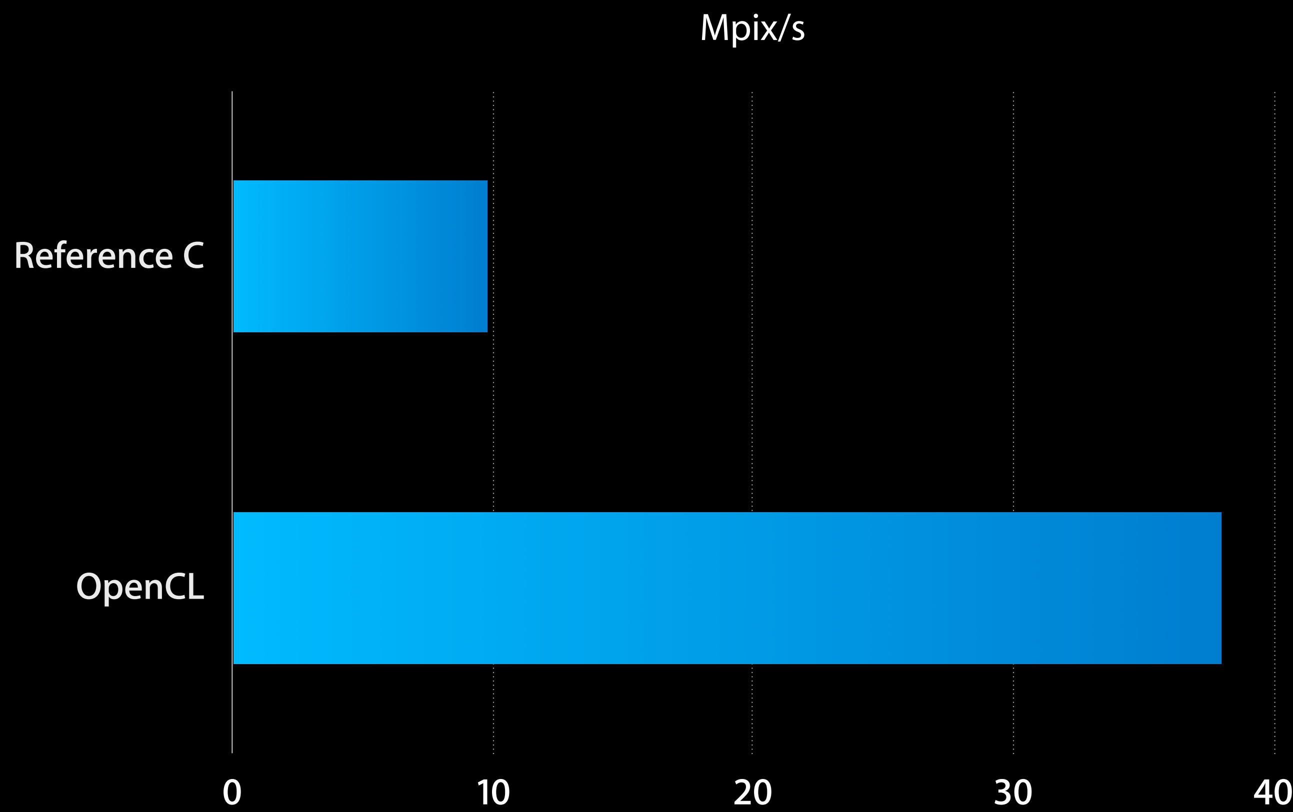
Why?



Gaussian Blur, 16 Mpix Image

OpenCL Kernel Tuning

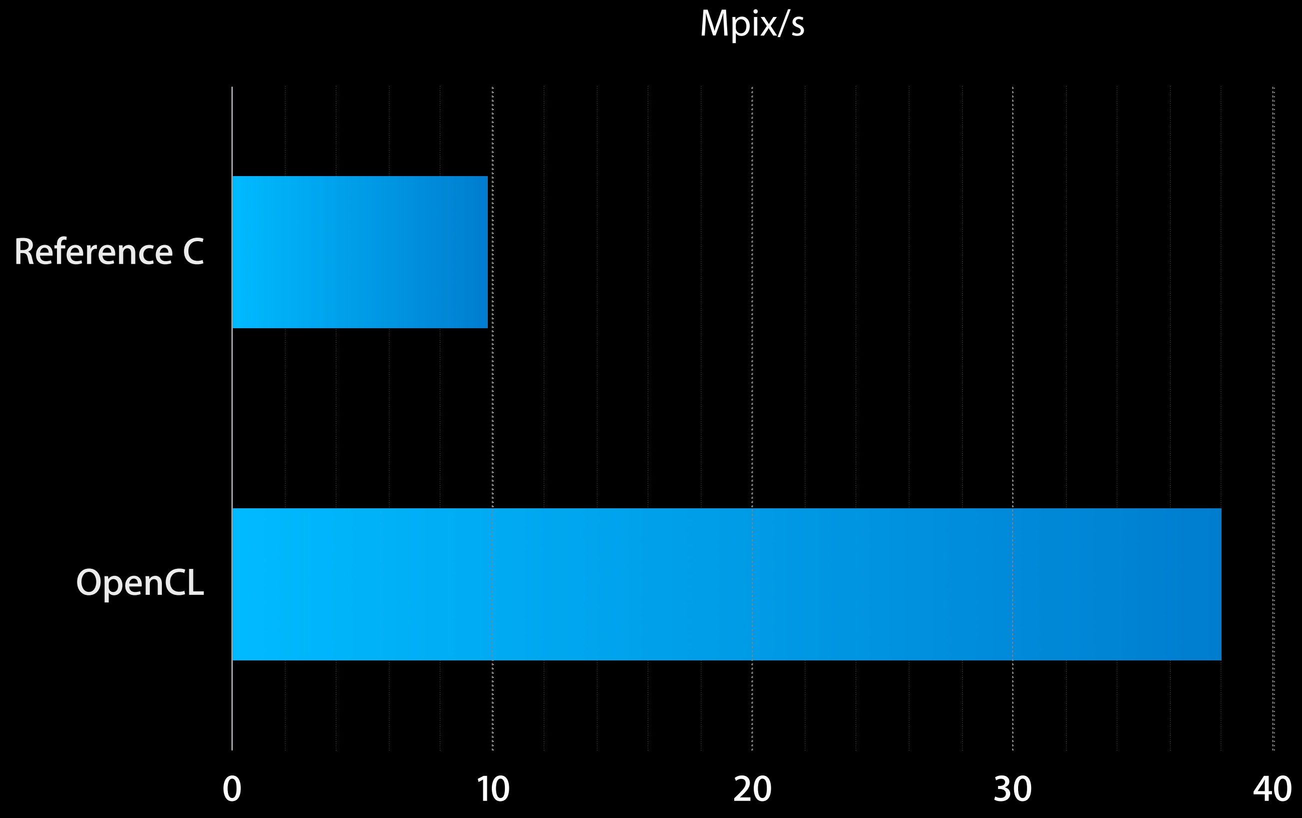
Why?



Gaussian Blur, 16 Mpix Image

OpenCL Kernel Tuning

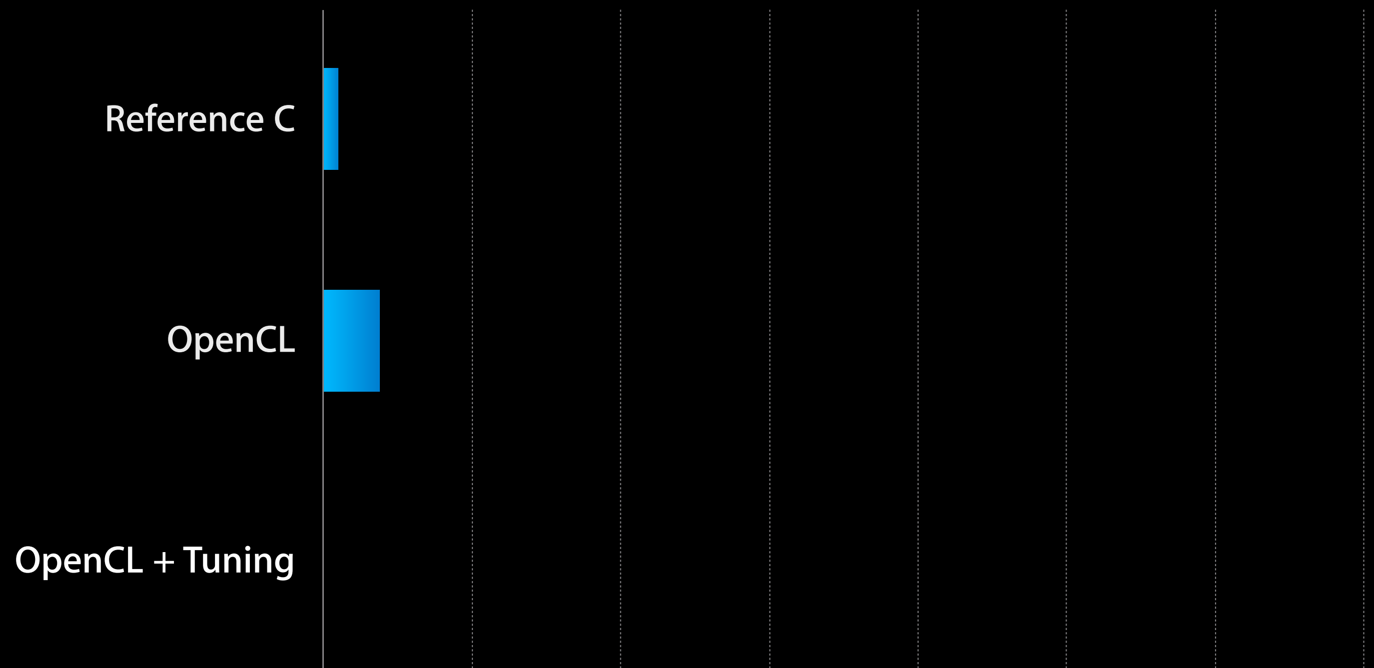
Why?



Gaussian Blur, 16 Mpix Image

OpenCL Kernel Tuning

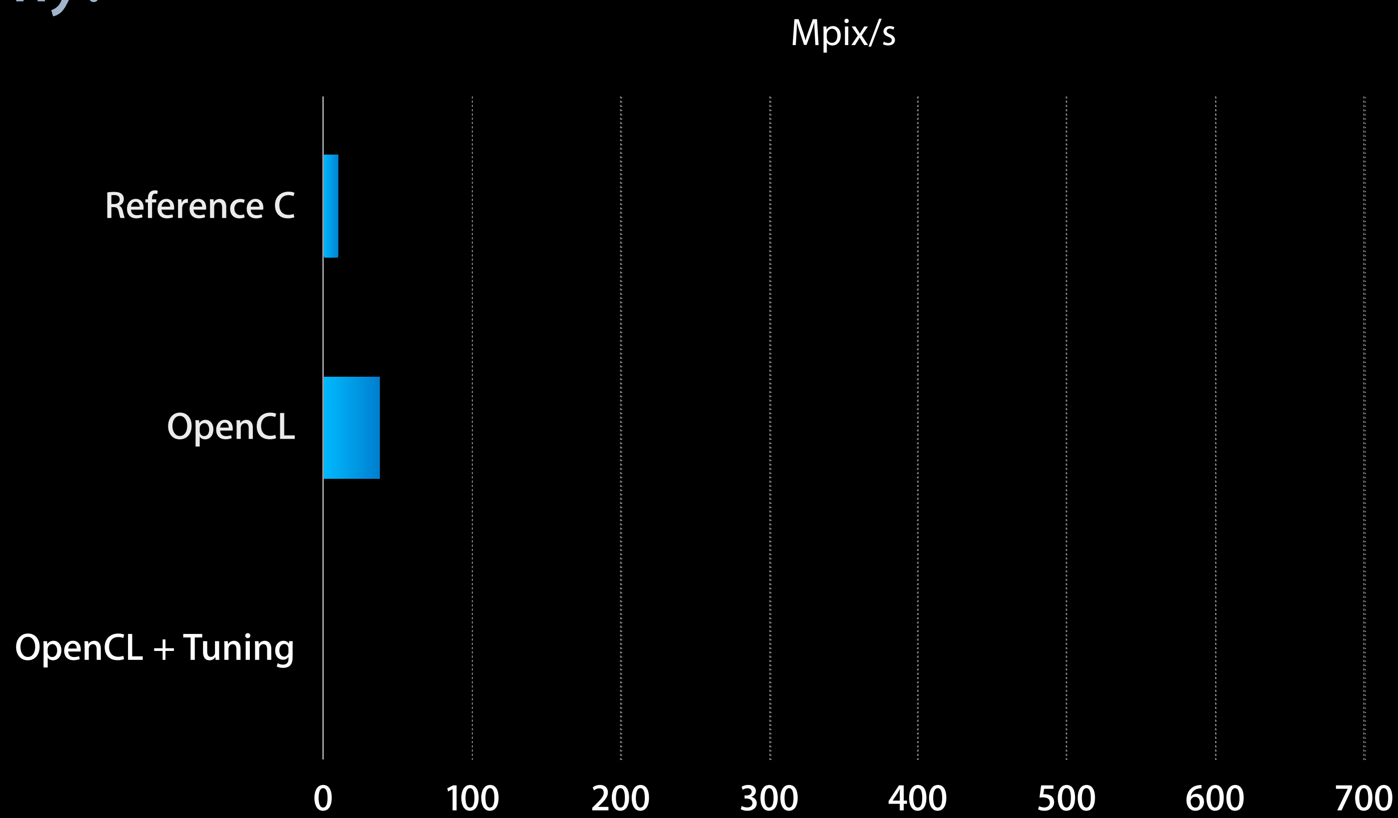
Why?



Gaussian Blur, 16 Mpix Image

OpenCL Kernel Tuning

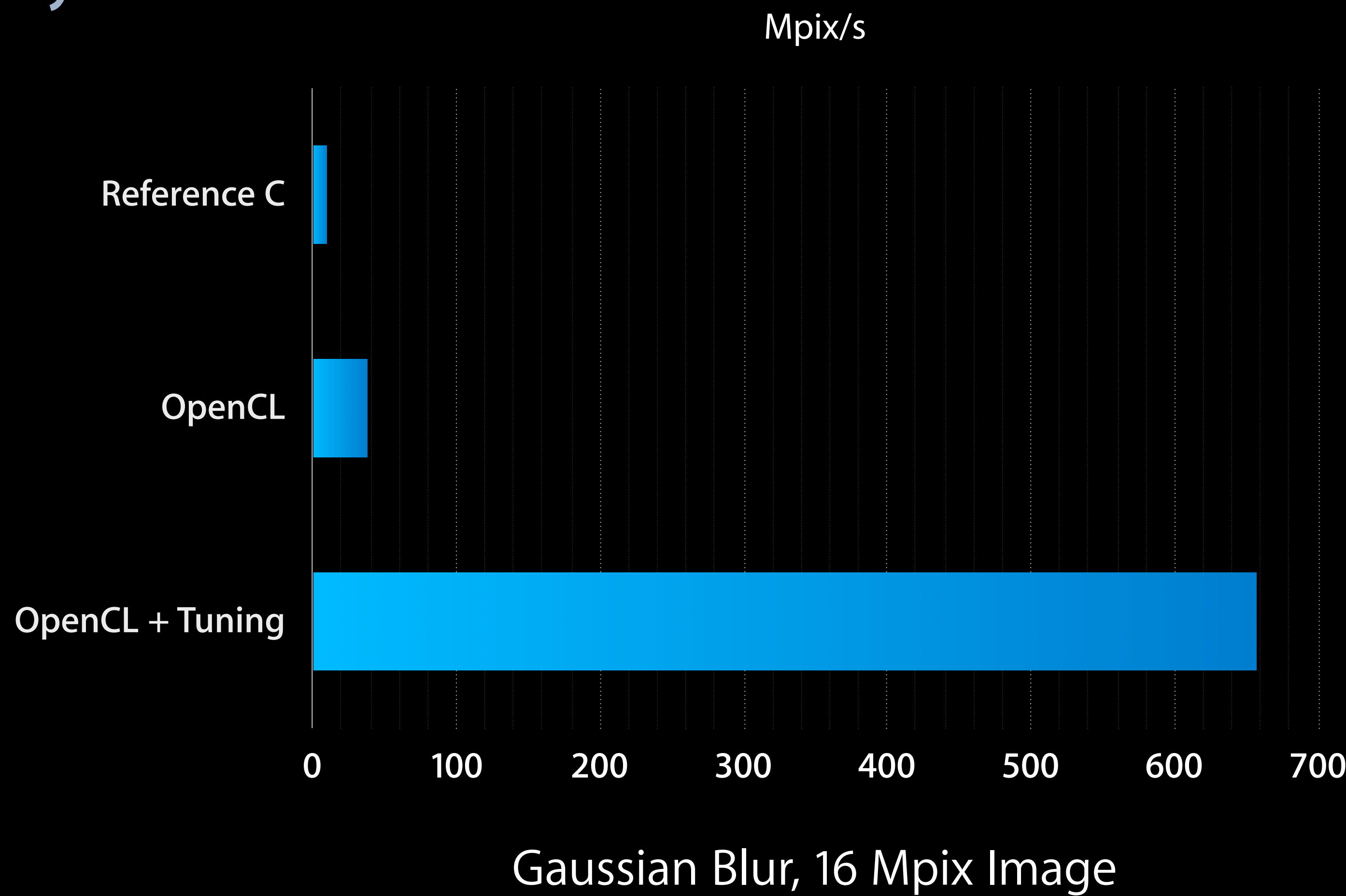
Why?



Gaussian Blur, 16 Mpix Image

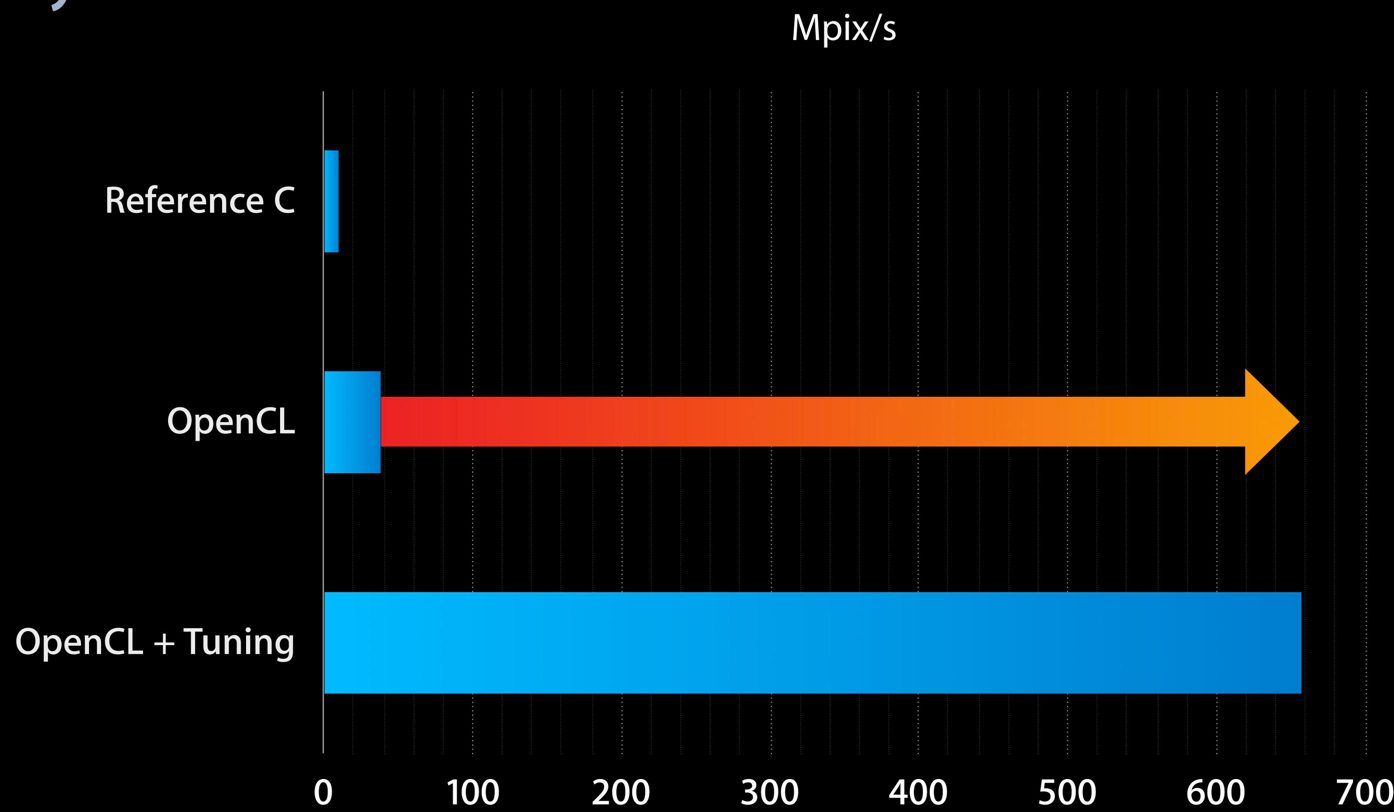
OpenCL Kernel Tuning

Why?



OpenCL Kernel Tuning

Why?



Gaussian Blur, 16 Mpix Image

Kernel Tuning BASIC(s)

```
 1 Choose the right algorithm
10 Write the code
20 Benchmark
21 if “fast enough” goto DONE
30 Identify bottlenecks
40 Find solution/workaround
50 goto 10
```

What Is “Fast Enough”?

And how to choose the right algorithm

Benchmarks

- `copy` kernel

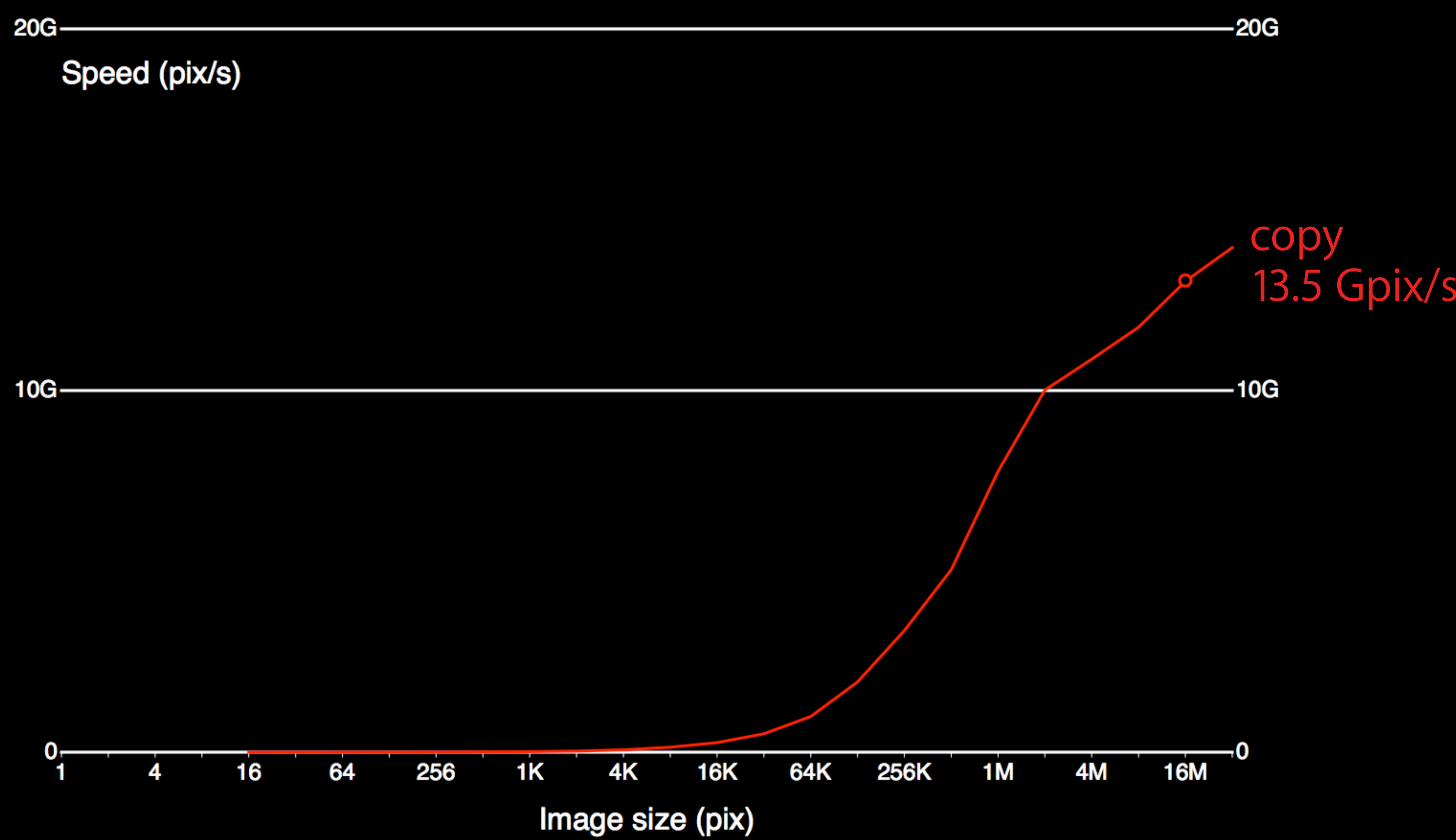
```
kernel void copy(global const float * in,
                 global float * out,
                 int w,int h)
{
  int x = get_global_id(0);  // (x,y) = pixel to process
  int y = get_global_id(1);  // in this work item

  out[x+y*w] = in[x+y*w];    // Load + Store
}
```

Benchmarks

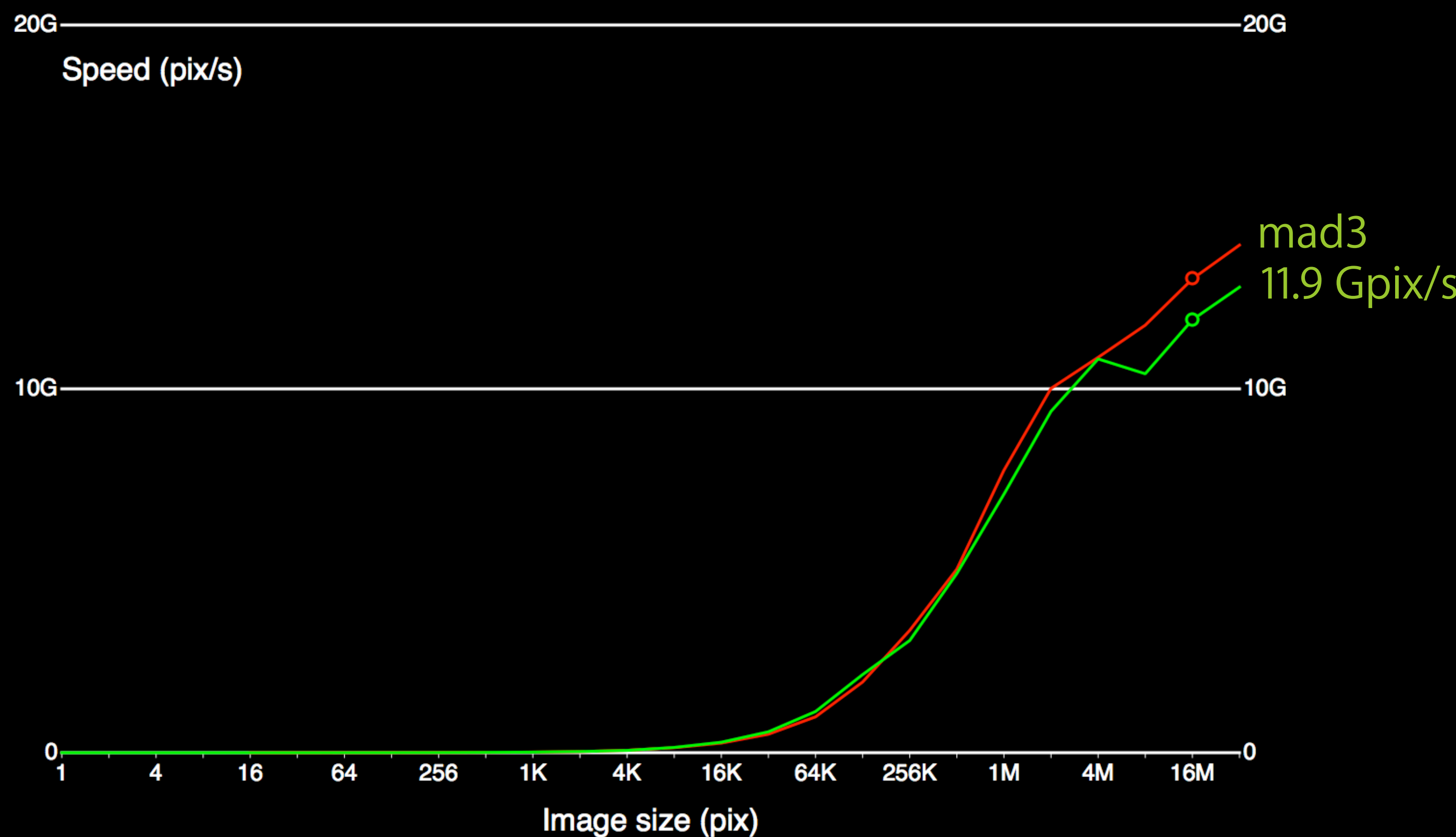
- `mad` kernels = `copy` + N flops

```
kernel void mad3(global const float * in,
                 global float * out,
                 int w,int h)
{
  int x = get_global_id(0);
  int y = get_global_id(1);

  float a = in[x+y*w];             // Load
  float b = 3.9f * a * (1.0f-a);   // 3 floating point ops
  out[x+y*w] = b;                  // Store
}
```

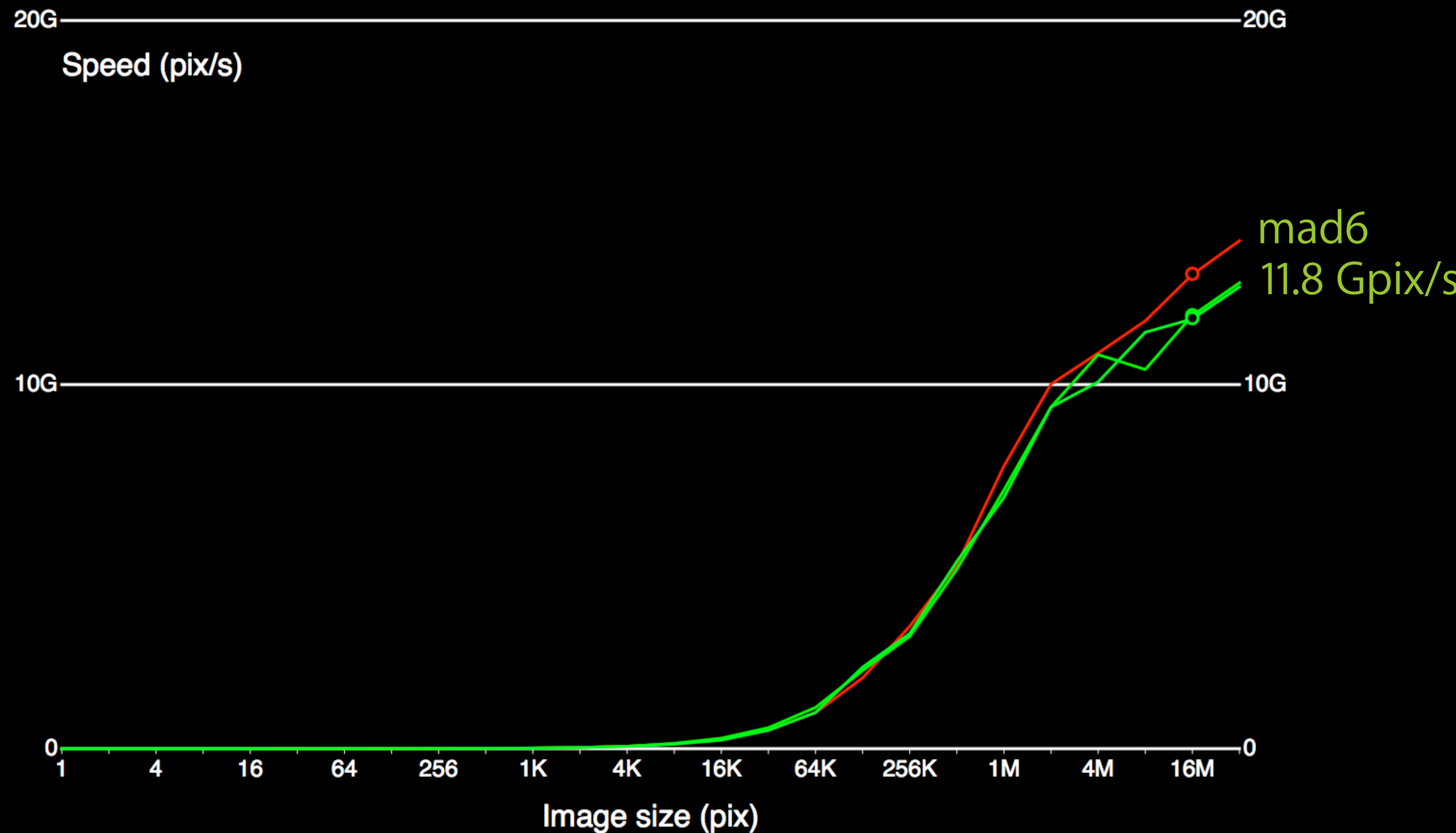
Benchmarks: copy



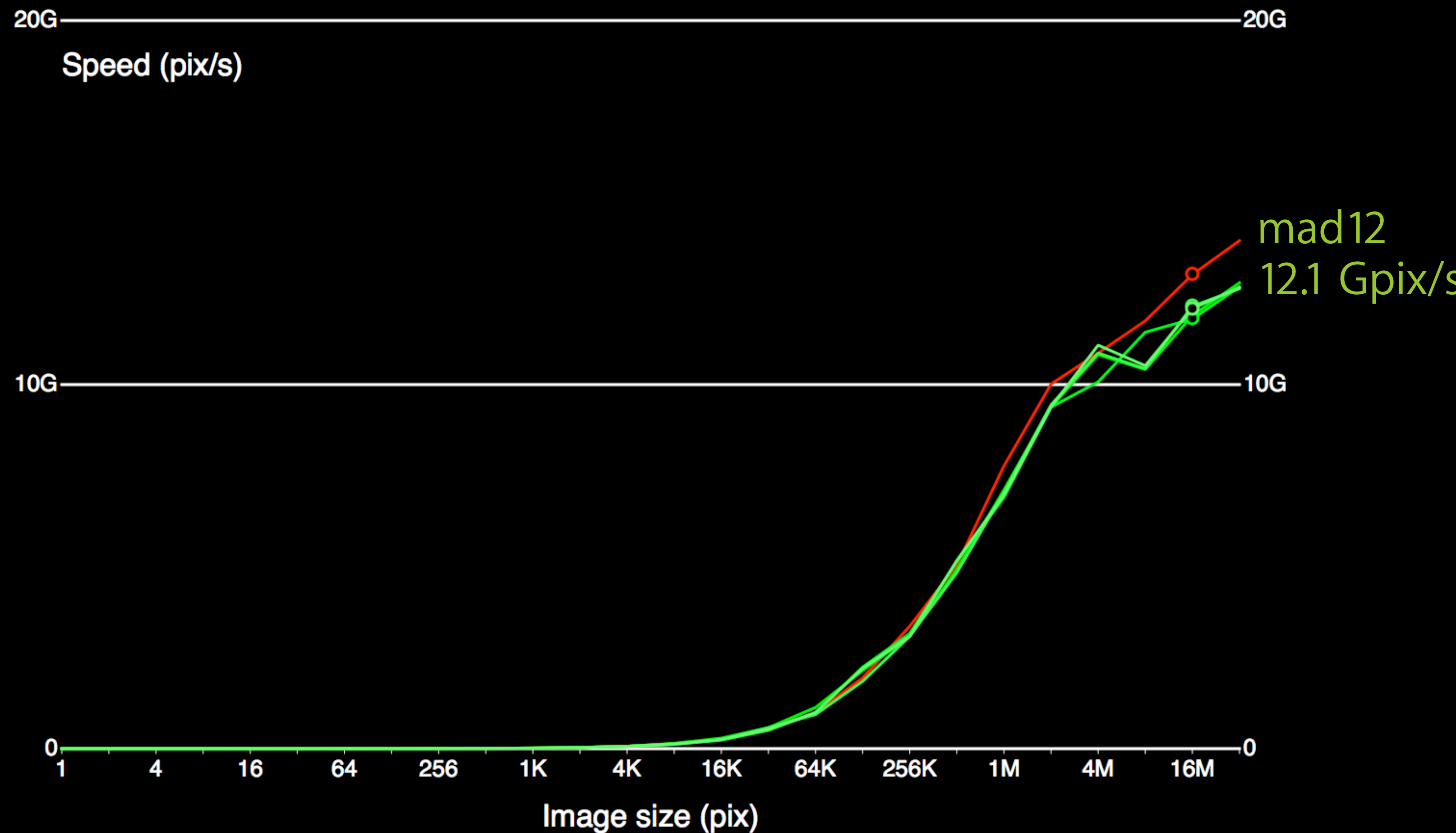
Benchmarks: copy, mad3



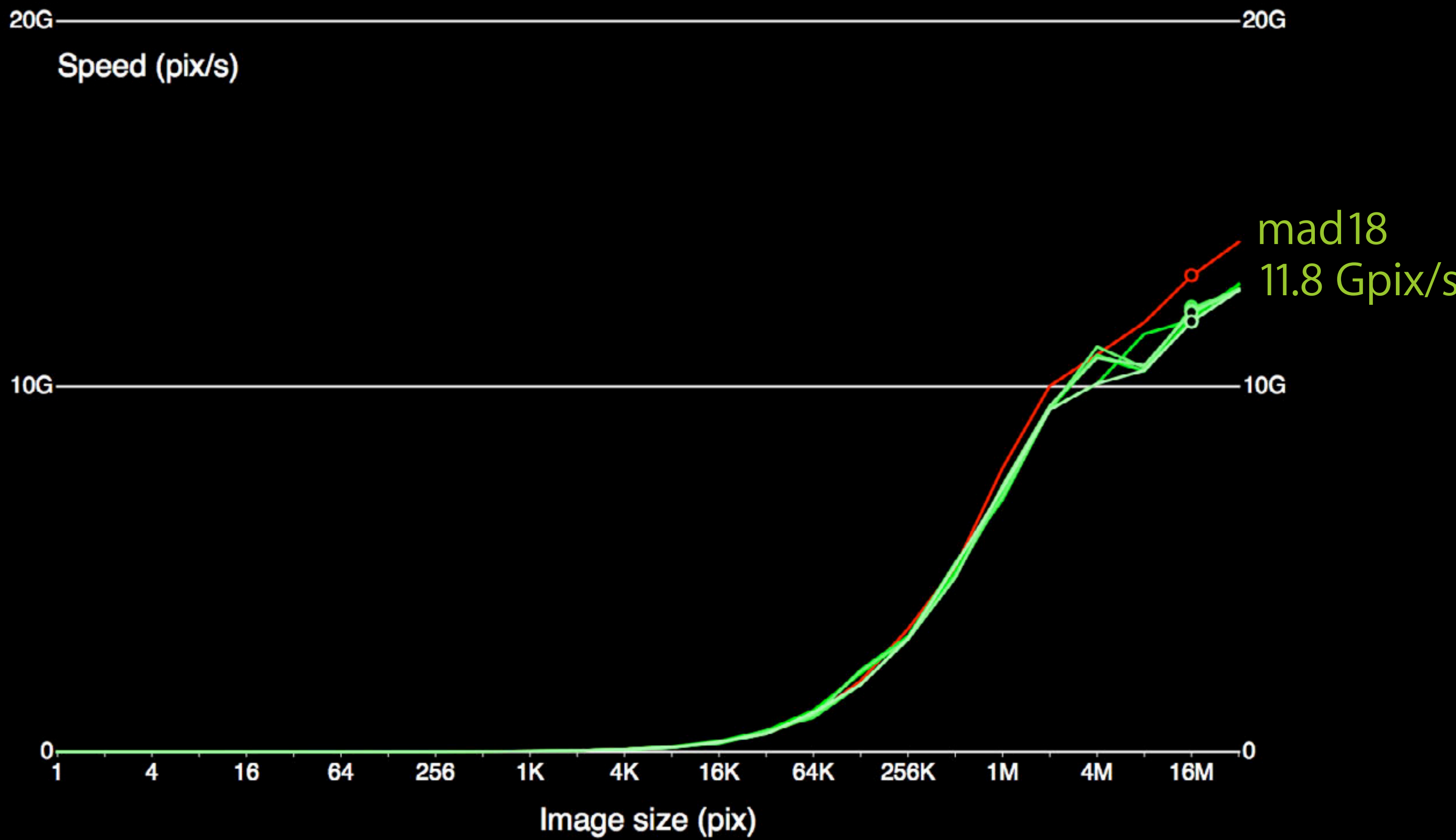
Benchmarks: …, mad6



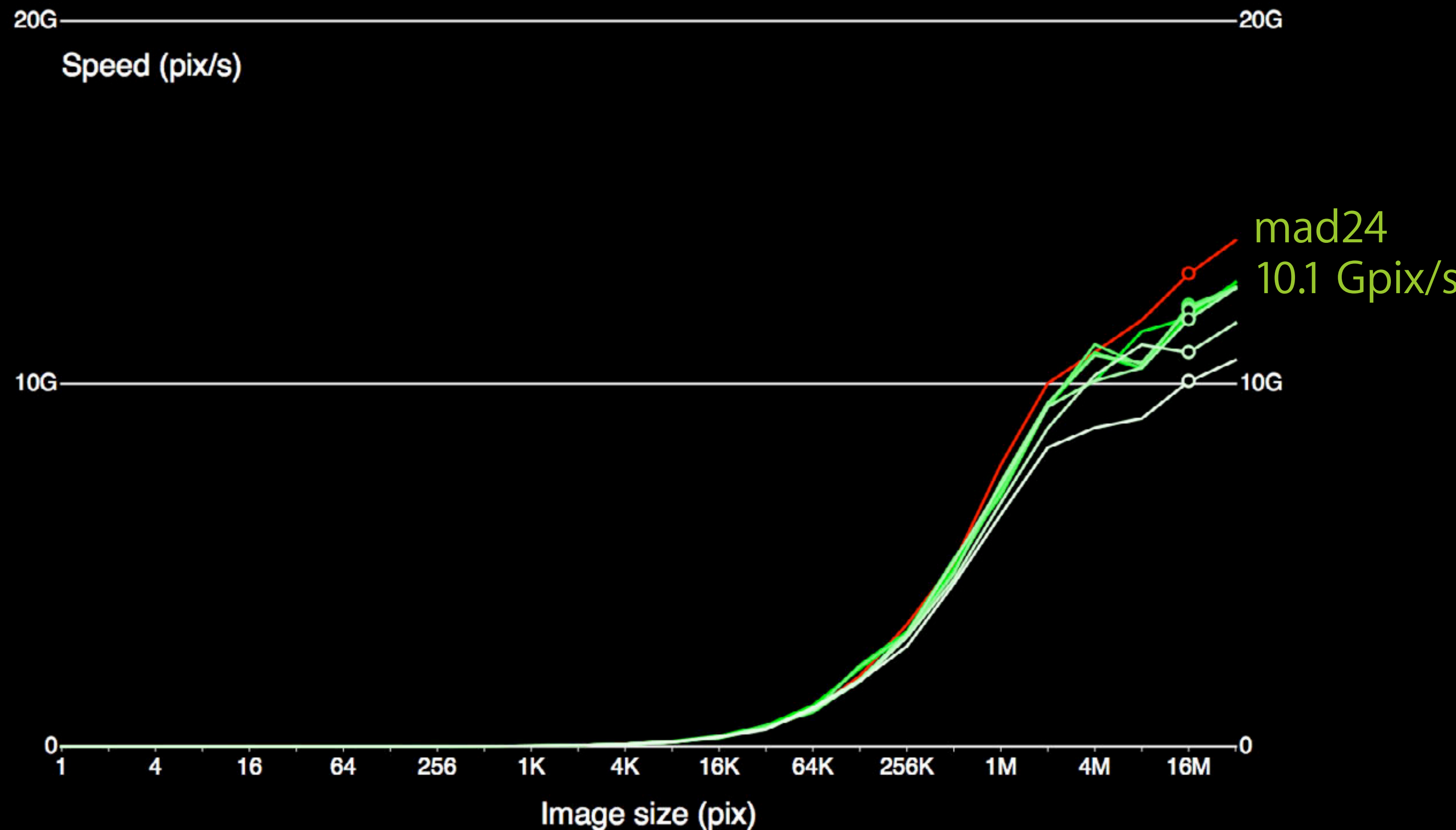
Benchmarks: …, mad12



Benchmarks: …, mad18



Benchmarks: …, mad24



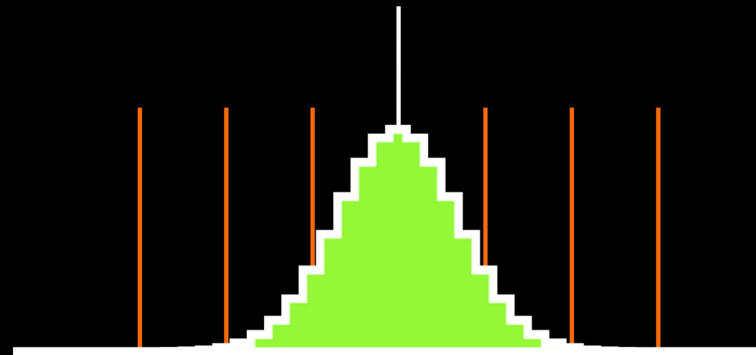
How to Choose an Algorithm?

- Parallel
- Minimize memory usage
- Maximize Compute/Memory ratio
- Benchmarks ➔ performance estimate

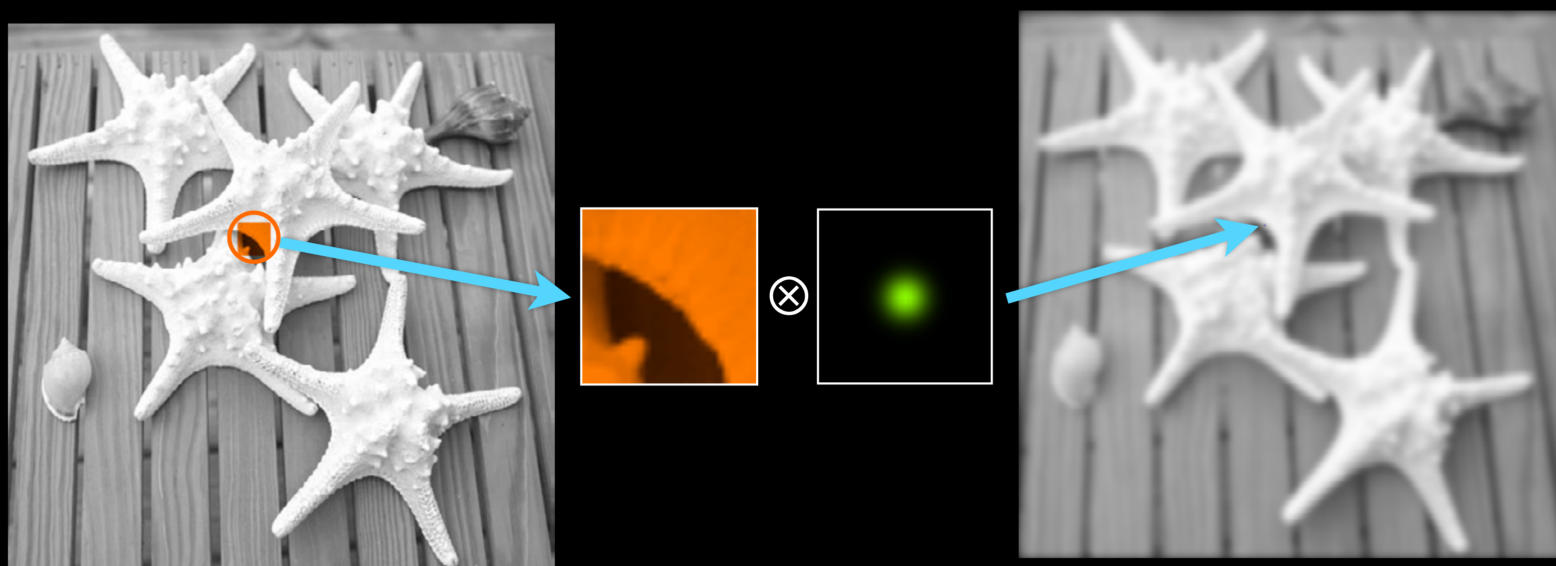
Kernel Tuning BASIC(s)

```
 1 Choose the right algorithm
10 Write the code
20 Benchmark
21 if “fast enough” goto DONE
30 Identify bottlenecks
40 Find solution/workaround
50 goto 10
```

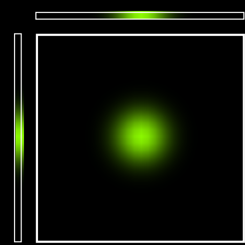
Classic 2D Convolution



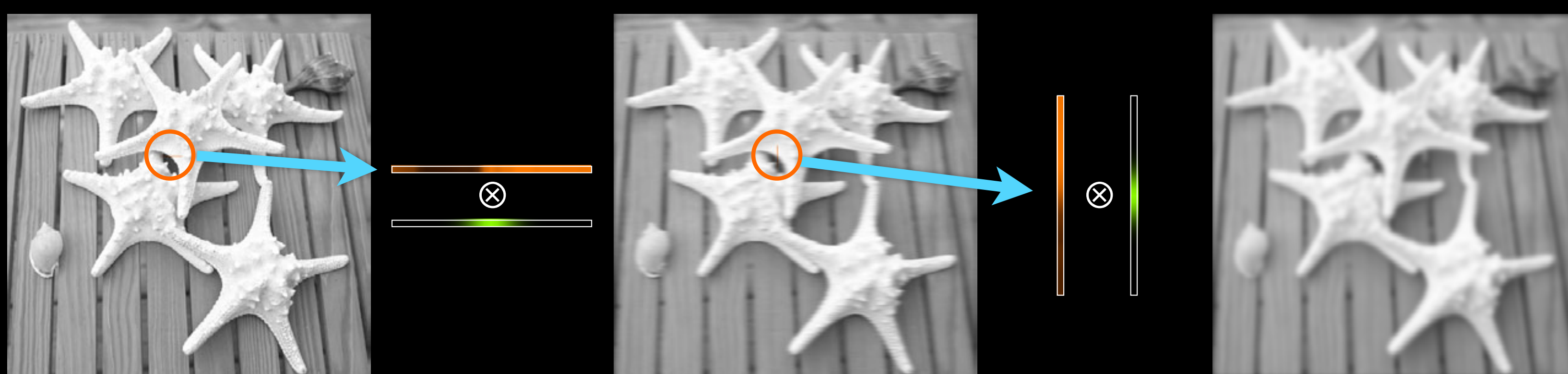
- Gaussian filter range: -3σ..3σ
- σ=5 ➔ 31x31 convolution kernel
- 962 read/write + 1922 flops / pixel



Separable 2D Convolution

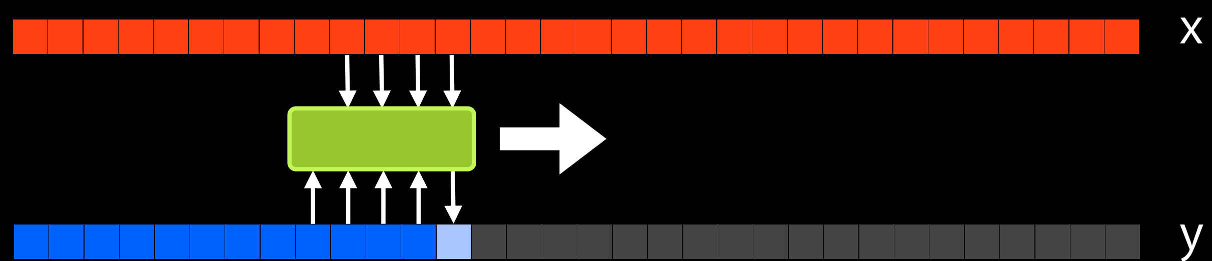


- $K_{2D}(x,y)=K_{1D}(x).K_{1D}(y)$
- 2 passes   H + V
- 64 read/write + 124 flops   / pixel

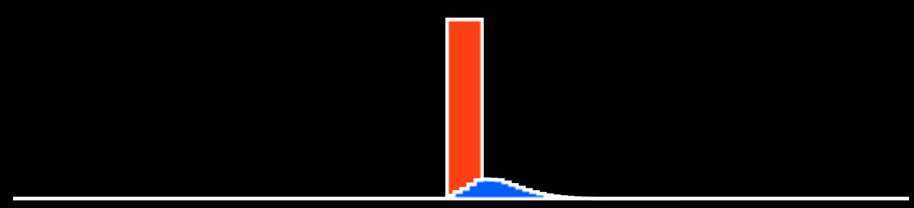


Recursive Gaussian Filter

1^{st} pass

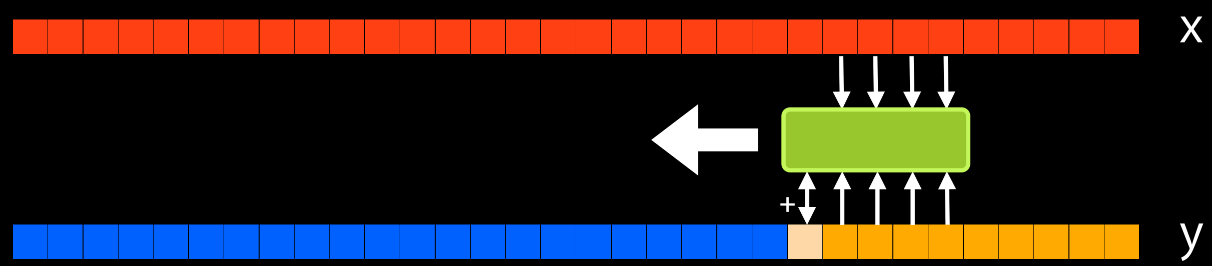


$$y_i \leftarrow a_0 x_i + a_1 x_{i-1} + a_2 x_{i-2} + a_3 x_{i-3}$$
$$+ b_1 y_{i-1} + b_2 y_{i-2} + b_3 y_{i-3} + b_4 y_{i-4}$$



Recursive Gaussian Filter

2nd pass



$$y_i \leftarrow y_i + c_1 x_{i+1} + c_2 x_{i+2} + c_3 x_{i+3} + c_4 x_{i+4}$$
$$+ d_1 y_{i+1} + d_2 y_{i+2} + d_3 y_{i+3} + d_4 y_{i+4}$$

Recursive Gaussian Filter

- 4 passes H→ + H← + V↓ + V↑
- 10 read/write + 64 flops per pixel

Recursive Gaussian Filter

- 4 passes H→ + H← + V↓ + V↑
- 10 read/write + 64 flops per pixel

Comparison

Algorithm	Memory (float R+W)	Compute (flops)	C / M Ratio	Estimate (Mpix/s)
Copy	2	0	0	14,200
2D Convolution	962	1,922	2	30
Separable Convolution	64	124	2	440
Recursive Gaussian	10	64	6	2,840

Comparison

Algorithm	Memory (float R+W)	Compute (flops)	C / M Ratio	Estimate (Mpix/s)
Copy	2	0	0	14,200
2D Convolution	962	1,922	2	30
Separable Convolution	64	124	2	440
Recursive Gaussian	10	64	6	$\boxed{2,840}$

Recursive Gaussian on GPU

10 Write the code

Kernel: rgH

```
// One work item per output row
kernel void rgH(global const float * in,global float * out,int w,int h)
{
  int y = get_global_id(0);      // Row to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2 = o3 = o4 = 0.0f;
  for (int x=0;x<w;x++)
  {
    float i0 = in[x+y*w];                             // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;  // Compute new output
    out[x+y*w] = o0;                                  // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

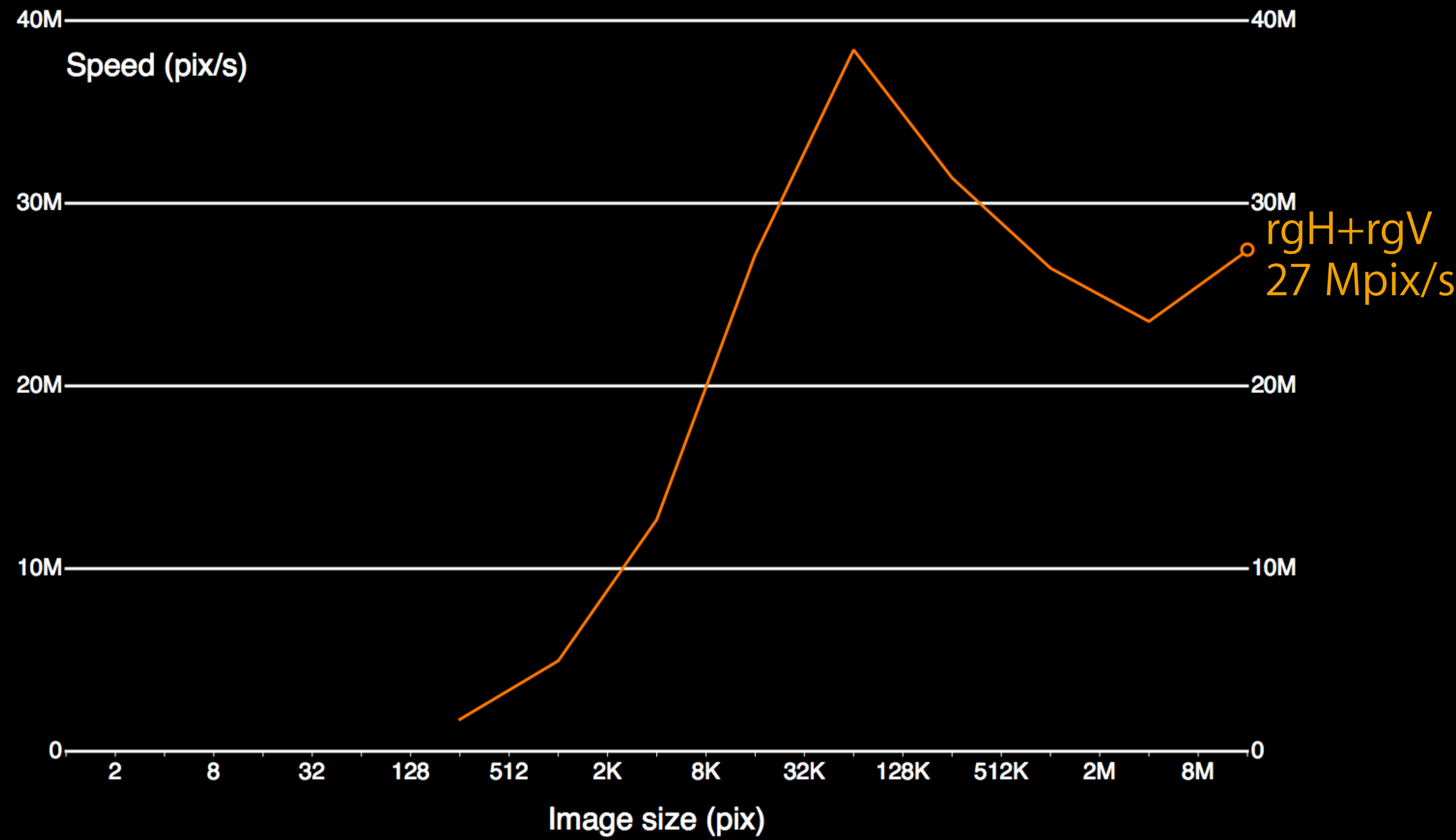
Kernel: rgV

```
// One work item per output column
kernel void rgV(global const float * in,global float * out,int w,int h)
{
  int x = get_global_id(0);      // Column to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2 = o3 = o4 = 0.0f;
  for (int y=0;y<h;y++)
  {
    float i0 = in[x+y*w];        // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;             // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

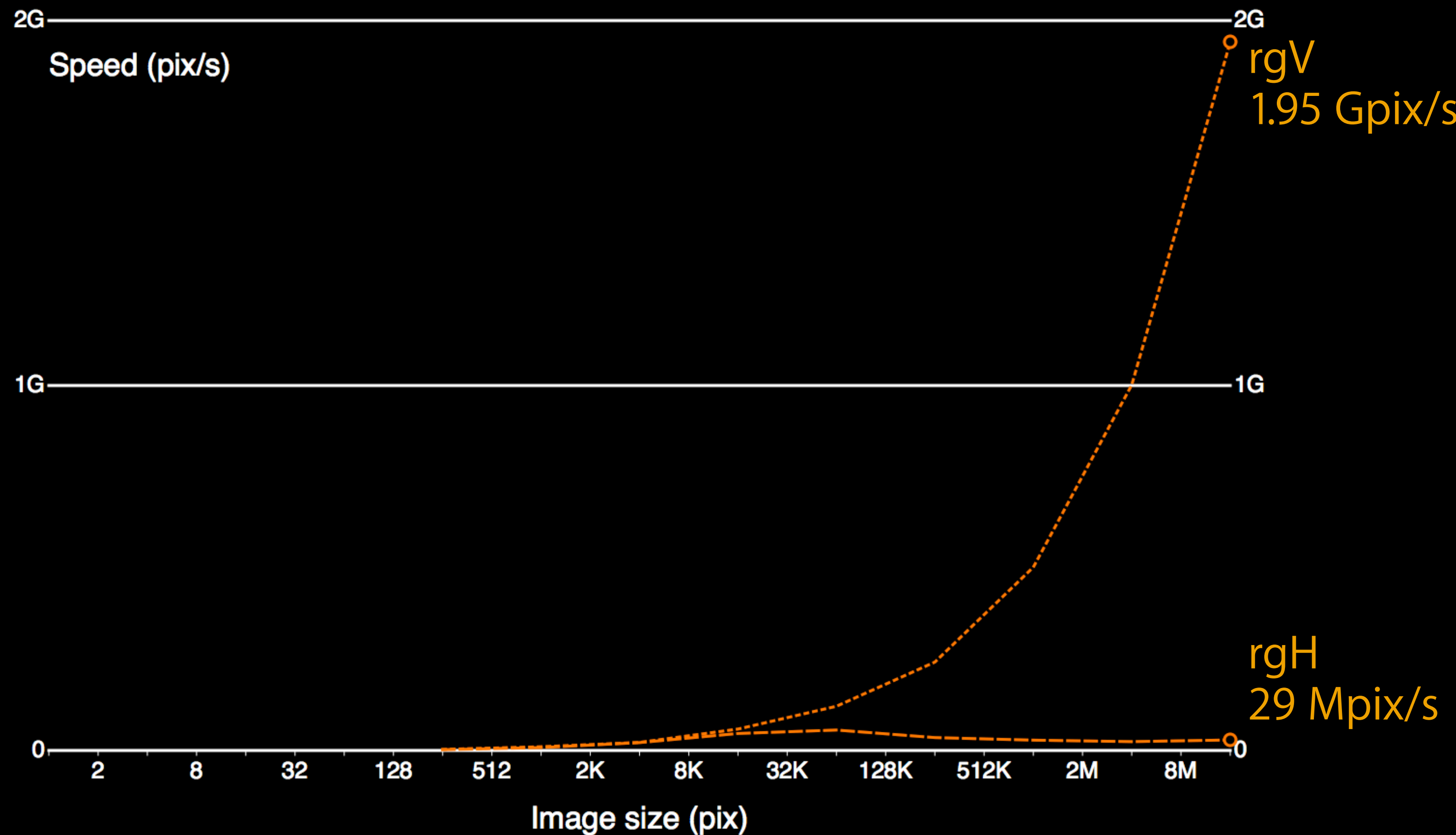
Kernel Tuning BASIC(s)

```
 1 Choose the right algorithm
10 Write the code
20 Benchmark
21 if “fast enough” goto DONE
30 Identify bottlenecks
40 Find solution/workaround
50 goto 10
```

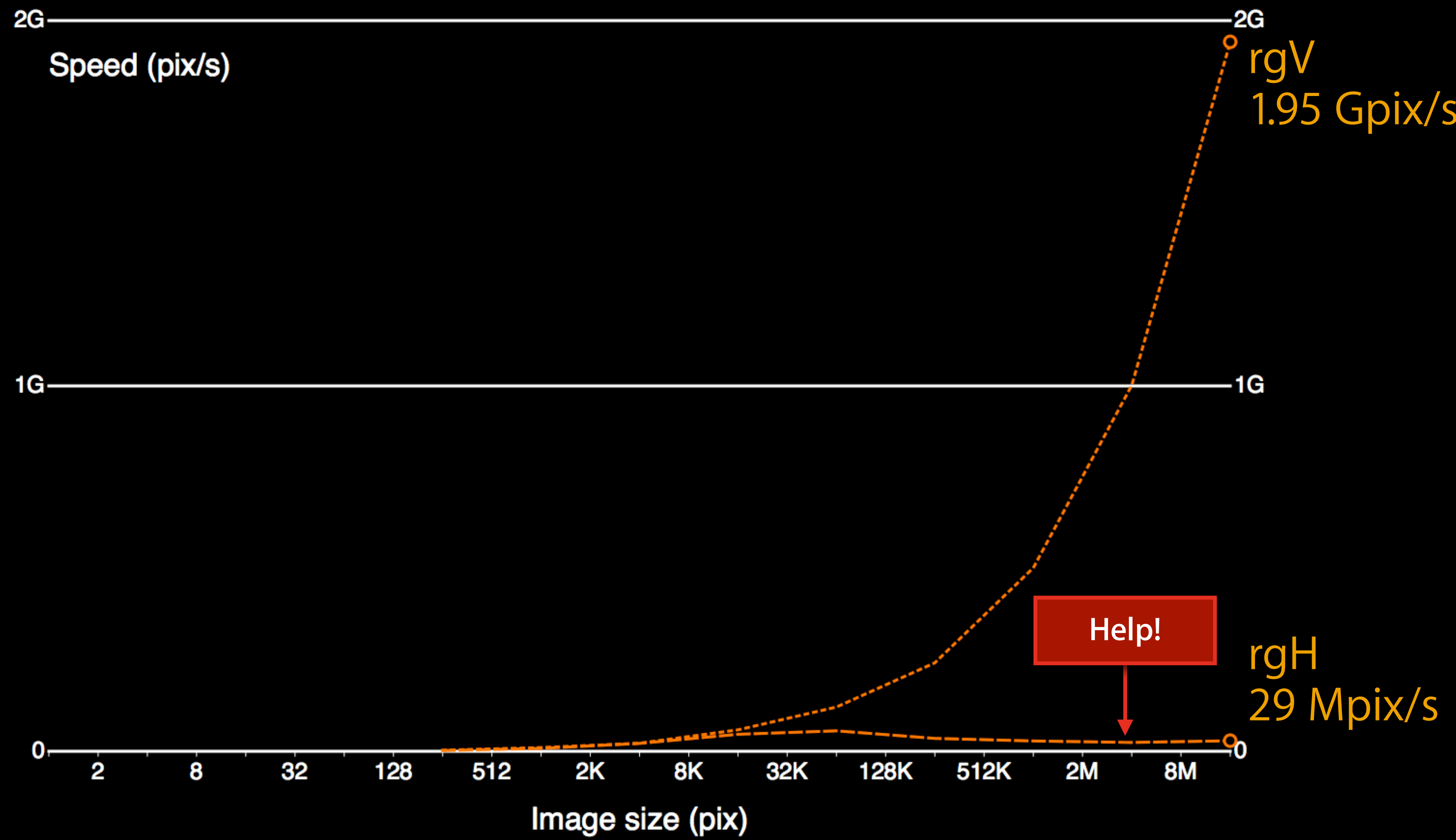
Benchmarks: rgH+rgV



Benchmarks: rgH, rgV



Benchmarks: rgH, rgV



Bottlenecks

30 `Identify bottlenecks`

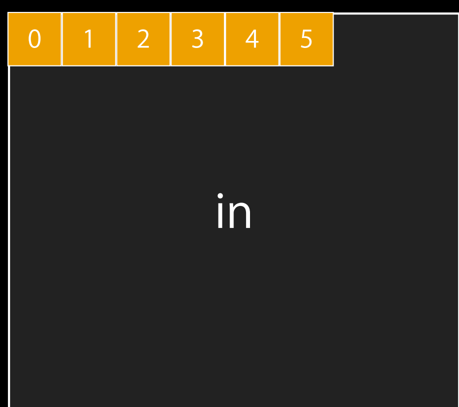
Memory Access Pattern

- Concurrent access pattern ➔ conflicts ➔ serialized = slow
- Simple rules

Memory Access Pattern

- Concurrent access pattern ➔ conflicts ➔ serialized = slow
- Simple rules

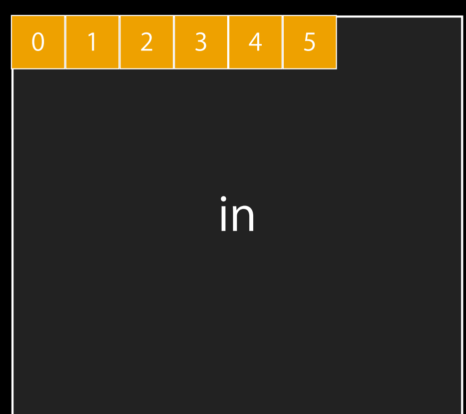
```
global float * in;
int i = get_global_id(0);
float v = in[i];  // FAST
```



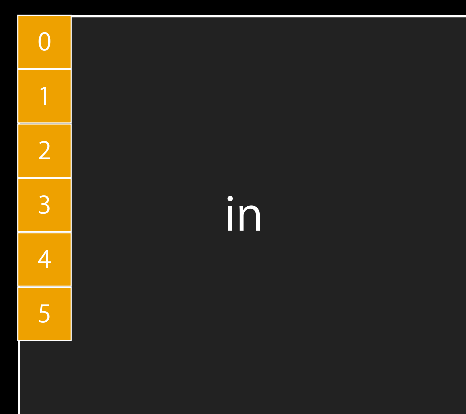
Memory Access Pattern

- Concurrent access pattern ➔ conflicts ➔ serialized = slow
- Simple rules

```
global float * in;
int i = get_global_id(0);
float v = in[i];  // FAST
```




```
global float * in;
int i = get_global_id(0);
float v = in[1024*i];  // SLOW
```

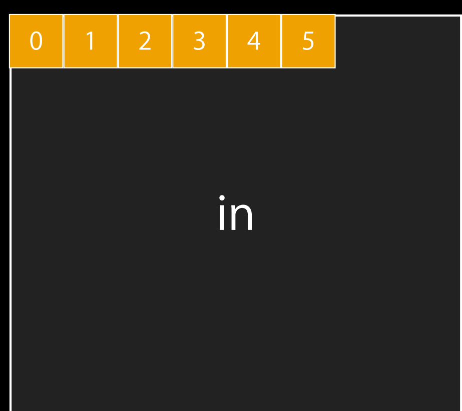


Memory Access Pattern

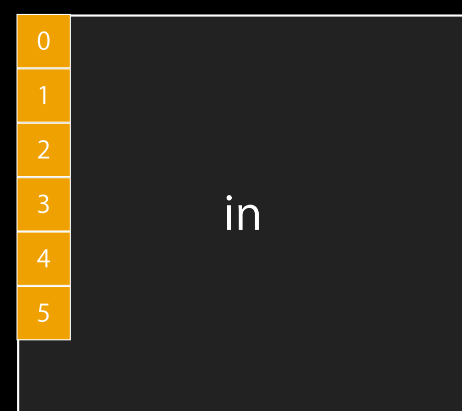
- Concurrent access pattern ➔ conflicts ➔ serialized = slow
- Simple rules

```
global float * in;
int i = get_global_id(0);
float v = in[i];  // FAST
```

```
global float * in;
int i = get_global_id(0);
float v = in[1024*i];  // SLOW
```



BENCHMARK!



copy Memory Access Pattern

```
kernel void copy(global const float * in,
                 global float * out,
                 int w,int h)
{
  int x = get_global_id(0);  // (x,y) = pixel to process
  int y = get_global_id(1);  // in this work item

  out[x+y*w] = in[x+y*w];    // Load + Store
}
```

copy Memory Access Pattern

```
kernel void copy(global const float * in,
                 global float * out,
                 int w,int h)
{
  int x = get_global_id(0);  // (x,y) = pixel to process
  int y = get_global_id(1);  // in this work item

  out[x+y*w] = in[x+y*w];    // Load + Store
}
```

Fast

copy Memory Access Pattern

```
kernel void copy(global const float * in,
                 global float * out,
                 int w,int h)
{
  int x = get_global_id(0);  // (x,y) = pixel to process
  int y = get_global_id(1);  // in this work item

  out[x+y*w] = in[x+y*w];    // Load + Store
}
```

Fast

Fast

rgV Memory Access Pattern

```
// One work item per output column
kernel void rgV(global const float * in,global float * out,int w,int h)
{
  int x = get_global_id(0);      // Column to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2 = o3 = o4 = 0.0f;
  for (int y=0;y<h;y++)
  {
    float i0 = in[x+y*w];        // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;             // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

rgV Memory Access Pattern

```
// One work item per output column
kernel void rgV(global const float * in,global float * out,int w,int h)
{
  int x = get_global_id(0);      // Column to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2         4 = 0.0f;
  for (int y=0;y<h;y++)
  {
    float i0 = in[x+y*w];        // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;             // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

Fast

rgV Memory Access Pattern

```
// One work item per output column
kernel void rgV(global const float * in,global float * out,int w,int h)
{
  int x = get_global_id(0);      // Column to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2   Fast   4 = 0.0f;
  for (int y=0;y<h;y++)
  {
 Fast      i0 = in[x+y*w];          // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;               // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

rgH Memory Access Pattern

```
// One work item per output row
kernel void rgH(global const float * in,global float * out,int w,int h)
{
  int y = get_global_id(0);      // Row to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2 = o3 = o4 = 0.0f;
  for (int x=0;x<w;x++)
  {
    float i0 = in[x+y*w];        // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;             // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

rgH Memory Access Pattern

```
// One work item per output row
kernel void rgH(global const float * in,global float * out,int w,int h)
{
  int y = get_global_id(0);      // Row to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2    Slow    4 = 0.0f;
  for (int x=0;x<w;x++)
  {
    float i0 = in[x+y*w];        // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;             // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

rgH Memory Access Pattern

```
// One work item per output row
kernel void rgH(global const float * in,global float * out,int w,int h)
{
  int y = get_global_id(0);       // Row to process
  // Forward pass
  float i1,i2,i3,o1,o2,o3,o4;
  i1 = i2 = i3 = o1 = o2          4 = 0.0f;
  for (int x=0;x<w;x++)
  {
              i0 = in[x+y*w];         // Load
    float o0 = a0*i0 + a1*i1 + a2*i2 + a3*i3
             - c1*o1 - c2*o2 - c3*o3 - c4*o4;
    out[x+y*w] = o0;                  // Store
    // Rotate values for next pixel
    i3 = i2; i2 = i1; i1 = i0;
    o4 = o3; o3 = o2; o2 = o1; o1 = o0;
  }
  // Backward pass
  ...
}
```

Slow

Slow

Solution

40 Find solution/workaround

Drop the rgH Kernel

Use rgV twice instead

- rgV + Transpose + rgV + Transpose = rgH + rgV

Drop the rgH Kernel

Use rgV twice instead

- rgV + Transpose + rgV + Transpose = rgH + rgV

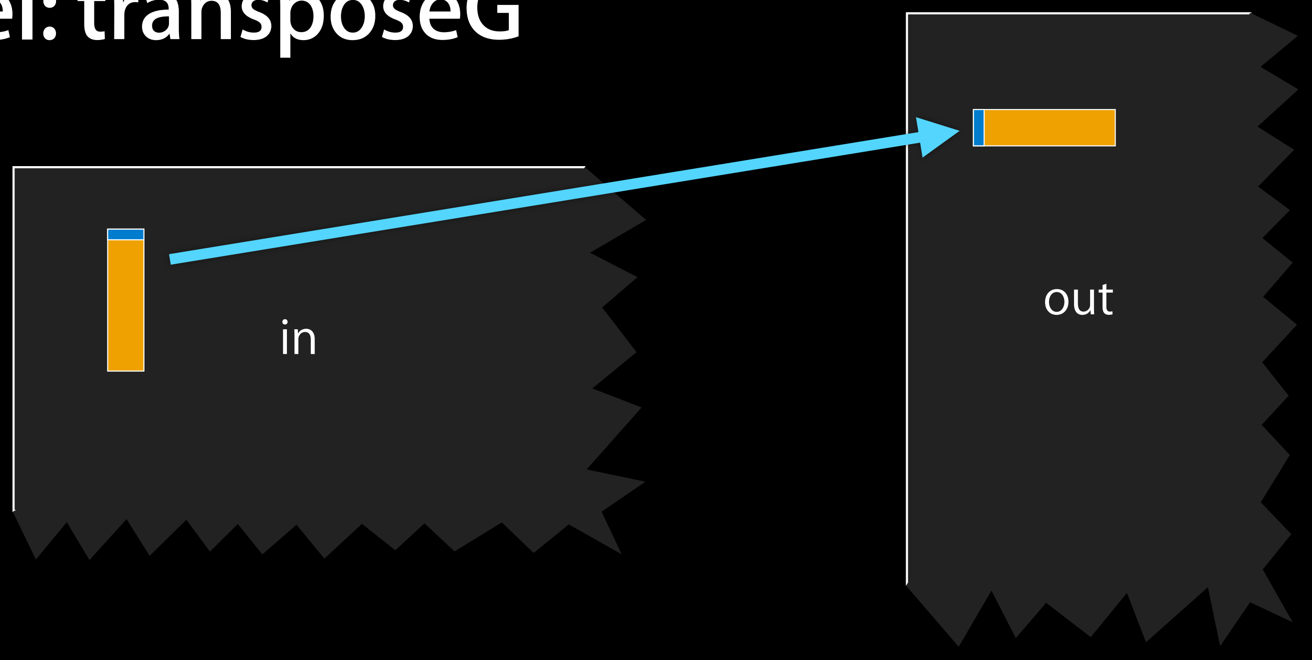
Drop the rgH Kernel

Use rgV twice instead

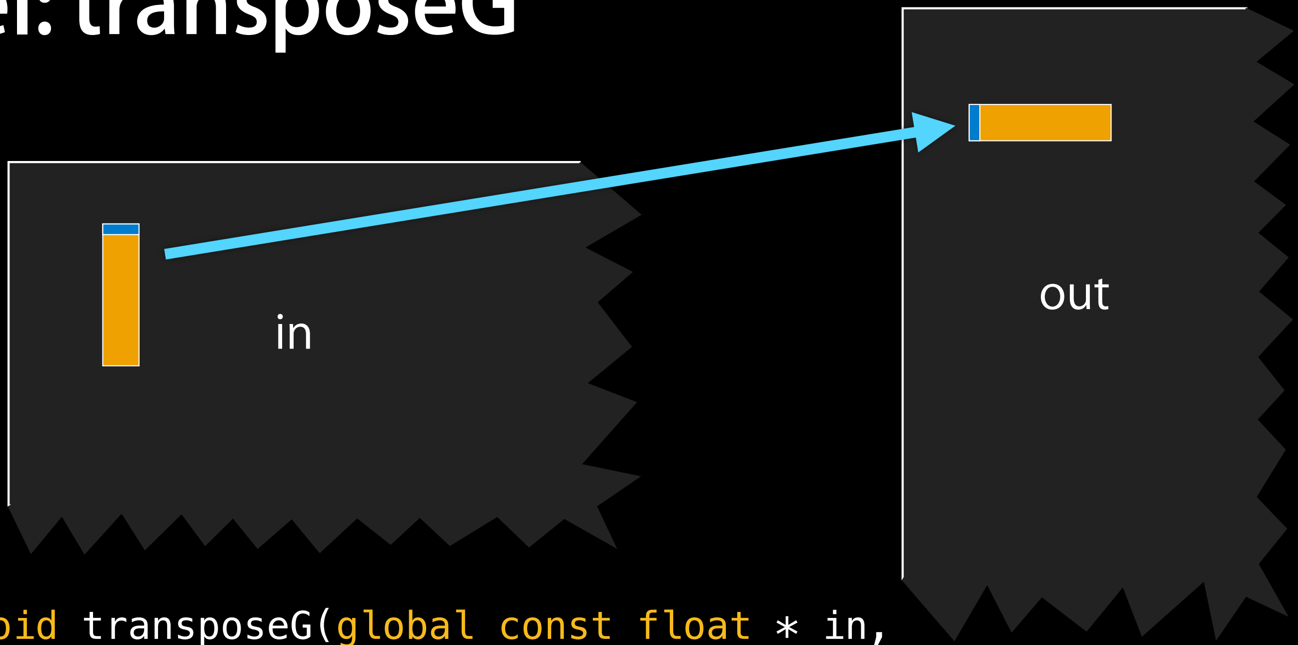
- rgV + Transpose + rgV + Transpose = rgH + rgV

Algorithm	Memory (float R+W)	Compute (flops)	C / M Ratio	Estimate (Mpix/s)
V+T+V+T	14	64	4.6	2,030

Kernel: transposeG

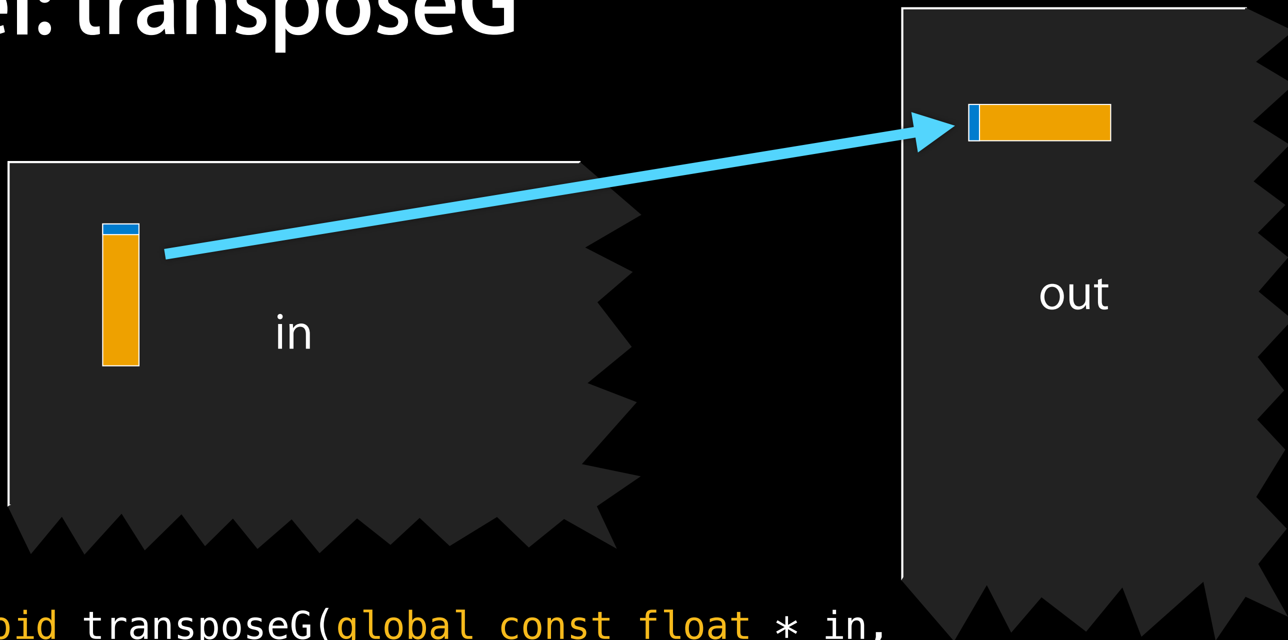


Kernel: transposeG




```
kernel void transposeG(global const float * in,
                       global float * out,
                       int w,int h)
{
  int x = get_global_id(0);
  int y = get_global_id(1);

  out[y+x*h] = in[x+y*w];  // (x,y) in input = (y,x) in output
}
```

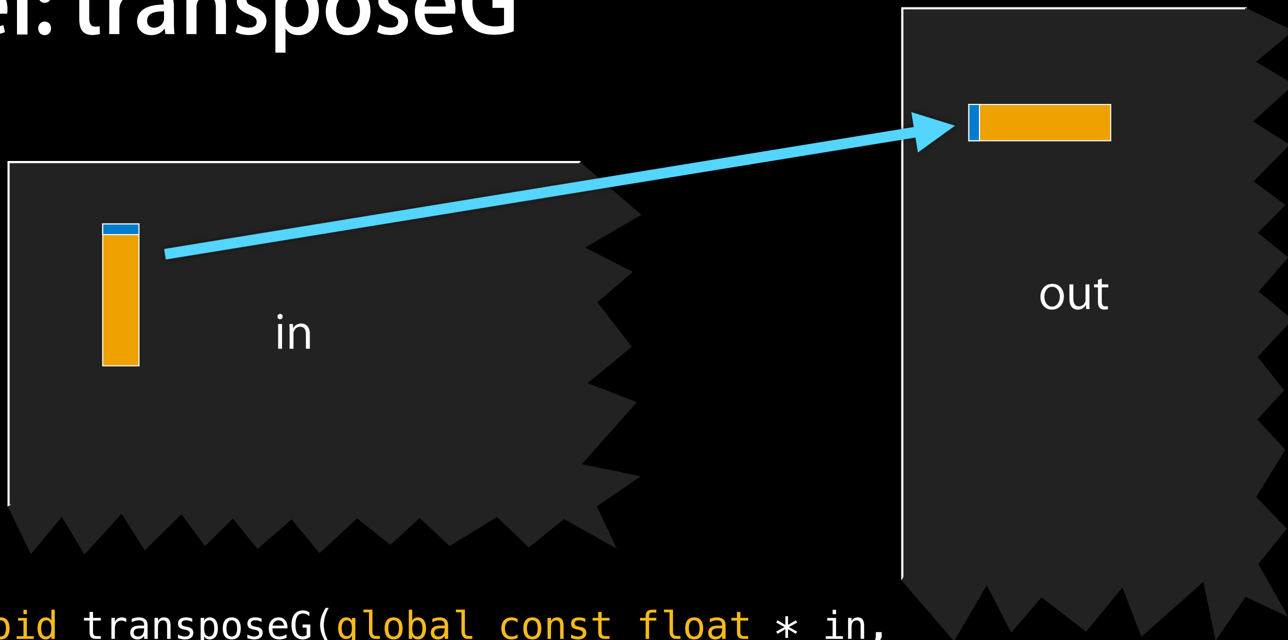
Kernel: transposeG




```
kernel void transposeG(global const float * in,
                       global float * out,
                       int w,int h)
{
  int x = get_global_id(0);
  int y = get_global_id(1);

  out[y+x*h] = in[x+y*w];  // (x,y) in input = (y,x) in output
}
```

Fast

Kernel: transposeG

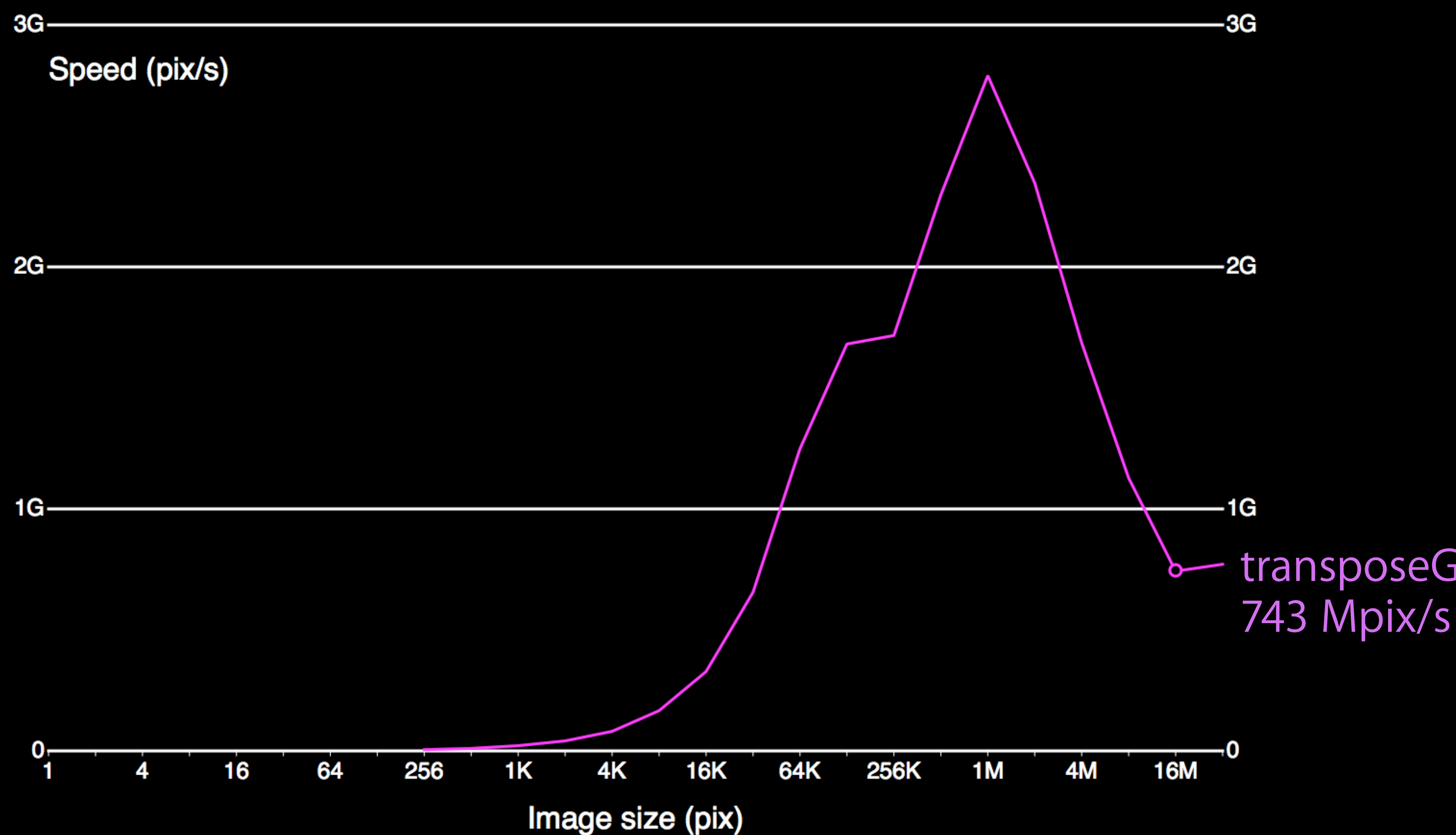


Slow

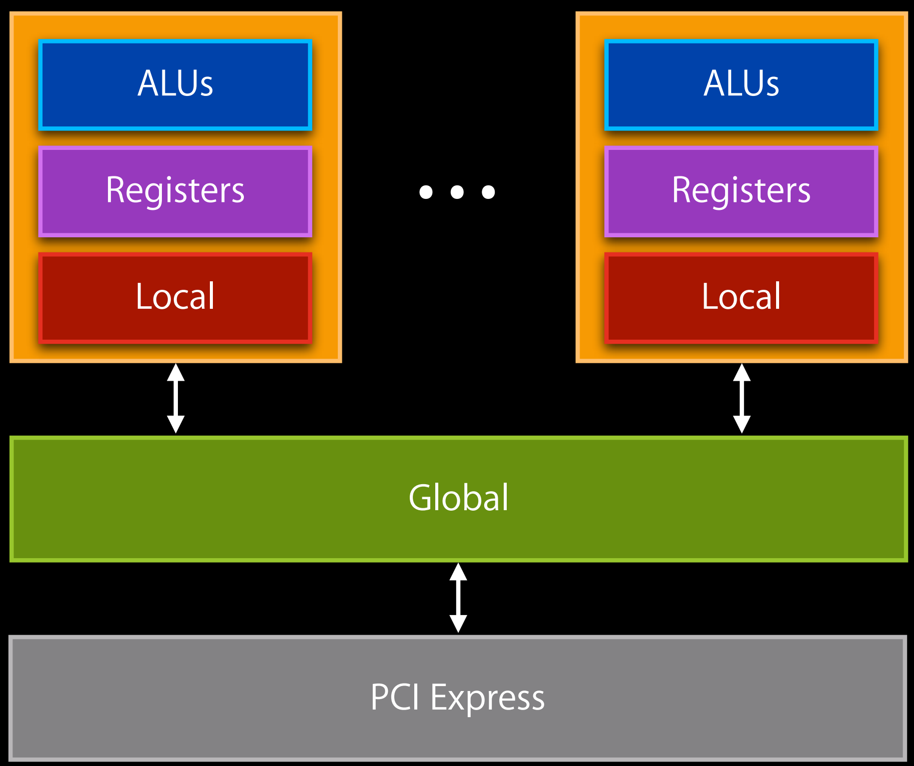
Fast

```
kernel void transposeG(global const float * in,
                       global float * out,
                       int w,int h)
{
  int x = get_global_id(0);
  int y = get_global_id(1);

  out[y+x*h] = in[x+y*w];  // (x,y) in input = (y,x) in output
}
```

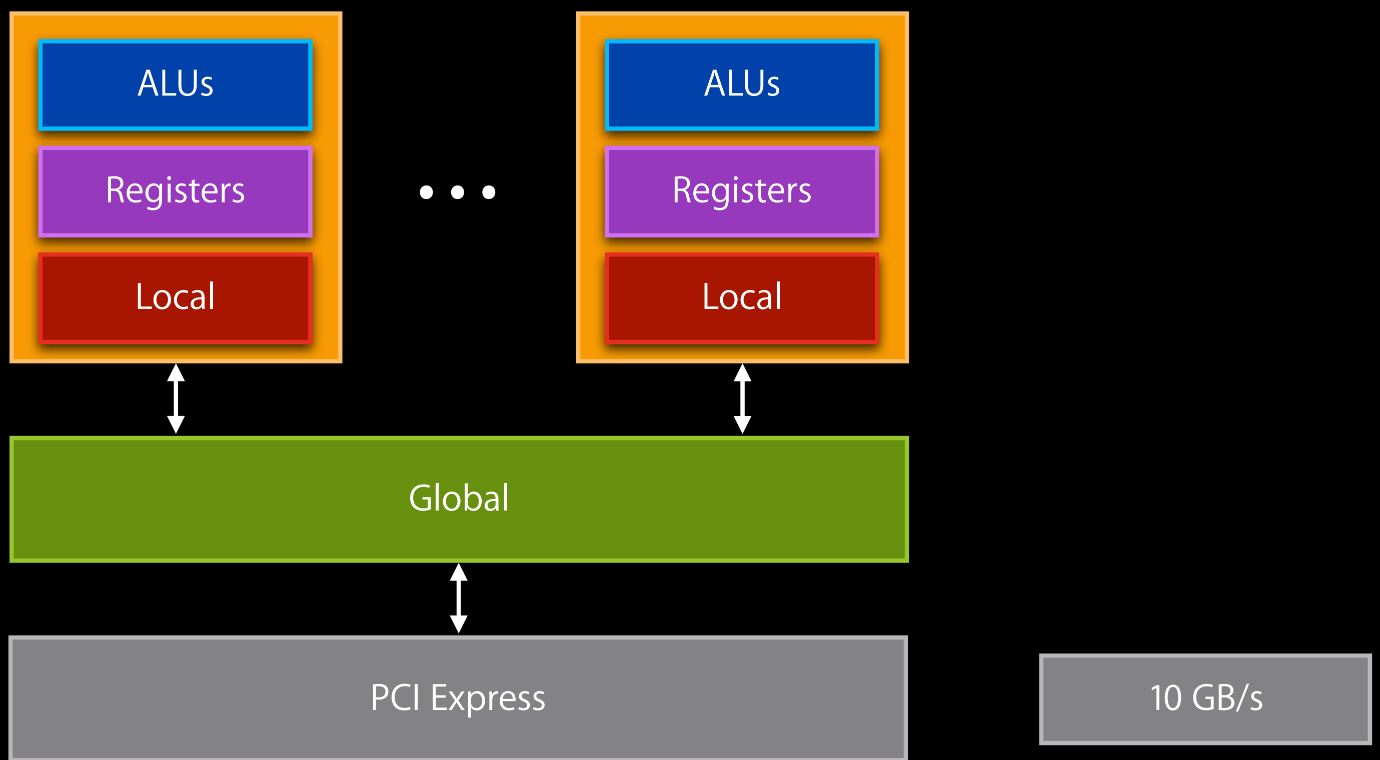
Benchmarks: transposeG



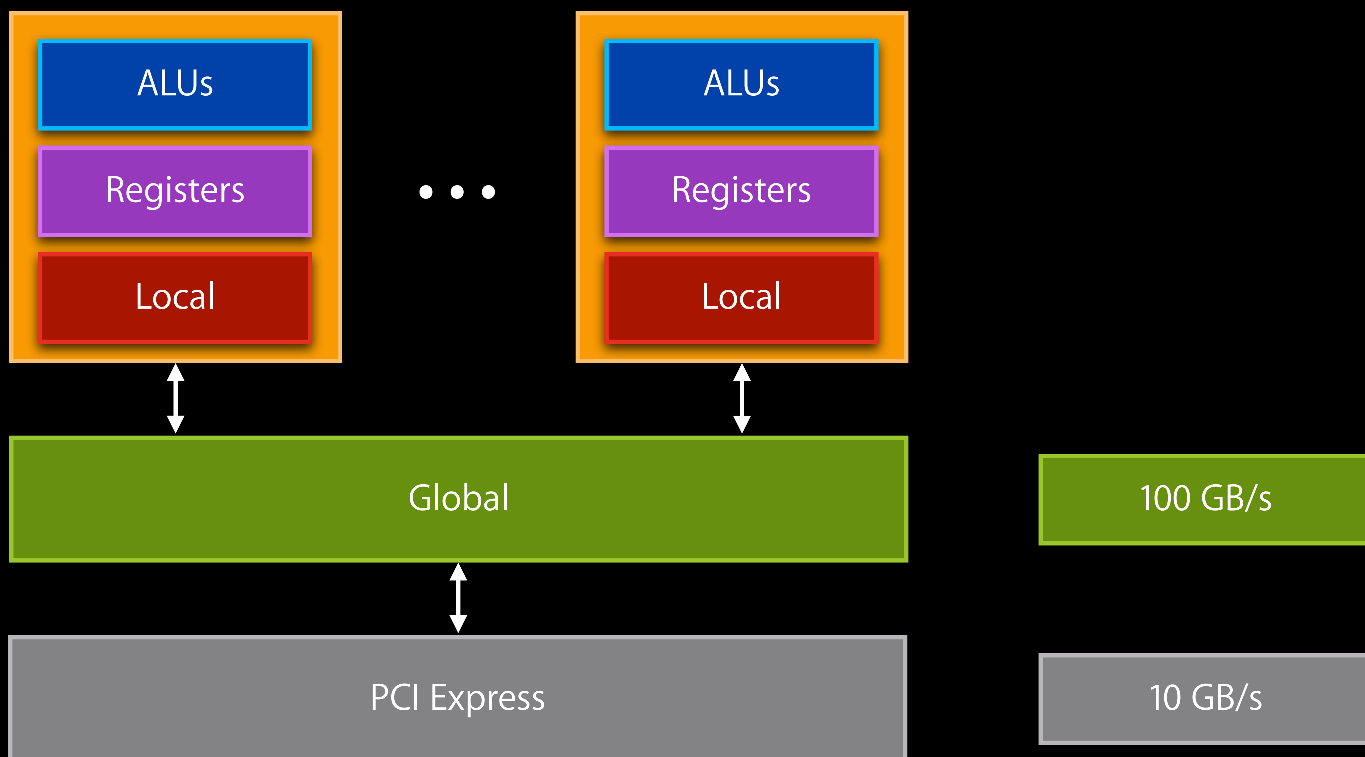
GPU Memory Hierarchy



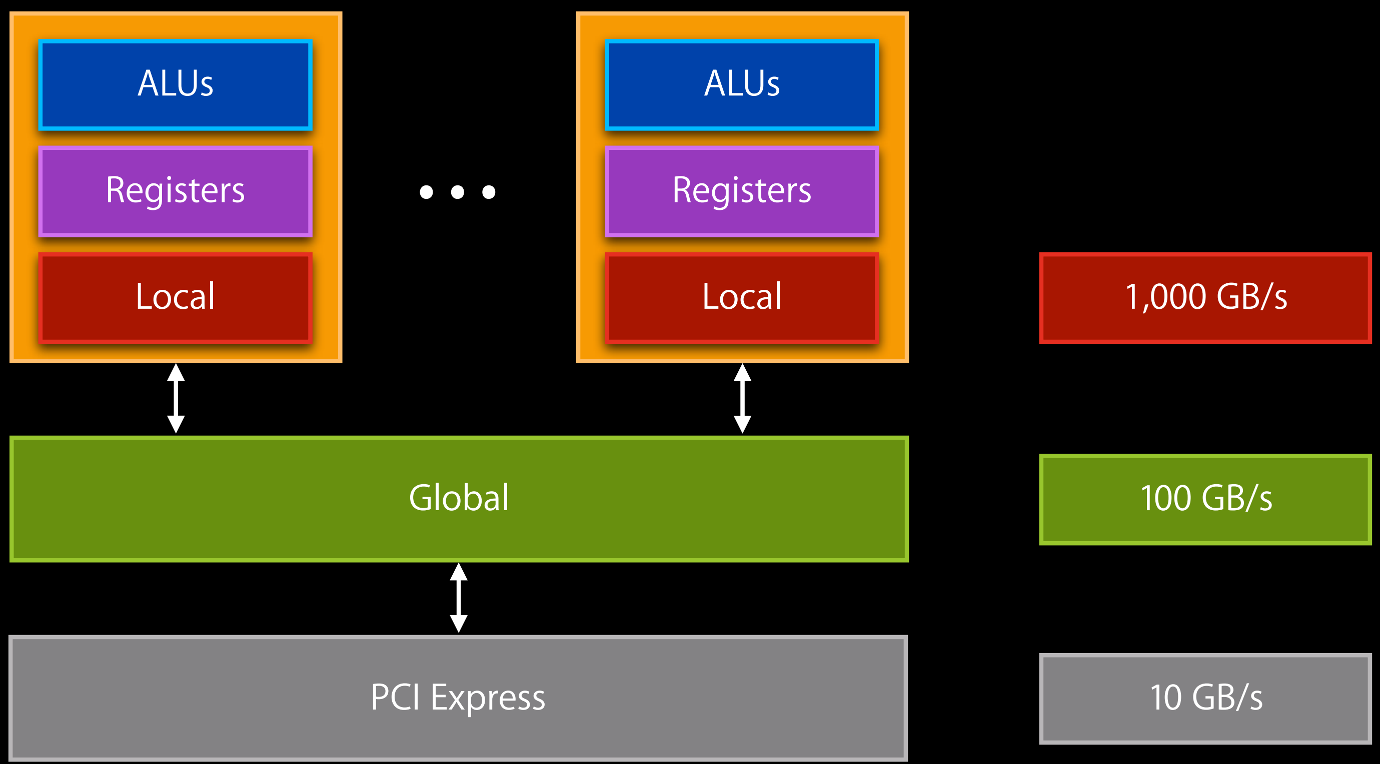
GPU Memory Hierarchy



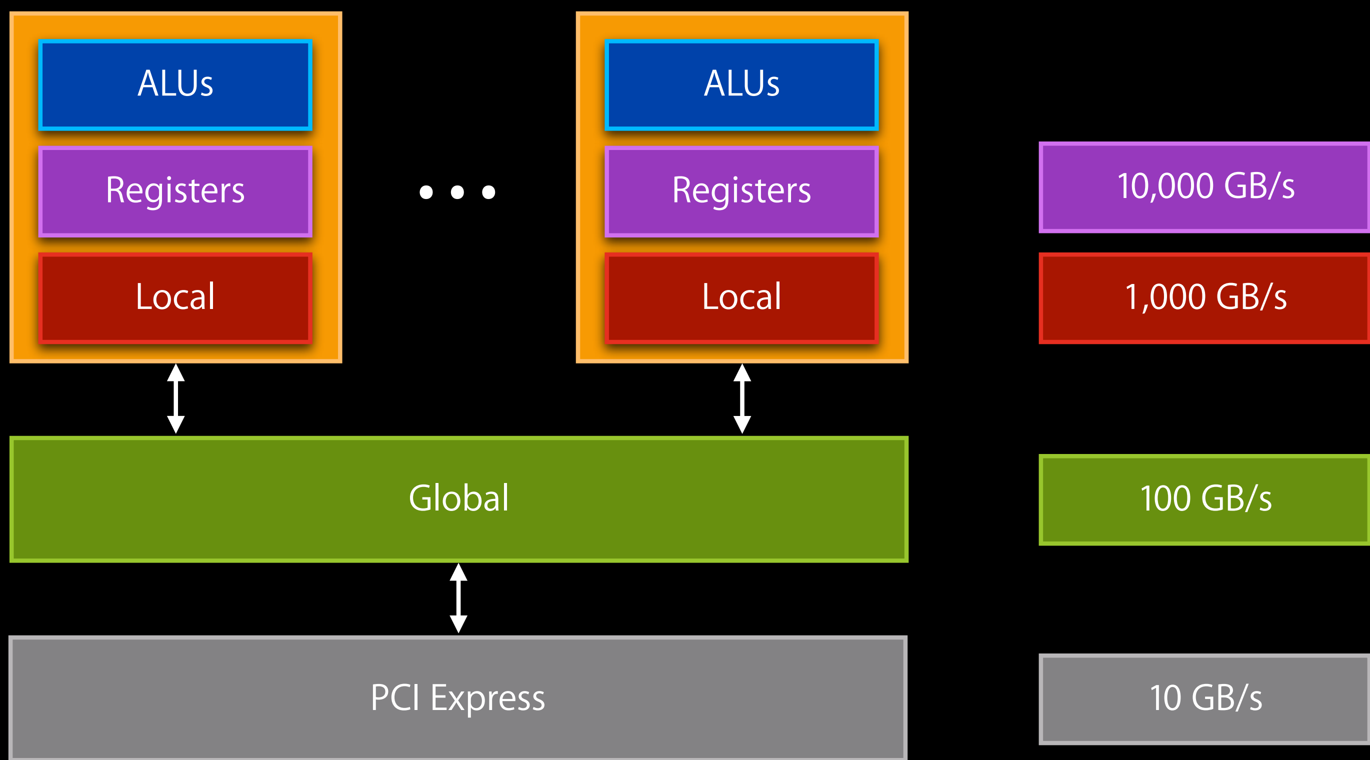
GPU Memory Hierarchy



GPU Memory Hierarchy

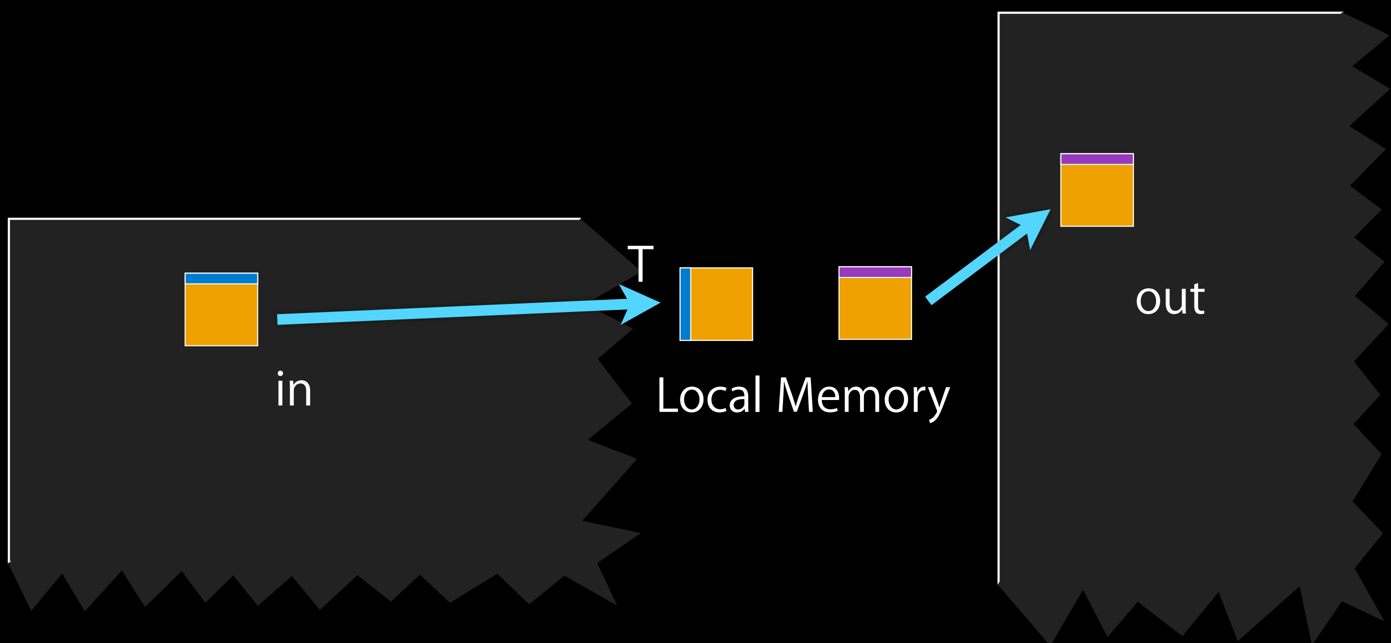


GPU Memory Hierarchy



Improving the Transpose Kernel

Using local memory



Kernel: transposeL

```
kernel void transposeL(global const float * in,
                       global float * out,
                       int w,int h)
{
  local float aux[256];        // Block size is 16x16
  int bx = get_group_id(0),    // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),    // (ix,iy) = pixel in block
      iy = get_local_id(1);

  in  += (bx*16)+(by*16)*w;    // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];     // Read block
  barrier(CLK_LOCAL_MEM_FENCE);    // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];    // Write block
}
```

Kernel: transposeL

```
kernel void transposeL(global const float * in,
                       global float * out,
                       int w,int h)
{
  local float aux[256];        // Block size is 16x16
  int bx = get_group_id(0),    // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),    // (ix,iy) = pixel in block
      iy = get_local_id(1);

  in  += (bx*16)+(by*16)*w;    // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];     // Read block
  barrier(CLK_LOCAL_MEM_FENCE);    // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];    // Write block
}
```

Kernel: transposeL

```
kernel void transposeL(global const float * in,
                       global float * out,
                       int w,int h)
{
  local float aux[256];         // Block size is 16x16
  int bx = get_group_id(0),     // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),     // (ix,iy) = pixel in block
      iy = get_local_id(1);

  in  += (bx*16)+(by*16)*w;     // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];      // Read block
  barrier(CLK_LOCAL_MEM_FENCE);     // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];     // Write block
}
```

Kernel: transposeL

```
kernel void transposeL(global const float * in,
                       global float * out,
                       int w,int h)
{
  local float aux[256];         // Block size is 16x16
  int bx = get_group_id(0),     // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),     // (ix,iy) = pixel in block
      iy = get_local_id(1);

  in  += (bx*16)+(by*16)*w;     // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];      // Read block
  barrier(CLK_LOCAL_MEM_FENCE);     // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];     // Write block
}
```

Kernel: transposeL

```
kernel void transposeL(global const float * in,
                       global float * out,
                       int w,int h)
{
  local float aux[256];          // Block size is 16x16
  int bx = get_group_id(0),      // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),      // (ix,iy) = pixel in block
      iy = get_local_id(1);

  in  += (bx*16)+(by*16)*w;       ve to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];       // Read block
  barrier(CLK_LOCAL_MEM_FENCE);      // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];      // Write block
}
```

Fast

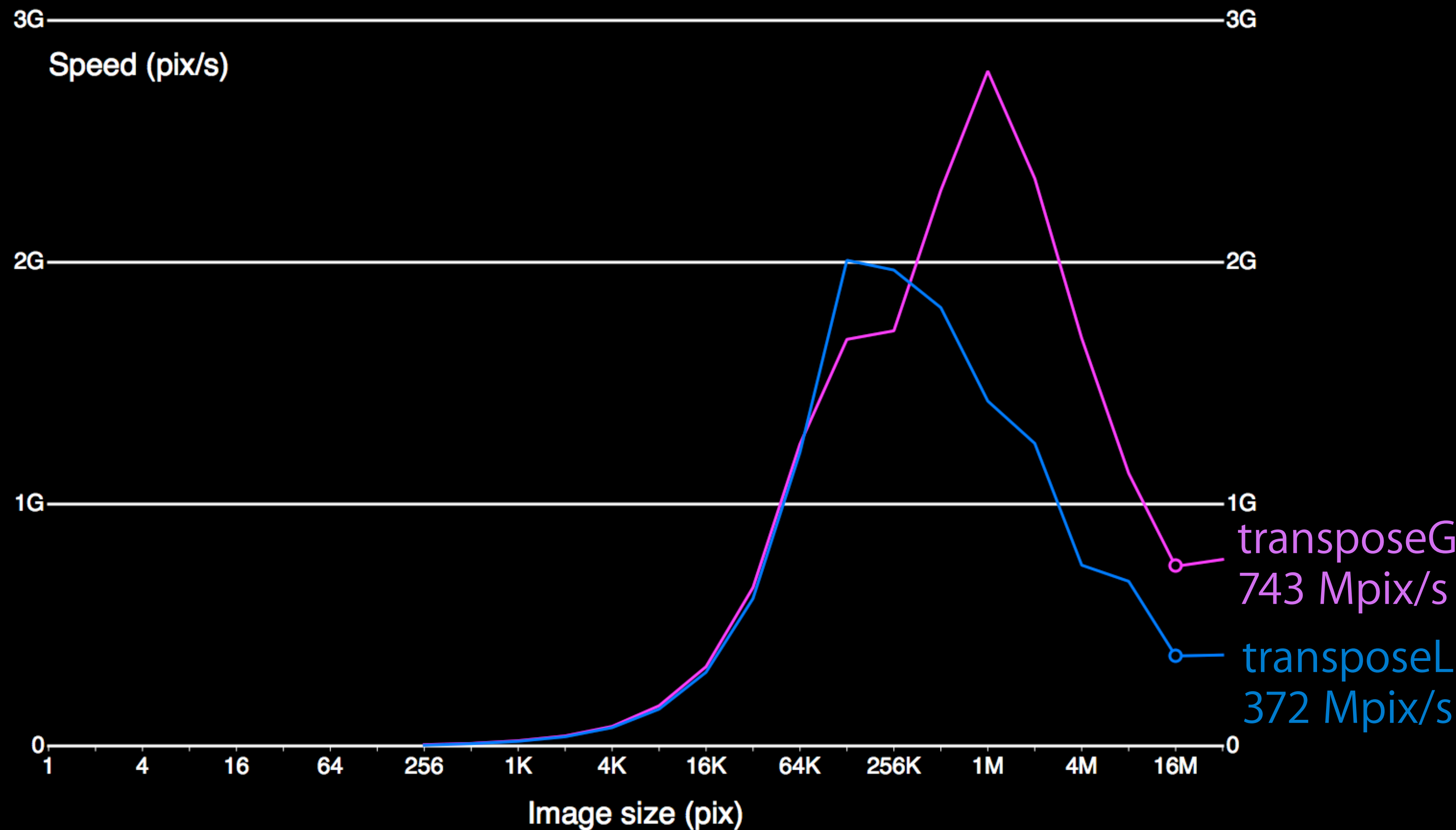
Kernel: transposeL

```
kernel void transposeL(global const float * in,
                       global float * out,
                       int w,int h)
{
  local float aux[256];        // Block size is 16x16
  int bx = get_group_id(0),    // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),    // (ix,iy) = pixel in block
      iy = get_local_id(1);

  in  += (bx*16)+(by*16)*w;    ve to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];      // Read block
  barrier(CLK_LOCAL_MEM_FENCE);     // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];     // Write block
}
```

Fast

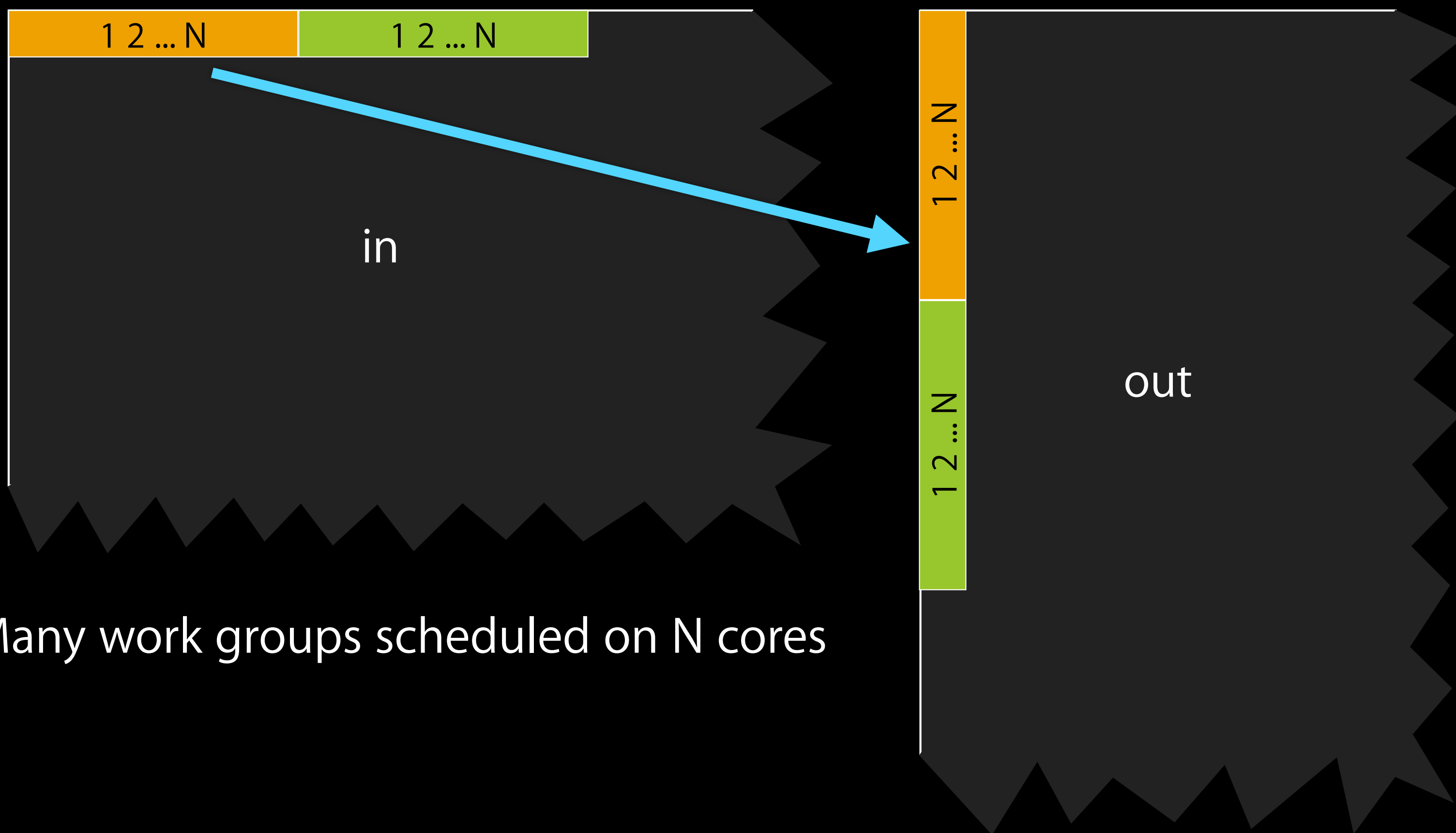
Fast

Benchmarks: transposeL



transposeL Memory Access Pattern

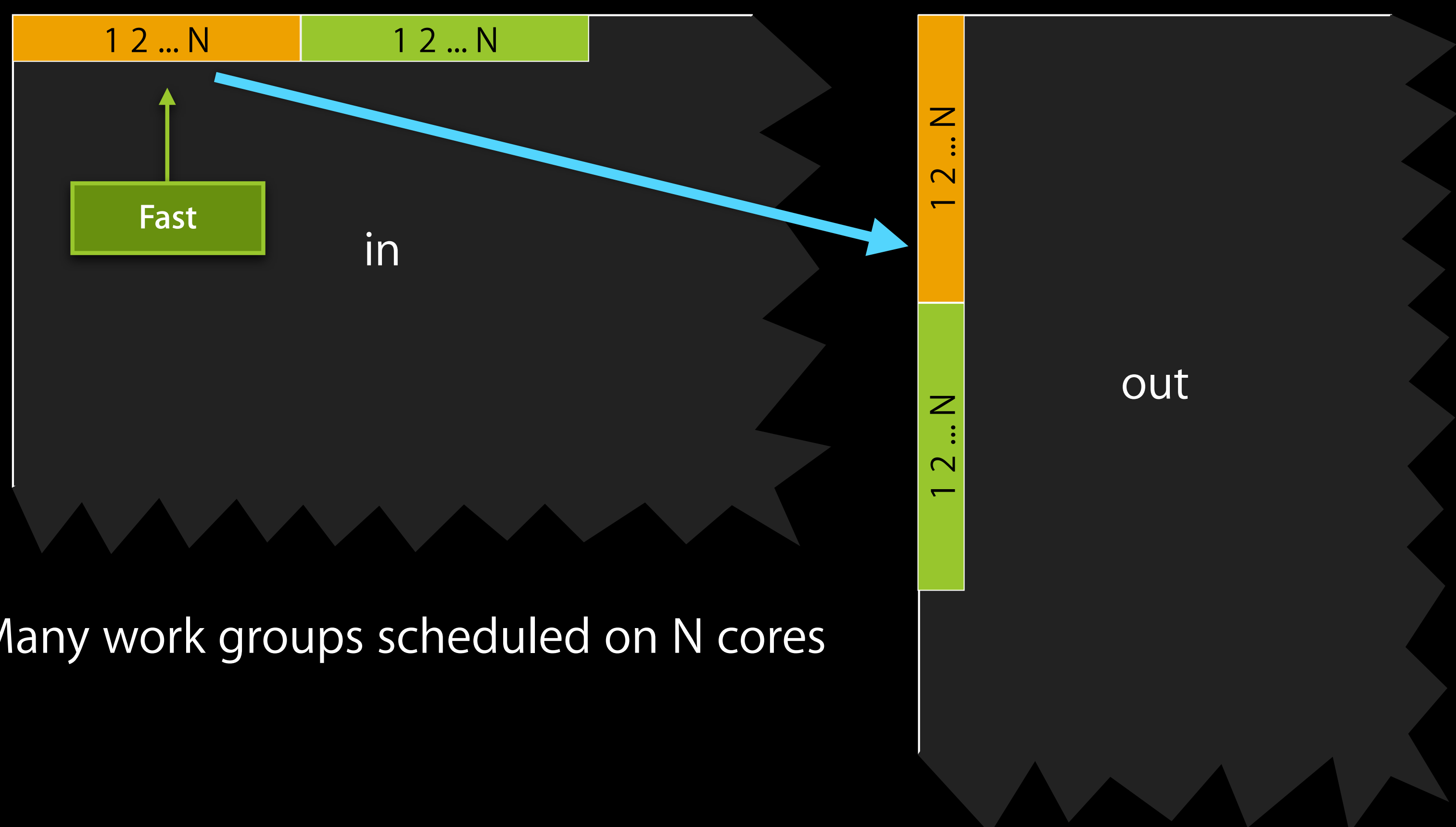
Partition camping



Many work groups scheduled on N cores

transposeL Memory Access Pattern

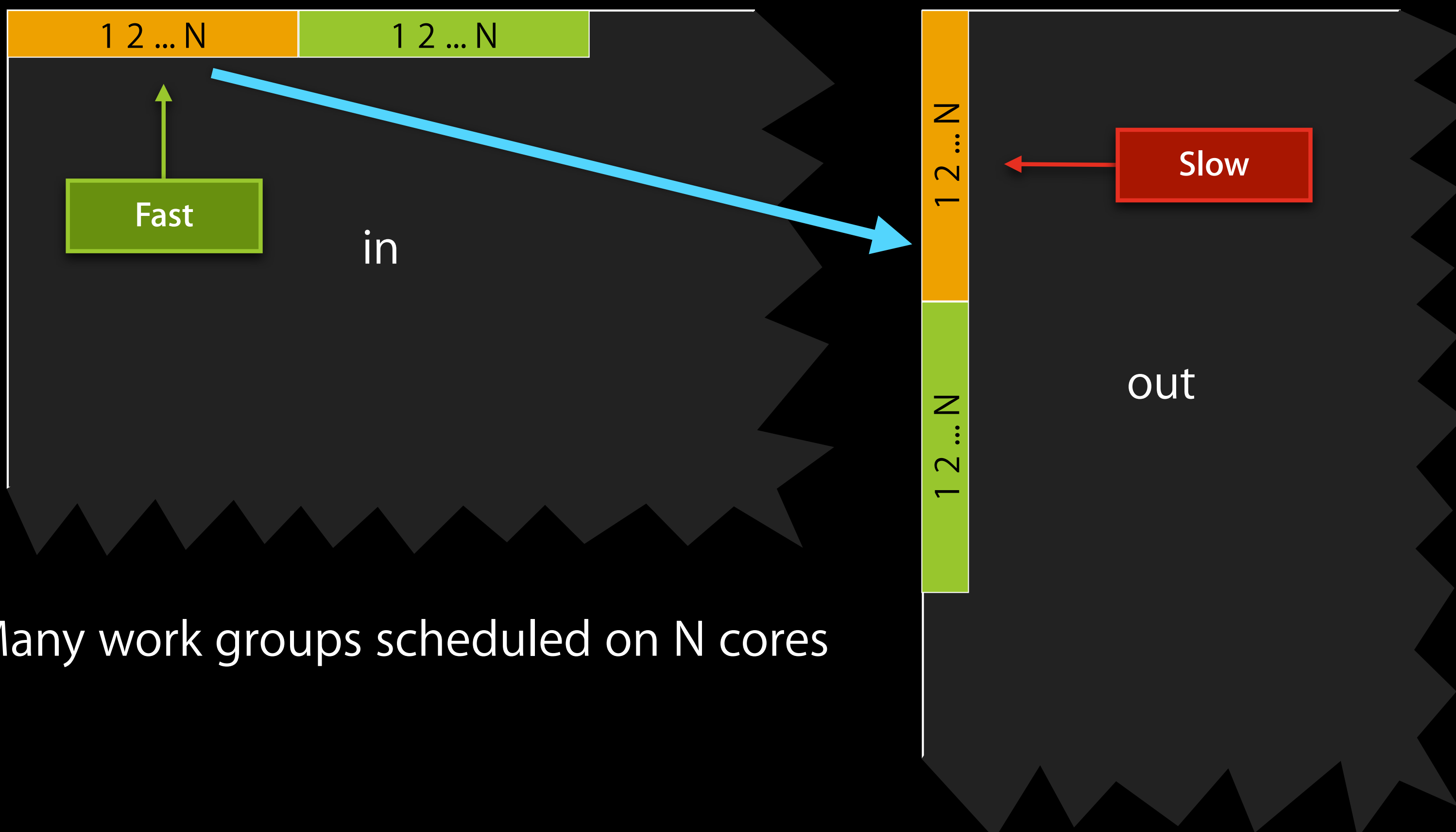
Partition camping



Many work groups scheduled on N cores

transposeL Memory Access Pattern

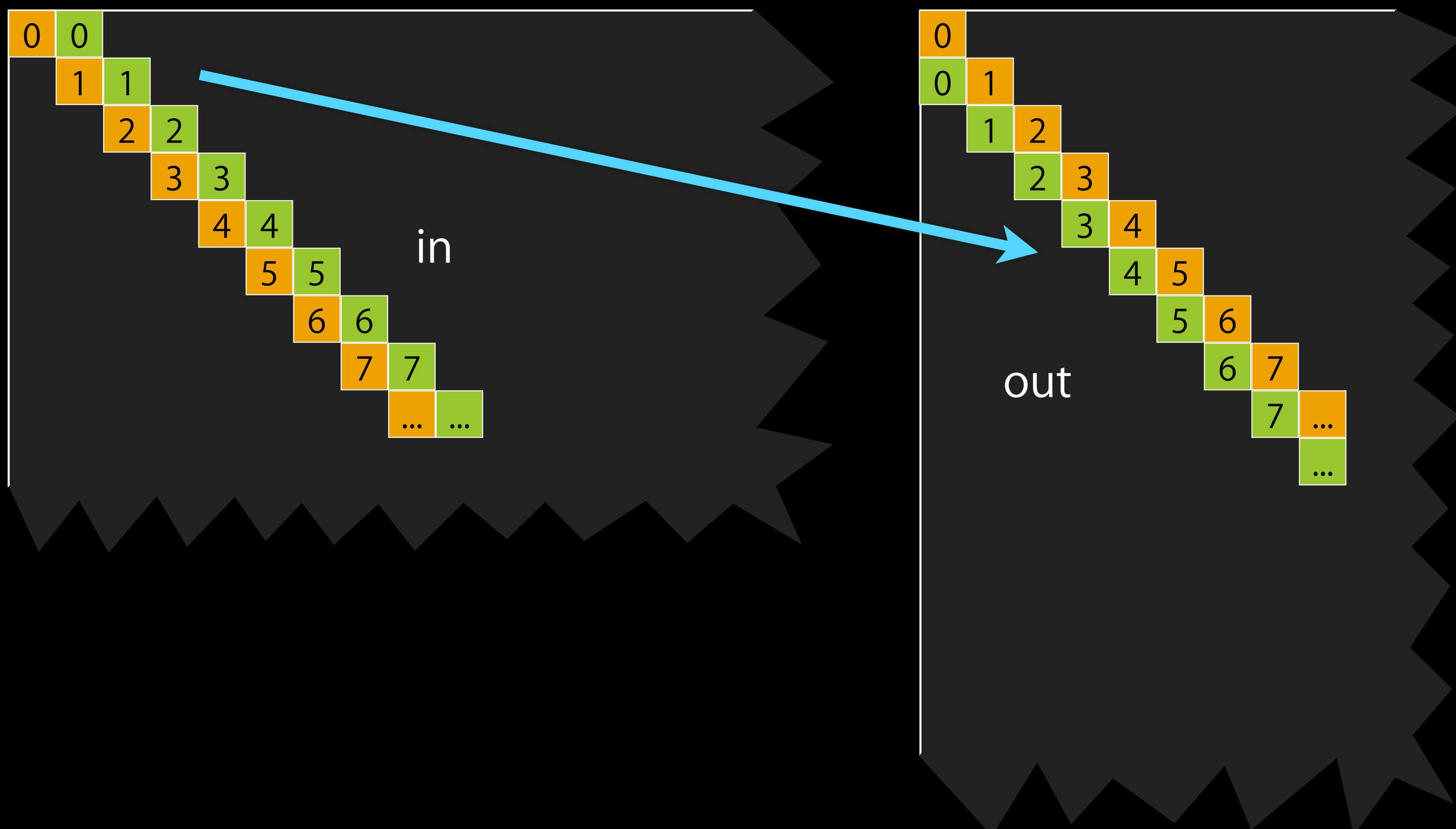
Partition camping



Many work groups scheduled on N cores

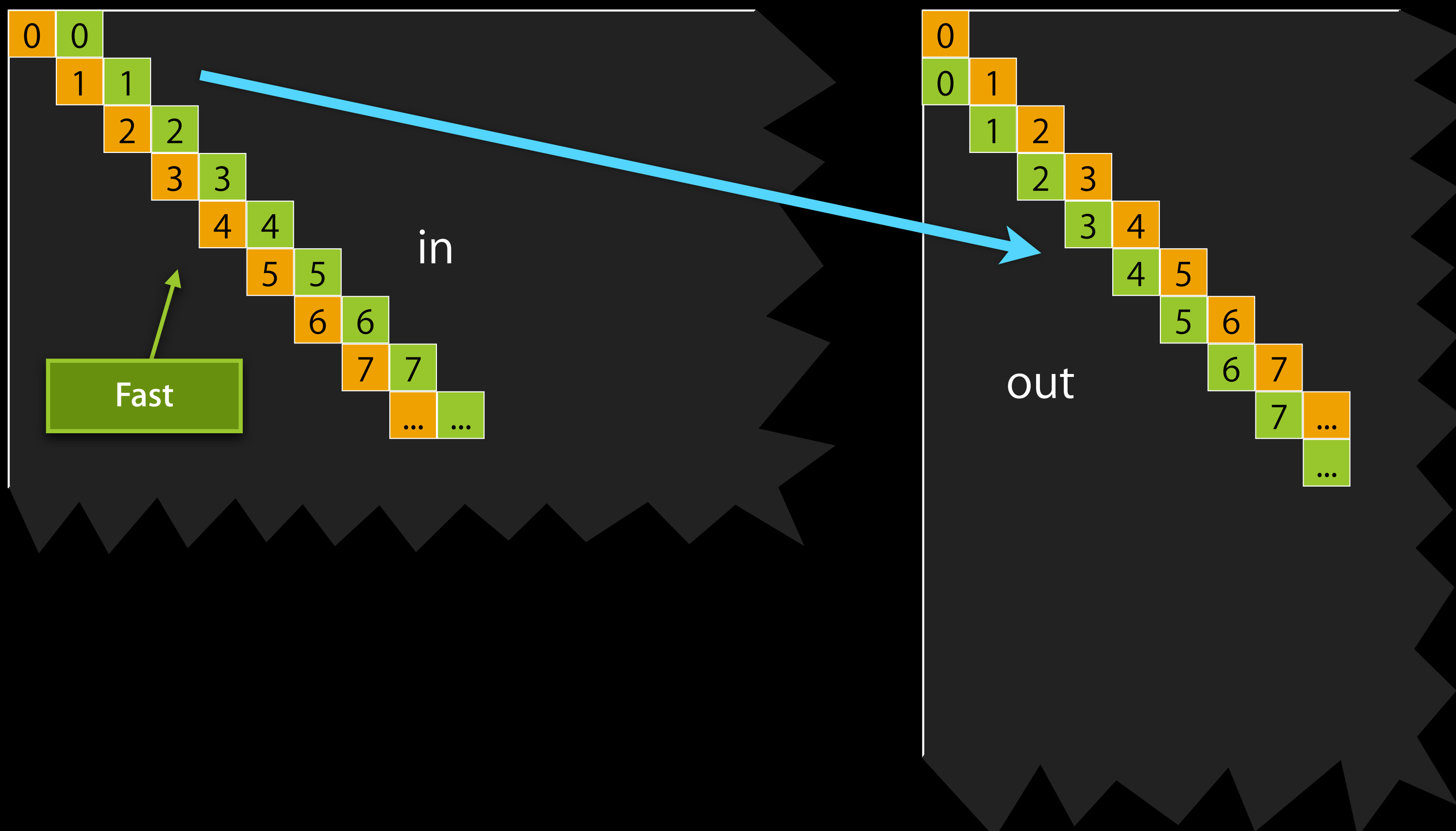
Solution

Skew block mapping



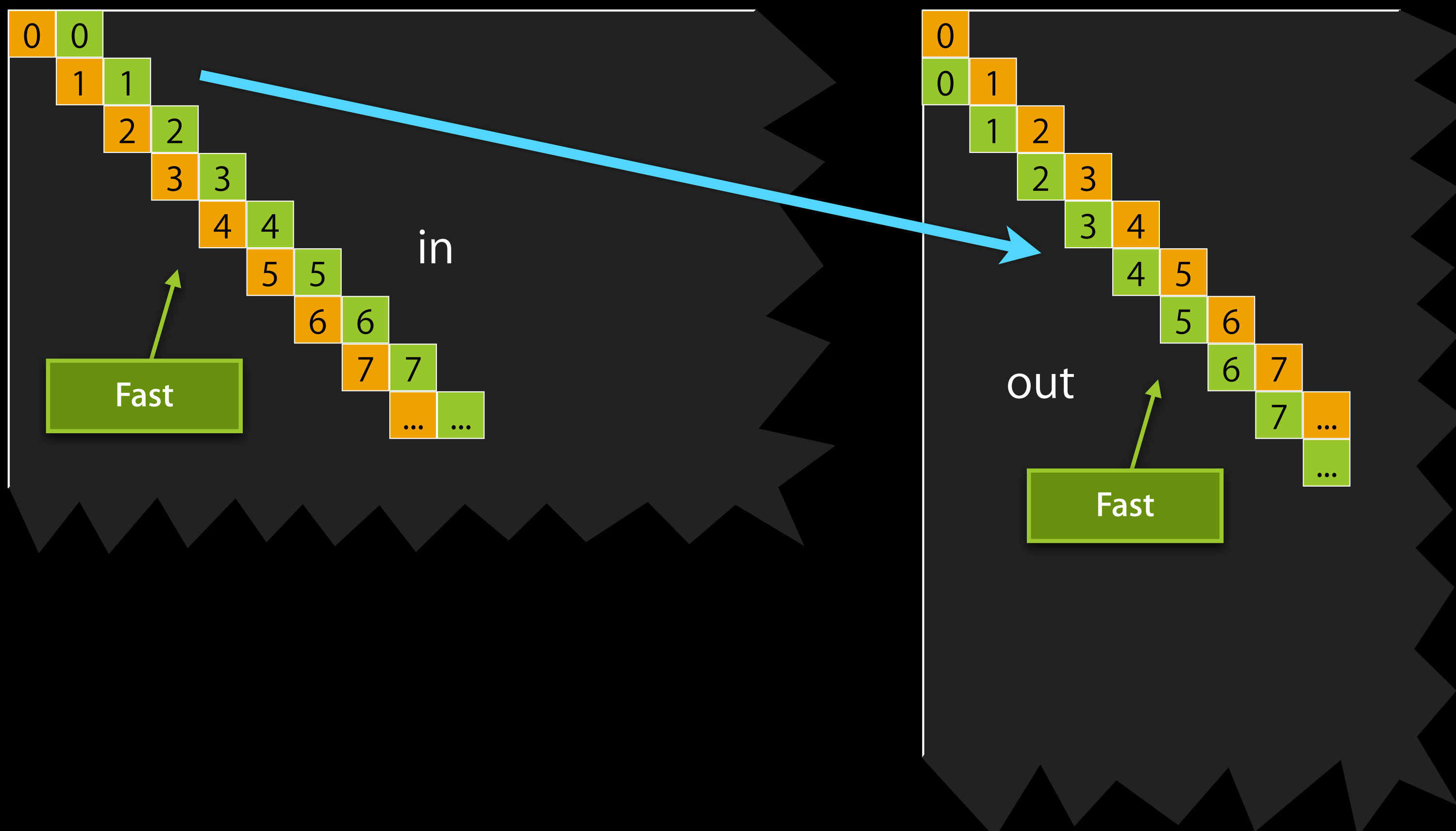
Solution

Skew block mapping



Solution

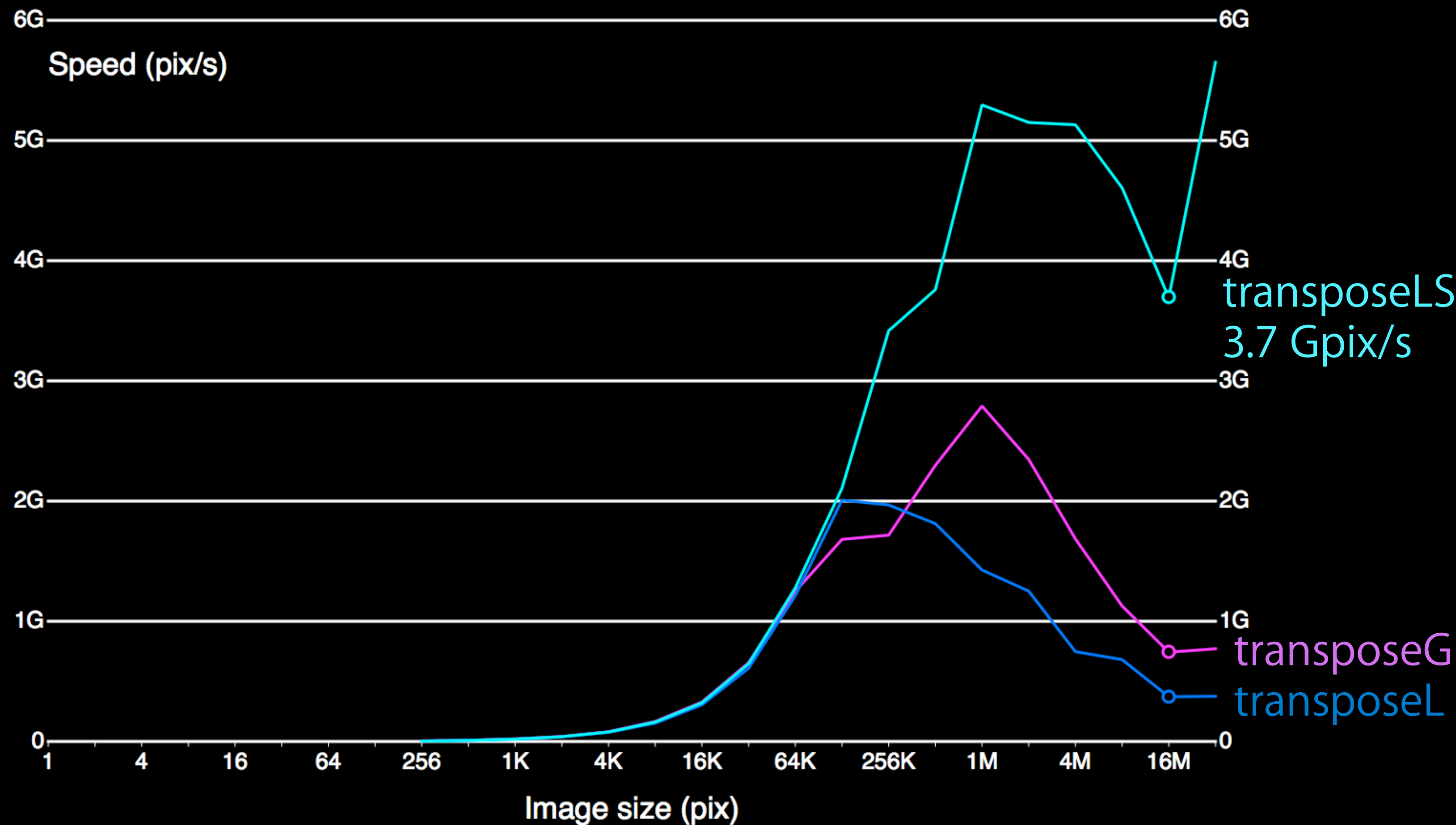
Skew block mapping



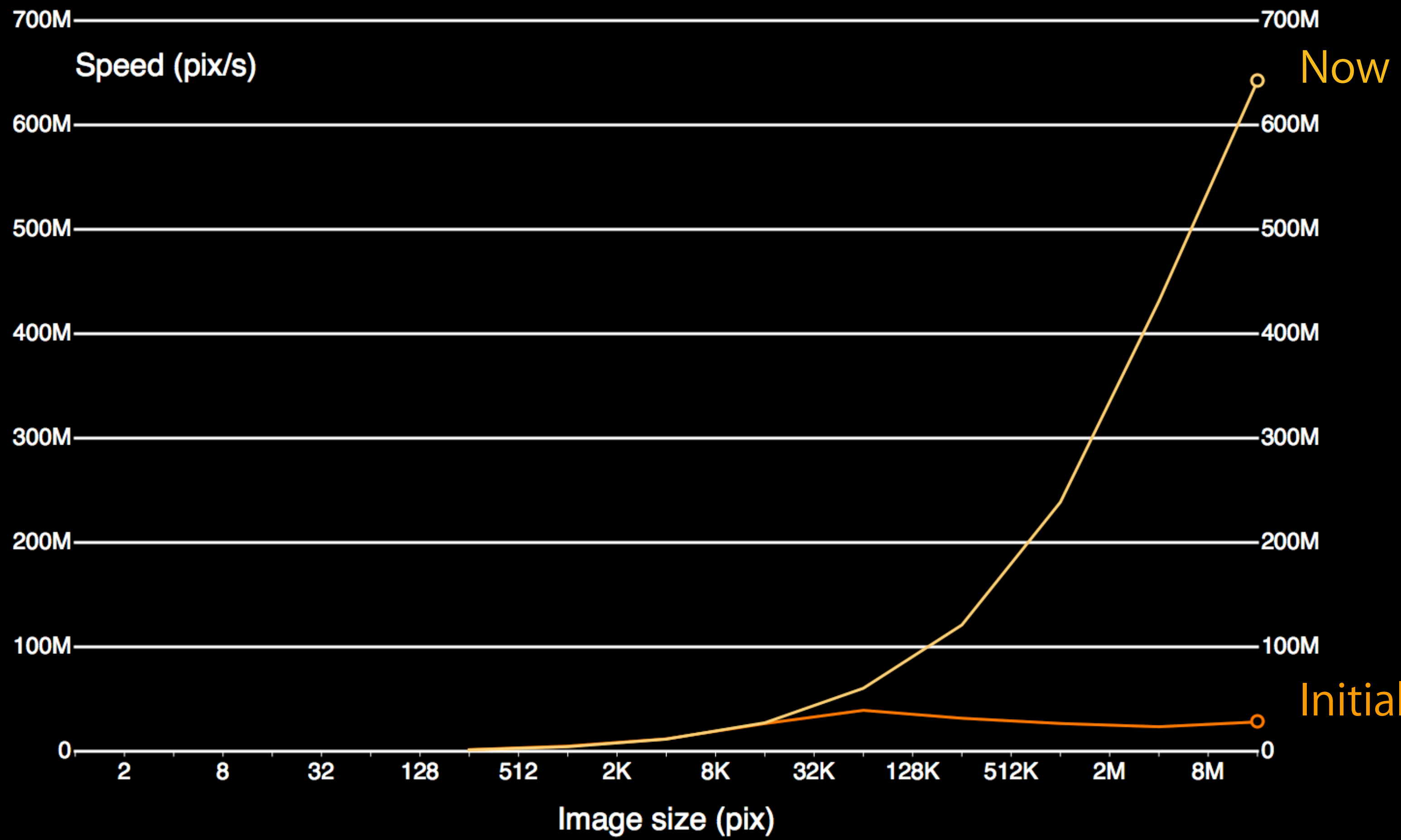
Kernel: transposeLS

```
kernel void transposeLS(global const float * in,
                        global float * out,
                        int w,int h)
{
  local float aux[256];        // Block size is 16x16
  int bx = get_group_id(0),    // (bx,by) = input block
      by = get_group_id(1);
  int ix = get_local_id(0),    // (ix,iy) = pixel in block
      iy = get_local_id(1);

  by = (by+bx)%get_num_groups(1);  // Skew mapping

  in  += (bx*16)+(by*16)*w;    // Move to origin of in,out blocks
  out += (by*16)+(bx*16)*h;

  aux[iy+ix*16] = in[ix+w*iy];     // Read block
  barrier(CLK_LOCAL_MEM_FENCE);    // Synchronize
  out[ix+h*iy] = aux[ix+iy*16];    // Write block
}
```

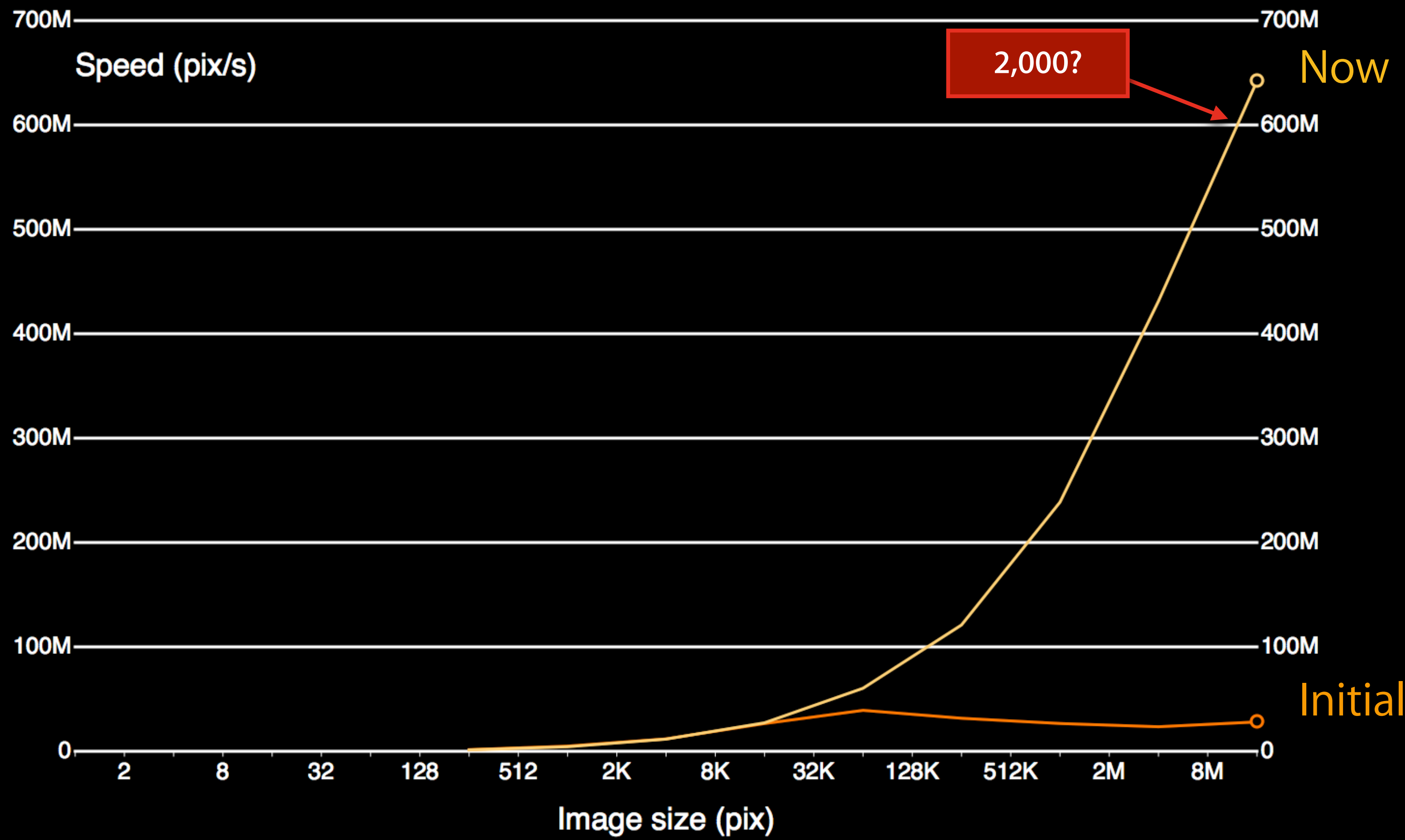
Benchmarks: transposeLS



Benchmarks: Putting All Together



Benchmarks: Putting All Together



Summary

Kernel Tuning BASIC(s)

```
 1 Choose the right algorithm
10 Write the code
20 Benchmark
21 if “fast enough” goto DONE
30 Identify bottlenecks
40 Find solution/workaround
50 goto 10
```

Photoshop OpenCL Notes

Russell Williams
Photoshop Architect
Adobe Systems Inc.

Demo

Blur Gallery

What Was That?

What Was That?

- OpenCL kernel simulating lens optics

What Was That?

- OpenCL kernel simulating lens optics
- Broken into 2K x 2K blocks for GPU

Why OpenCL

Why OpenCL

- Only cross-platform GPGPU solution

Why OpenCL

- Only cross-platform GPGPU solution
- Advantages over OpenGL
  - Learning curve; data formats; debugging

Why OpenCL

- Only cross-platform GPGPU solution
- Advantages over OpenGL
  - Learning curve; data formats; debugging
- Increasing maturity and ubiquity

GPU Challenges

GPU Challenges

- Need good candidate algorithms
  - (bandwidth || compute) && embarrassingly parallel

GPU Challenges

- Need good candidate algorithms
  - (bandwidth || compute) && embarrassingly parallel
- Need debugged C algorithm first

GPU Challenges

- Need good candidate algorithms
    - (bandwidth || compute) && embarrassingly parallel
- Need debugged C algorithm first
- GPU win over CPU variable and unpredictable

GPU Challenges

- Need good candidate algorithms
  - (bandwidth || compute) && embarrassingly parallel
- Need debugged C algorithm first
- GPU win over CPU variable and unpredictable
- Resource limits

GPU Challenges

- Need good candidate algorithms
  - (bandwidth || compute) && embarrassingly parallel
- Need debugged C algorithm first
- GPU win over CPU variable and unpredictable
- Resource limits
- Platform variation

Would We Do It Again: Yes!

Would We Do It Again: Yes!

- More OpenCL in future versions

Would We Do It Again: Yes!

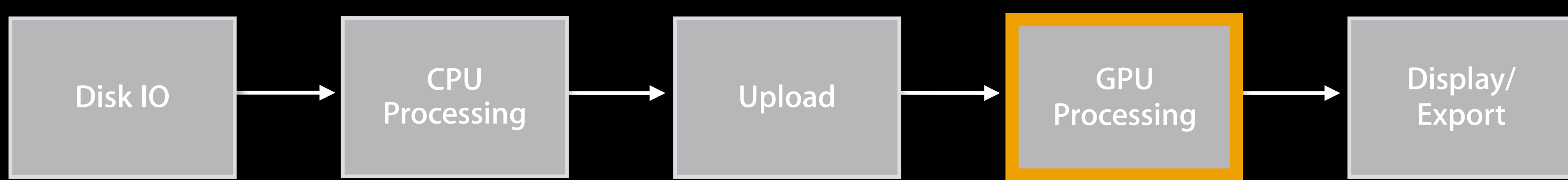
- More OpenCL in future versions
- Investigate OpenCL for CPU

Creating a Mobile Video Workstation with Premiere Pro and OpenCL

David McGavran
Engineering Manager Premiere Pro
Adobe Systems Inc.

Demo

Overall Design



Accelerated Effects

Accelerated Effects

- Intrinsics
  - Adjustment Layers
  - Color Space Conversion
  - Deinterlacing
  - Compositing
  - Blending Modes
  - Nested Sequences
  - Multicam
  - Time Remapping
- Transitions
  - Additive Dissolve
  - Cross Dissolve
  - Dip to Black
  - Dip to White
  - Film Dissolve
  - Push
- Effects
  - Alpha Adjust
  - Black & White
  - Brightness & Contrast
  - Color Balance
  - Color Pass
  - Color Replace
  - Crop
  - Drop Shadow
  - Extract
  - Fast Color Corrector
  - Feather Edges
  - Gamma Correction
  - Garbage Matte
  - Horizontal Flip
  - Invert
  - Luma Corrector
  - Luma Curve
  - Noise
  - Proc Amp
  - RGB Color Corrector
  - RGB Curves
  - Sharpen
  - Three-way Color Corrector
  - Timecode
  - Tint
  - Track Matte
  - Ultra Keyer
  - Veritcal Flip
  - Video Limiter
  - Warp Stabilizer

Tips and Tricks

- Loading kernels asynchronously
- OpenGL texture interop
  - clCreateFromGLBuffer
  - clEnqueueAcquireGLObjects
- OpenCL Images vs. Buffers
- Avoided pinned memory
- Flatten structures
- File RADARS!

Other Possibilities for OpenCL

- Increase set of supported effects
- Supporting third-party effects and codecs
- GPU encoding and decoding
- Multiple GPU support
- GPU Scopes

Wrapping Up

More Information

Allan Schaffer
Graphics and Game Technologies Evangelist
aschaffer@apple.com

Past WWDC Presentations
http://developer.apple.com/videos

Apple Developer Forums
http://devforums.apple.com

Labs

OpenCL Lab	Graphics, Media & Games Lab A Friday 9:00AM

WWDC2012