

1. Update Graphics Drivers
2. Install OpenCL SDK
3. Git clone/download the folder
<https://github.com/premsasidharan/GpgpuOpenCL>
4. Open GpGpuOpenCL/vs2015/GpGpuOpenCL.sln
5. Try to build VectorAdd

HETEROGENEOUS / GPGPU COMPUTING WITH OPENCL

PREM SASIDHARAN

STAFF ENGINEER
DOLBY LABORATORIES

02-01-2018

From Wikipedia ---

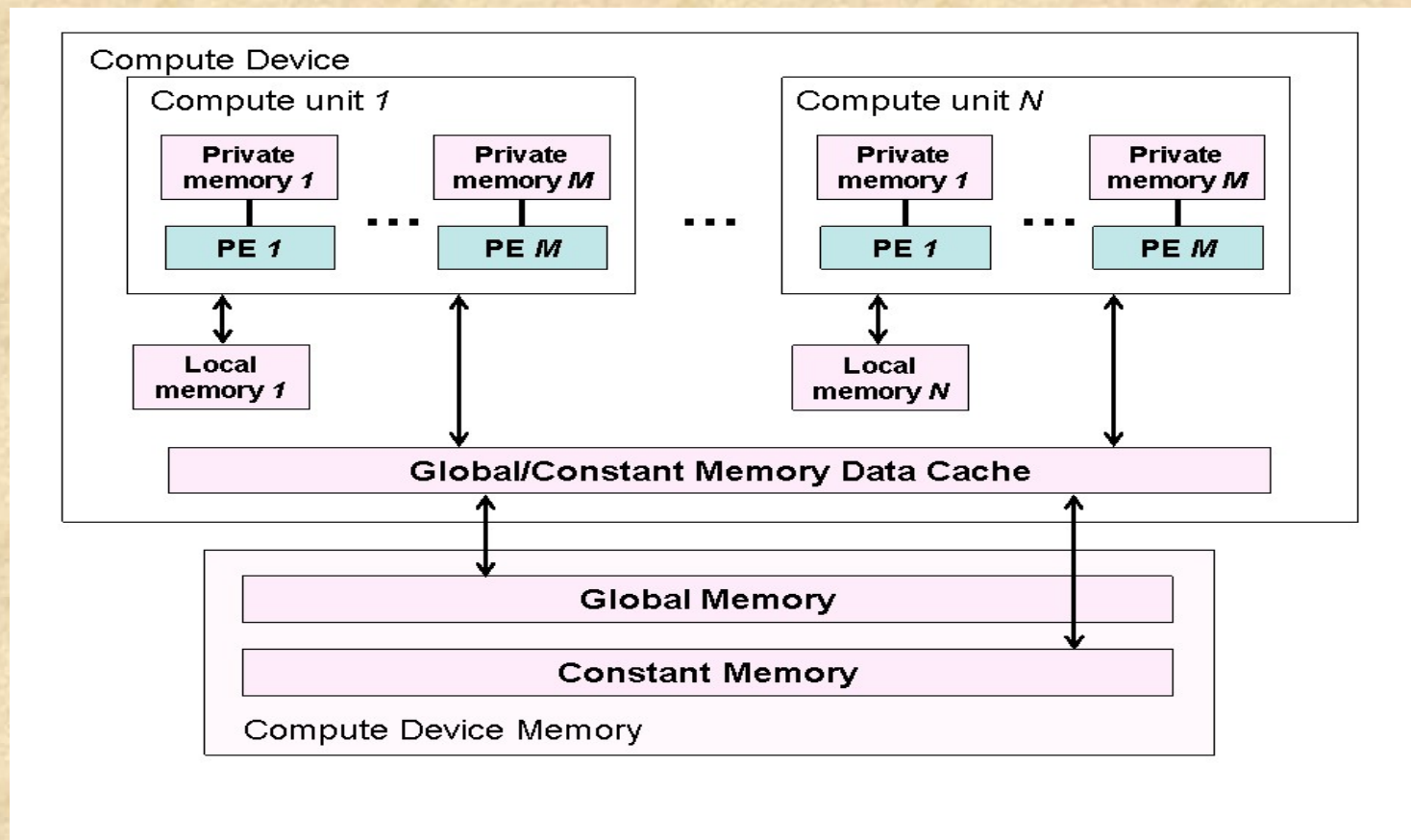
“Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators”

Platform	Silicon Vendor
x86/x64 SoC's (Laptop, Desktop)	Intel, AMD
Mobile SoC's	Qualcomm, Mediatek, Intel Atom, Nvidia Tegra, Samsung Exynos, etc
Discrete GPUs	Nvidia, AMD
FPGA	Xilinx, Altera(Intel)
DSP	TI

Agenda

- Abstract Hardware Architecture
- OpenCL Programming Model
- Coding Example 1: Vector Addition
- Coding Example 2: Color Inversion
- Coding Example 3: Image Gradient
- Coding Example 4: Canny Edge Detection
- Coding Example 5: Image Histogram
- Parallel Processing Patterns
- Coding Example 6: Reduce Sum
- Coding Example 7: Prefix Sum
- Coding Example 8: Harris Corner detection (Stream Compaction)
- OpenCL 2.0
- Coding Example 9: Prefix Sum using OpenCL 2.0
- Coding Example 10: Harris Corner detection using OpenCL 2.0
- Coding Example 11: Radix Sort
- Coding Example 12: Matrix Multiplication
- Reference
- Exercise: RGB Histogram

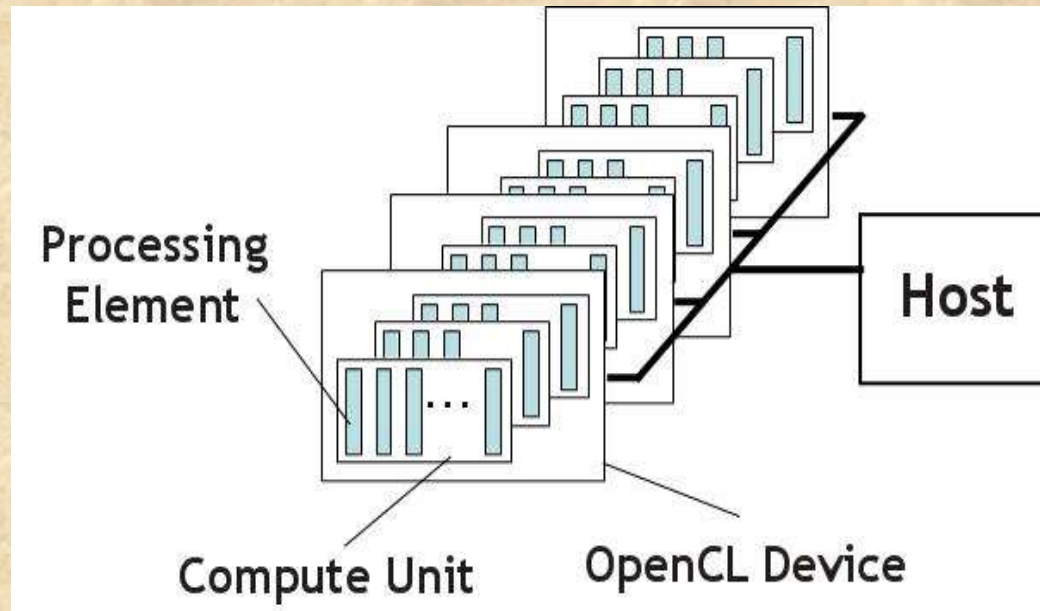
Abstract Hardware Architecture



Processor Architecture

SIMD (Single Instruction Multiple Data)	SIMT (Single Instruction Multiple Threads)	SMT (Simultaneous Multithreading)
<ul style="list-style-type: none">• Vector processing instructions• Performance issues with branch divergence <p>Example: Old GPU architectures</p>	<ul style="list-style-type: none">• A hybrid between vector processing and hardware threading• Not affected by branch divergence <p>Example: New GPU architectures</p>	<ul style="list-style-type: none">• Instructions of several threads run in parallel <p>Example: Modern multicore CPUs</p>

OpenCL Platform Model



- **Data Parallelism**

is a form of parallelization across multiple processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel.

- **Task Parallelism**

is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing tasks concurrently performed by processes or threads across different processors.

Courtesy: Wikipedia

OpenCL Memory Model

- Private Memory (registers)
 - Per work-item
- Local Memory
 - Shared within a work-group
- Global Memory
 - OpenCL Buffers
 - OpenCL Images (1D, 2D and 3D)
- Host Memory

OpenCL Application

- Host Code
 - Written in a High level language (C, C++).
 - Runs on the Host machine
 - Host code issues data transfer commands
 - Host code issues commands for OpenCL device execution
- OpenCL Device Code
 - Written in OpenCL C
 - Runs on OpenCL Device
 - Parallel code executes on many PEs

Work Item (thread): The basic unit of work on an OpenCL device

Kernel: The code for a work Item (like a C Function)

Program: Collection of kernels and other functions

Context: The environment within which work items execute

CommandQueue: Queue used by the host application to submit work to an OpenCL device. Work is queued in order. Work can be executed in-order or out-of-order.

OpenCL Execution Model

Tradition C Code (vector addition)

```
void add(const float* a, const float* b, float* c, int N)
{
    for (int i = 0; i < N; i++)
    {
        c[i] = a[i]+b[i];
    }
}
```

OpenCL Kernel (vector addition)

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

- Kernels are executed across a global domain of work items (threads)
- Work items (threads) can have 1D, 2D or 3D id
- Work item (thread) ID dimension is configurable from the host side depending on the data
- Work items (threads) are grouped into local work-groups (blocks)

- `cl::Platform`
- `cl::Context`
- `cl::Device`
- `cl::CommandQueue`
- `Cl::Event`
- `cl::Program`
- `cl::Kernel`
- `cl::Buffer`, `cl::BufferGL`
- `cl::Image1D`, `cl::Image2D`, `cl::Image3D`, `cl::ImageGL`
- `cl::Sampler`

How to schedule an OpenCL kernel on a GPU (OpenCL Device)?

1. Create a OpenCL context instance

```
cl::Context context(CL_DEVICE_TYPE_GPU);  
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
```

2. Create a CommandQueue instance using the OpenCL context and on the desired OpenCL device.

```
cl::CommandQueue queue(context, devices[0], CL_QUEUE_PROFILING_ENABLE);
```

3. Create the required OpenCL input, output buffers using OpenCL context

```
cl::Buffer bufferA(context, CL_MEM_READ_WRITE, count*sizeof(cl_int));
```

4. Build the OpenCL program

```
cl::Program program(context, sSource);  
program.build();
```

5. Create the OpenCL kernel instance

```
cl::Kernel kernel(program, "add");
```

6. Transfer data from host to OpenCL input buffers(map, or copy) using CommandQueue

```
cl_int* pDataA = (cl_int *)queue.enqueueMapBuffer(bufferC, CL_TRUE, CL_MAP_WRITE, 0, count*sizeof(cl_int));  
for (size_t i = 0; i < count; i++) pDataA[i] = rand()/100;  
queue.enqueueUnmapMemObject(bufferA, pDataA);
```

7. Set Input, Output arguments to kernel instance

```
kernel.setArg(0, bufferA);
```

8. Enqueue the OpenCL kernel on the Command Queue

```
cl::Event event;  
queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(count), cl::NullRange, NULL, &event);
```

9. Wait for the kernel to finish execution

```
event.wait();
```

```
kernel void add(global const int* pA, global const int* pB,  
               global int* pC)  
{  
    const int id = get_global_id(0);  
    pC[id] = (pA[id] + pB[id]);  
}
```


Kernel work item ID/Thread ID

- Global ID (Global ID for the work Item)
 - `size_t get_global_id(size_t dim)`
- Local ID (Work Item ID within the Block)
 - `size_t get_local_id(size_t dim)`
- Work group ID (Work Group ID for the work Item)
 - `size_t get_group_id(size_t dim)`
- `size_t get_global_size(size_t dim)`
- `size_t get_local_size(size_t dim)`
- `size_t get_num_groups(size_t dim)`

Kernel Work Item ID (1D data)

Data (0-255) Block 0	Data (256-511) Block 1	Data (512 – 767) Block 2	...	Data (N-255, N) Block M
--------------------------------	----------------------------------	------------------------------------	-----	-----------------------------------

`size_t get_global_id(0)`

`size_t get_local_id(0)`

`size_t get_group_id(0)`

`size_t get_num_groups(0)`

Kernel Work Item ID (2D Data)

Ex:- Image with 32x24 pixels organized into blocks of size 8x8

Block (0, 0)	Block (1, 0)	Block (2, 0)	Block(3, 0)
Block(0, 1)	Block(1, 1)	Block(2, 1)	Block(3, 1)
Block(0, 2)	Block(1, 2)	Block(2, 2)	Block(3, 2)

8, 8	9, 8	10, 8	11, 8	12, 8	13, 8	14, 8	15, 8
8, 9	9, 9	10, 9	11, 9	12, 9	13, 9	14, 9	15, 9
8, 10	9, 10	10, 10	11, 10	12, 10	13, 10	14, 10	15, 10
8, 11	9, 11	10, 11	11, 11	12, 11	13, 11	14, 11	15, 11
8, 12	9, 12	10, 12	11, 12	12, 12	13, 12	14, 12	15, 12
8, 13	9, 13	10, 13	11, 13	12, 13	13, 13	14, 13	15, 13
8, 14	9, 14	10, 14	11, 14	12, 14	13, 14	14, 14	15, 14
8, 15	9, 15	10, 15	11, 15	12, 15	13, 15	14, 15	15, 15

size_t get_global_id(0)
size_t get_global_id(1)

size_t get_local_id(0)
size_t get_local_id(1)

size_t get_group_id(0)
size_t get_group_id(1)

size_t get_global_size(0)
size_t get_global_size (1)

size_t get_local_size(0)
size_t get_local_size (1)

size_t get_num_groups(0)
size_t get_num_groups(1)

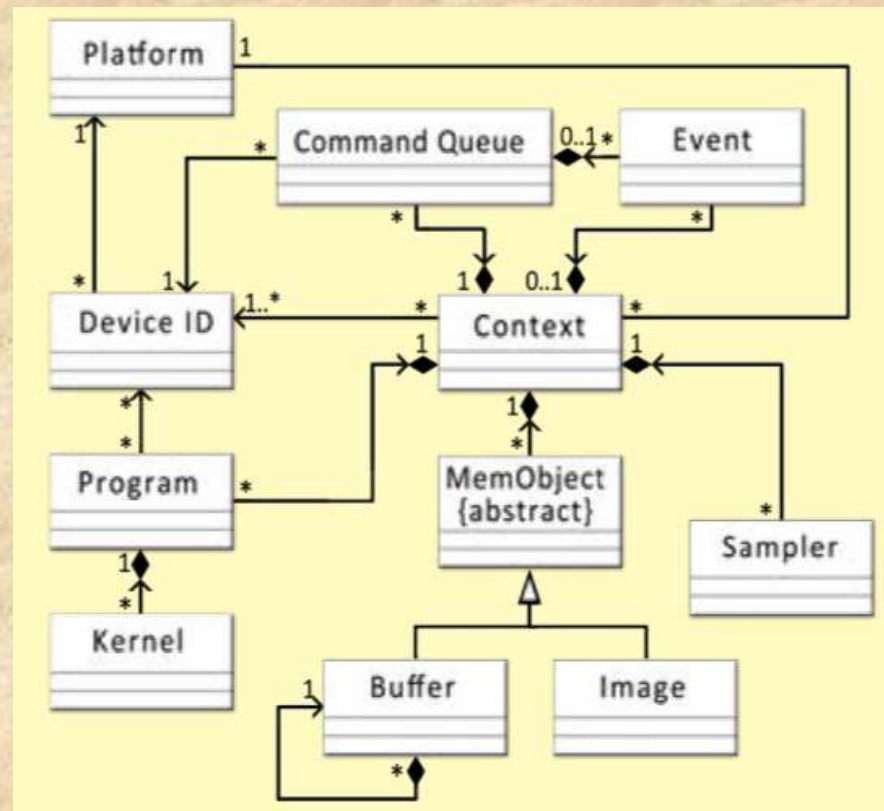
Coding Example 1:

Vector Addition

Summary (Vector Addition)

- Learned to launch a basic kernel on the GPU
- Optimization tips
 - Utilize the available memory band width per processing element
 - Use int4 instead of int and process the 4 elements within the thread

OpenCL Programming Model



Host	Kernel
<pre> cl::Buffer, cl::BufferGL size_t count = 1024; cl::Buffer buffA(context, CL_MEM_READ_WRITE, count*sizeof(cl_int)); ... kernel.setArg(0, buffA); kernel.setArg(1, (cl_int)count); ... </pre>	<pre> kernel void test(write_only global int* pData, int count) { int id = get_global_id(0); pData[id] = value; } </pre>
<pre> cl::Image2D, cl::ImageGL cl::Image2D inImg(context, CL_MEM_READ_ONLY, cl::ImageFormat(CL_RGBA, CL_FLOAT), w, h); cl::Image2D outImg(context, CL_MEM_READ_WRITE, cl::ImageFormat(CL_R, CL_FLOAT), w, h); kernel.setArg(0, inImg); kernel.setArg(1, outImg); ... </pre>	<pre> kernel void avg(read_only image2d_t inp, write_only image2d_t out) { const int x = get_global_id(0); const int y = get_global_id(1); float4 color = read_imagef(inp, (int2)(x, y)); float idata = (color.x+color.y+color.z)/3.0; write_imagef(out, (int2)(x, y), idata) } </pre>

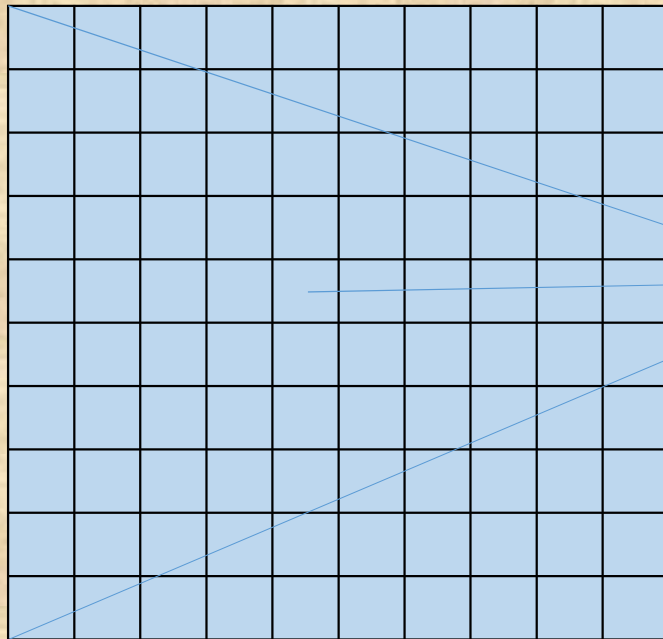
Inter-op with OpenGL

- OpenCL Context Creation with OpenGL on Windows

```
cl_context_properties props[] =  
    { CL_GL_CONTEXT_KHR, (cl_context_properties)wglGetCurrentContext(),  
      CL_WGL_HDC_KHR, (cl_context_properties)wglGetCurrentDC(),  
      CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[i])(), 0 };  
  
mCtxtCL.reset(new cl::Context(CL_DEVICE_TYPE_GPU, props));
```

- OpenGL Buffers, Textures can be converted to OpenCL buffers and Images.
 - cl::BufferGL, cl::ImageGL
 - cl::BufferGL buff_gl(ContextCL, CL_MEM_READ_WRITE, glBufferID)
 - cl::ImageGL img_gl(ContextCL, CL_MEM_READ_ONLY, GL_TEXTURE_2D, 0, glTextureID)
 - std::vector<cl::Memory> gl_objs = { buff_gl, img_gl };
 - queueCL.enqueueAcquireGLObjects(&gl_objs)
 -
 - queueCL.enqueueReleaseGLObjects(&gl_objs);
- For some GPUs (**Intel HD graphics**) interop works only if OpenGL texture internal format is float32 or float16 (GL_R32F, GL_R16F, GL_RG32F, GL_RGBA32F etc)
void glTexImage2D(GLenum target, GLint level, **GLint internalformat**, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * data)

Memory



Texture Cache

Sampler

cl::image1D
cl::image2D
cl::image3D
cl::imageGL

Normalized-mode	CLK_NORMALIZED_COORDS_{TRUE, FALSE}
Address-mode	CLK_ADDRESS_{REPEAT, CLAMP, CLAMP_TO_EDGE, MIRRORED_REPEAT, NONE}
Filter-mode	CLK_FILTER_NEAREST, CLK_FILTER_LINEAR

Coding Example 2: Color Inversion

D E V I C E C O D E	<pre>kernel void invert_color(read_only image2d_t inpImg, write_only image2d_t outImg) { int2 coord = (int2)(get_global_id(0), get_global_id(1)); float3 idata = read_imagef(inpImg, coord).xyz; float4 odata = (float4)(1.0f-idata.x, 1.0f-idata.y, 1.0f-idata.z, 1.0f); write_imagef(outImg, coord, odata); }</pre>
H O S T	<pre>cl::ImageGL inImgGL(mCtxtCL, CL_MEM_READ_ONLY, GL_TEXTURE_2D, 0, mBgrImg()); cl::ImageGL outImgGL(mCtxtCL, CL_MEM_WRITE_ONLY, GL_TEXTURE_2D, 0, mInvImg()); std::vector<cl::Memory> gl_objs = { inImgGL, outImgGL }; cl::Event event; mKernel.setArg(0, inImgGL); mKernel.setArg(1, outImgGL); mQueueCL.enqueueAcquireGLObjects(&gl_objs); mQueueCL.enqueueNDRangeKernel(mKernel, cl::NullRange, cl::NDRange(mBgrImg.width(), mBgrImg.height()), cl::NDRange(8, 8), NULL, &event); event.wait(); mQueueCL.enqueueReleaseGLObjects(&gl_objs);</pre>

Coding Example 3:

Image Gradient

	MaskX	
-1.0f	+0.0f	+1.0f
-2.0f	+0.0f	+2.0f
-1.0f	+0.0f	+1.0f

	MaskY	
+1.0f	+2.0f	+1.0f
+0.0f	+0.0f	+0.0f
-1.0f	-2.0f	-1.0f

1. Apply MaskX on Every Pixel and Compute I_x
2. Apply MaskY on Every Pixel and Compute I_y
3. $I = \sqrt{(I_x * I_x) + (I_y * I_y)}$

Summary: Image Gradient

```
kernel void gradient(read_only image2d_t inImg, write_only image2d_t outImg, global const float* plx, global const float* ply)
{
    const int x = get_global_id(0);
    const int y = get_global_id(1);
    const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE|CLK_ADDRESS_CLAMP|CLK_FILTER_LINEAR;

    float ix = 0.0f;
    float iy = 0.0f;
    for (int i = -1; i <= +1; i++)
    {
        for (int j = -1; j <= +1; j++)
        {
            int index = (3*(i+1))+(j+1);
            ix += (plx[index]*read_imagef(inImg, sampler, (int2)(x+j, y+i)).x);
            iy += (ply[index]*read_imagef(inImg, sampler, (int2)(x+j, y+i)).x);
        }
    }
    float odata = sqrt((ix*ix)+(iy*iy));
    write_imagef(outImg, (int2)(x, y), odata);
}
```

- OpenGL OpenCL interop
- Texture cache (cl::Image)
- Samplers
 - Normalized-mode: CLK_NORMALIZED_COORDS_{TRUE, FALSE}
 - Address-mode: CLK_ADDRESS_{REPEAT, CLAMP, CLAMP_TO_EDGE, MIRRORED_REPEAT, NONE}
 - Filter-mode: CLK_FILTER_NEAREST, CLK_FILTER_LINEAR
- Constant Memory (Accessed through cache)

Coding Example 4: Canny Edge Detection

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Summary: Canny Edge Detection

- Event wait lists
- Chaining kernel's using Event wait list
- Measuring Kernel Execution time

```
cl::CommandQueue queue(context, devices[0], CL_QUEUE_PROFILING_ENABLE);
...
...

bool kernelExecTime(const cl::CommandQueue& queue, const cl::Event& event, int& time)
{
    cl_command_queue_properties qProp;
    queue.getInfo<cl_command_queue_properties>(CL_QUEUE_PROPERTIES, &qProp);

    if (qProp & CL_QUEUE_PROFILING_ENABLE)
    {
        int start_time = event.getProfilingInfo<CL_PROFILING_COMMAND_START>();
        int end_time = event.getProfilingInfo<CL_PROFILING_COMMAND_END>();
        time = (end_time-start_time);
        return true
    }
    time = 0;
    return false;
}
```


- Barrier

- All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. This function must be encountered by all work-items in a work-group executing the kernel.
- If barrier is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the barrier.
- If barrier is inside a loop, all work-items must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier
- CLK_LOCAL_MEM_FENCE, CLK_GLOBAL_MEM_FENCE.

- Atomic Operations

- atomic_add, atomic_sub, atomic_xchg, atomic_inc, atomic_dec, atomic_cmpxchg, atomic_min, atomic_max, atomic_and, atomic_or, atomic_xor

Coding Example 5:

Image Histogram computation

```
kernel void init(global int* pHistData)
{
    pHistData[get_global_id(0)] = 0;
}

kernel void histogram(read_only image2d_t inImg, global int* pHistData)
{
    const int x = get_global_id(0);
    const int y = get_global_id(1);
    int idata = (int)ceil(255.0f*read_imagef(inImg, (int2)(x, y)).x);
    atomic_inc(&pHistData[idata]);
}
```


How to improve performance?

- Atomic operation on global memory is very slow
- Intermediate Image Histogram using local (shared) memory and atomic ops
- Accumulate the intermediate histograms using atomic addition to global memory
- Histogram without atomics ??? [\[Exercise\]](#)

```
kernel void init(global int* pHistData)
{
    pHistData[get_global_id(0)] = 0;
}

kernel void histogram(read_only image2d_t inImg, global int* pHistData)
{
    local int shHistData[256];
    const int x = get_global_id(0);
    const int y = get_global_id(1);

    int index = (get_local_size(0)*get_local_id(1)) + get_local_id(0);
    if (index < 256) shHistData[index] = 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    int idata = (int)ceil(255.0f*read_imagef(inImg, (int2)(x, y)).x);
    atomic_inc(&shHistData[idata]);
    barrier(CLK_LOCAL_MEM_FENCE);

    if (index < 256) atomic_add(&pHistData[index], shHistData[index]);
}
```


More performance improvement

- Write the intermediate histogram's into global memory
- Accumulate the intermediate histogram data using a second kernel
 - Accumulation using Simple addition
 - Accumulation using reduce sum

[illegible]

Parallel Processing Patterns

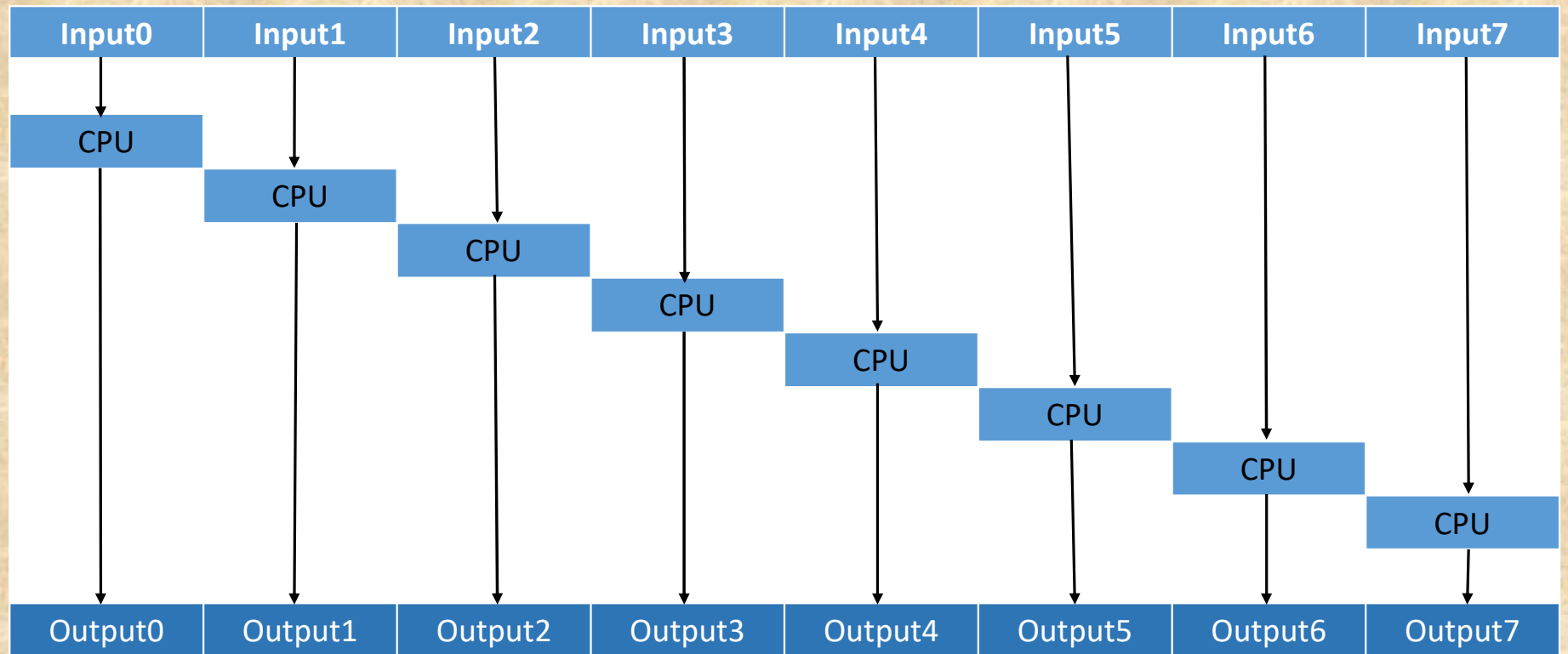
- Map
- Stencil
- Gather
- Scatter
- Reduction (Eg:- sum, min, max etc)
- Scan
 - Inclusive Sum (Prefix Sum)
 - Exclusive Sum
- Stream Compaction
- Histogram

Map

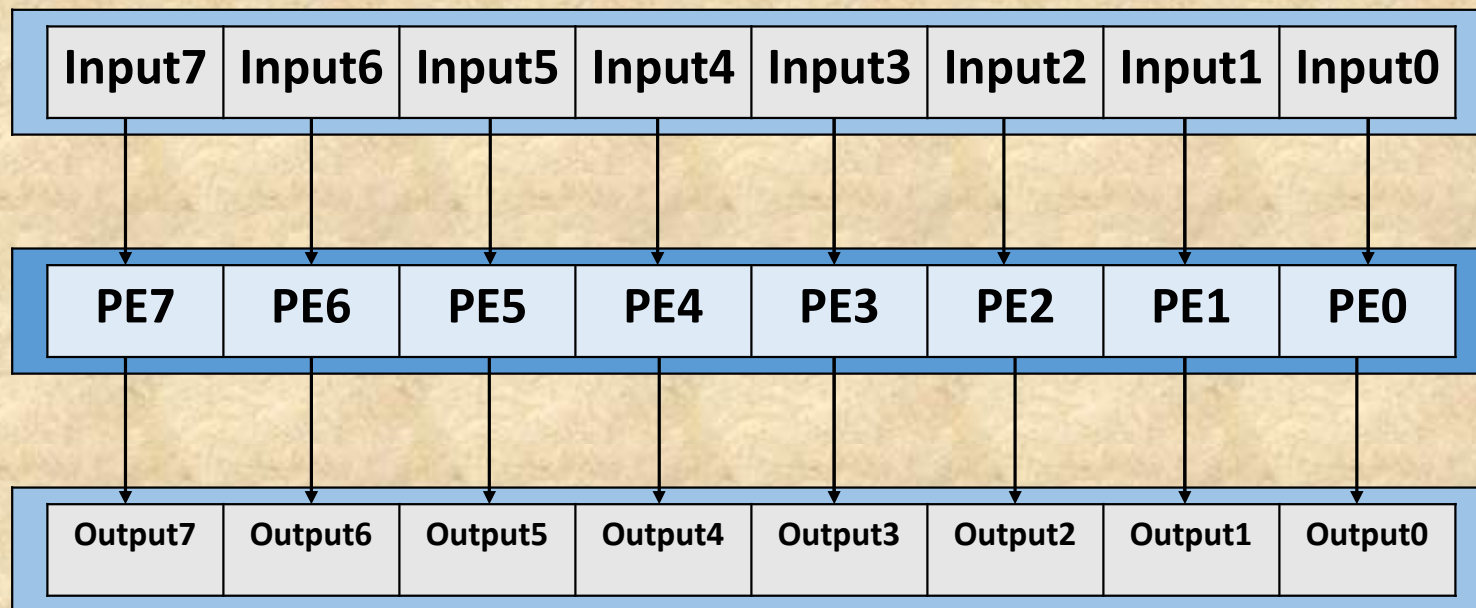
- In the map pattern a function, which we call an elemental function, is applied to different data.

Ex:- vector addition, color inversion, image gradient

Map (Serial)



Map (Parallel)



- Gather

- Given a collection of locations (addresses or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an output collection.

- Scatter

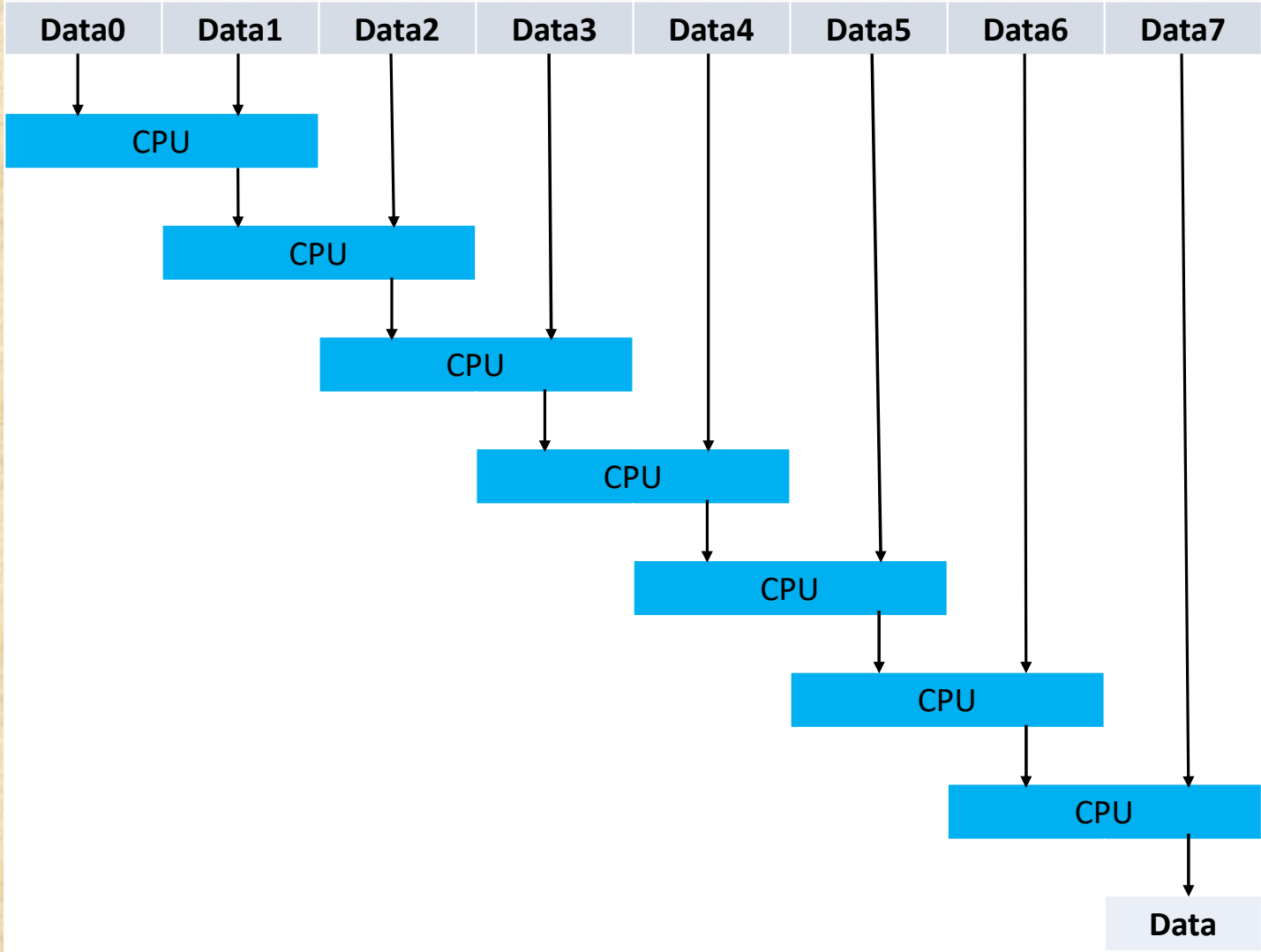
- Scatter is similar to gather, but write locations rather than read locations are provided as input.

Reduce

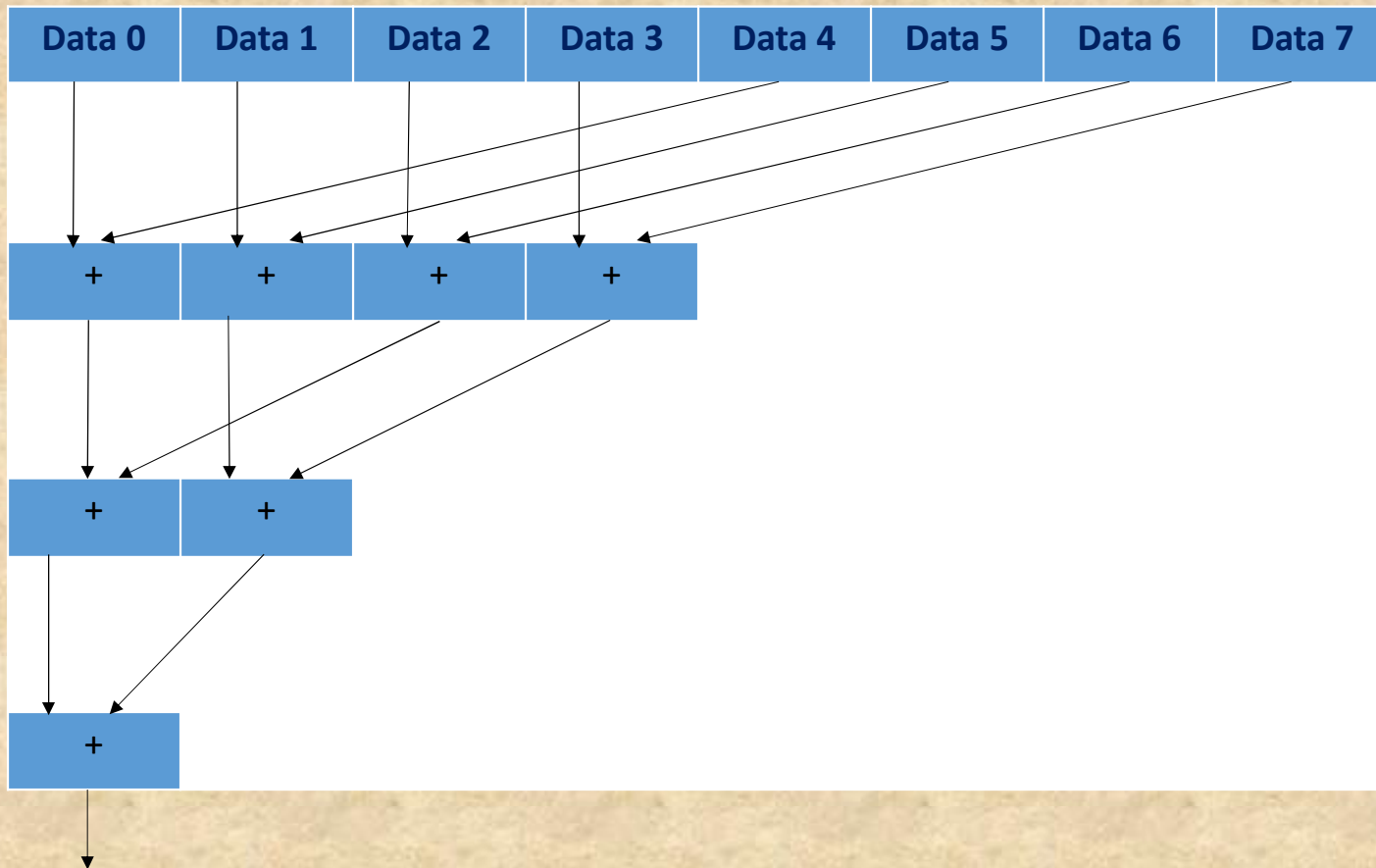
The reduce pattern allows data to be summarized; it combines all the elements in a collection into a single element using some associative combiner operator.

Ex:- min, max, sum, etc

Serial
Reduction



Parallel Reduction



Parallel Reduction (Sum)

Index	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Initial Data in shared Memory
Step 1									10	12	14	16	18	20	22	24	If (id < 8) data[id] += data[id+8]
Step 2													28	32	36	40	If (id < 4) data[id] += data[id+4]
Step 3															64	72	If (id < 2) data[id] += data[id+2]
Step 4																136	If (id < 1) data[id] += data[id+1]

Block(workgroup) Reduction

Load the Elements into shared memory. For example if the Block contains 256 work items, Bring 512 data items into shared memory and do the reduction as described in the next block.

```
int bSize = get_local_size(0);
int i = get_local_id(0);
While (bSize > 0)
{
    if (i < bSize) data[i] += data[bSize+i];
    barrier(CLK_LOCAL_MEM_FENCE);
    bSize /= 2;
}
```

Coding Example 6: Reduce Sum

DEVICE CODE	<pre>kernel void reduceSum(global const int* pInput, global int* pOutput, const int count) { local int shData[SH_MEM_SIZE]; int i = get_local_id(0); int id = (SH_MEM_SIZE*get_group_id(0))+i; shData[i] = (id < count)?pInput[id]:0; i += get_local_size(0); id += get_local_size(0); shData[i] = (id < count)?pInput[id]:0; barrier(CLK_LOCAL_MEM_FENCE); breduce_sum(shData); if (get_local_id(0) == 0) pOutput[get_group_id(0)] = shData[0]; }</pre>	<pre>inline void breduce_sum(local volatile int* shData) { int bSize = get_local_size(0); while (bSize > 0) { int i = get_local_id(0); if (i < bSize) shData[i] += shData[bSize+i]; barrier(CLK_LOCAL_MEM_FENCE); bSize /= 2; } }</pre>
HOST CODE	<pre>size_t count = 1000000; size_t gSize = count/256; if ((count % 256) != 0) ++gSize; cl::Buffer buffInp(context, CL_MEM_READ_ONLY, count*sizeof(cl_int)); cl::Buffer buffOut(context, CL_MEM_READ_WRITE, gSize*sizeof(cl_int)); reduce.setArg(0, buffInp); reduce.setArg(1, buffOut); reduce.setArg(2, (int)count); queue.enqueueNDRangeKernel(reduce, cl::NullRange, cl::NDRange(256*gSize), cl::NDRange(128), NULL, &events[evtCount]); while (gSize > 1) { reduce.setArg(0, buffOut); reduce.setArg(1, buffOut); reduce.setArg(2, (int)gSize); wEvents[wEvtCount].push_back(events[evtCount++]); size_t ngSize = (gSize/256)+(((gSize%256)!=0)?1:0); queue.enqueueNDRangeKernel(reduce, cl::NullRange, cl::NDRange(ngSize*256), cl::NDRange(128), &wEvents[wEvtCount++], &events[evtCount]); gSize = ngSize; } events[evtCount].wait();</pre>	

Optimized Reduce Sum

```
inline void breduce_sum(local volatile int* sh_data)
{
    if (get_local_id(0) < 256) sh_data[get_local_id(0)] += sh_data[256+get_local_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);

    if (get_local_id(0) < 128) sh_data[get_local_id(0)] += sh_data[128+get_local_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);

    if (get_local_id(0) < 64) sh_data[get_local_id(0)] += sh_data[64+get_local_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);

    if (get_local_id(0) < 32) sh_data[get_local_id(0)] += sh_data[32 + get_local_id(0)];
    if (get_local_id(0) < 16) sh_data[get_local_id(0)] += sh_data[16 + get_local_id(0)];
    if (get_local_id(0) < 8) sh_data[get_local_id(0)] += sh_data[8 + get_local_id(0)];
    if (get_local_id(0) < 4) sh_data[get_local_id(0)] += sh_data[4 + get_local_id(0)];
    if (get_local_id(0) < 2) sh_data[get_local_id(0)] += sh_data[2 + get_local_id(0)];
    if (get_local_id(0) < 1) sh_data[get_local_id(0)] += sh_data[1 + get_local_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

No need to place
barrier instruction
within WARP

Scan

- The scan collective operation produces all partial reductions of an input sequence, resulting in a new output sequence. There are two variants: inclusive scan and exclusive scan. For inclusive scan, the n th output value is a reduction over the first n input values. For exclusive scan, the n th output value is a reduction over the first $n-1$ input values.

Prefix Sum

- The prefix sum, scan, or cumulative sum of a sequence of numbers x_0, x_1, x_2, \dots is a second sequence of numbers y_0, y_1, y_2, \dots , the sums of prefixes of the input sequence: $y_0 = x_0$ $y_1 = x_0 + x_1$ $y_2 = x_0 + x_1 + x_2 \dots$

Input	1	2	3	4	5	6	7	8
Prefix Sum(Inclusive Sum)	1	3	6	10	15	21	28	36

Input	1	2	3	4	5	6	7	8
Exclusive Sum	0	1	3	6	10	15	21	28

Prefix Sum

ID	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Step 1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		if (id >= 1) data[id] += data[id-1]
Step 2	4	4	4	4	4	4	4	4	4	4	4	4	4	3	2		if (id >= 2) data[id] += data[id-2]
Step 3	8	8	8	8	8	8	8	8	8	7	6	5					if (id >= 4) data[id] += data[id-4]
Step 4	16	15	14	13	12	11	10	9									if (id >= 8) data[id] += data[id-8]

```

inline void bscan(local volatile int* shData)
{
    int level = 1;
    const int id = get_local_id(0);
    while (level < SH_MEM_SIZE)
    {
        if (id >= level) shData[id] += shData[id-level];
        barrier(CLK_LOCAL_MEM_FENCE);
        level *= 2;
    }
}

```


Efficient Prefix Sum on a GPU workgroup

SIMD7 PE (255-224)	SIMD6 PE (223-192)	SIMD5 PE (191-160)	SIMD4 PE (159-128)	SIMD3 PE (127-96)	SIMD2 PE (95-64)	SIMD1 PE (63-32)	SIMD0 PE (31-0)
-----------------------	-----------------------	-----------------------	-----------------------	----------------------	---------------------	---------------------	--------------------

For most of the GPUs the workgroup contains eight 32-wide SIMD processors. (Nvidia uses the term WARP to denote these SIMD processors, AMD calls them WAVEFRONT).

Follow the below steps for doing a block scan (Data is available in shared memory)

1. Perform Eight WARP/WAVEFRONT/32 wide prefix sum on the shared Memory (WARP wide sum). Eight 32-wide prefix sums are available in memory.
2. Grab the 31st, 63rd, 95th, 127th, 159th, 191th, 223rd and 255th elements and place them in shared memory variable

255th(7th)	223rd(6th)	191th(5th)	159th(4th)	127th(3rd)	95th(2nd)	63rd(1st)	31 st (0)
------------	------------	------------	------------	------------	-----------	-----------	----------------------

3. Do a warp prefix sum on the above shared memory variable
4. Add the above elements WARP wide on the WARP wide sum (add 0th element to elements 63-32, add 1st elements 95-64, add 2nd element to elements 127-96, add 3rd element to elements 159-128, add 4th element to elements 191-160, add 5th element to elements 223-192 and 6th element to elements 255-224)

D0	D1	D30	D31
D32	D33	D62	D63
D64	D65	D94	D95
D96	D97	D126	D127
D128	D129	D158	D159
D160	D161	D190	D191
D192	D193	D232	D233
D234	D235	D254	D255

SIMD/T 0
SIMD/T 1
SIMD/T 2
SIMD/T 3
SIMD/T 4
SIMD/T 5
SIMD/T 6
SIMD/T 7

R0_0	R0_1	R0_30	R0_31
R1_0	R1_1	R1_30	R1_31
R2_0	R2_1	R2_30	R2_31
R3_0	R2_1	R3_30	R3_31
R4_0	R2_1	R4_30	R4_31
R5_0	R2_1	R5_30	R5_31
R6_0	R2_1	R6_30	R6_31
R7_0	R2_1	R7_30	R7_31

Warp Level
Prefix Sum

P0	P1	P30	P31
P32	P33	P62	P63
P64	P65	P94	P95
P96	P97	P126	P127
P128	P129	P158	P159
P160	P161	P190	P191
P192	P193	P232	P233
P234	P235	P254	P255

R0_0	R0_1	R0_30	R0_31
R1_0	R1_1	R1_30	R1_31
R2_0	R2_1	R2_30	R2_31
R3_0	R2_1	R3_30	R3_31
R4_0	R2_1	R4_30	R4_31
R5_0	R2_1	R5_30	R5_31
R6_0	R2_1	R6_30	R6_31
R7_0	R2_1	R7_30	R7_31

SIMD/T 0	S0
SIMD/T 1	S1
SIMD/T 2	S2
SIMD/T 3	S3
SIMD/T 4	S4
SIMD/T 5	S5
SIMD/T 6	S6
SIMD/T 7	S7

SIMD/
T

(WARP
LEVEL
PREFIX
SUM)

Coding Example 7: Prefix Sum

```
inline void wscan(int i, local volatile int* sh_data)
{
    const int wid = i & (WARP_SIZE-1);
    if (wid >= 1) sh_data[i] += sh_data[i-1];
    if (wid >= 2) sh_data[i] += sh_data[i-2];
    if (wid >= 4) sh_data[i] += sh_data[i-4];
    if (wid >= 8) sh_data[i] += sh_data[i-8];
    if (WARP_SIZE == 32)
    {
        if (wid >= 16) sh_data[i] += sh_data[i-16];
    }
}

inline void bscan(int i, local volatile int* shData, local volatile int* shDataT)
{
    wscan(i, shData);
    barrier(CLK_LOCAL_MEM_FENCE);

    if (i < (SH_MEM_SIZE/WARP_SIZE)) shDataT[i] = shData[(WARP_SIZE*i)+(WARP_SIZE-1)];
    barrier(CLK_LOCAL_MEM_FENCE);

    if (i < WARP_SIZE) wscan(i, shDataT);
    barrier(CLK_LOCAL_MEM_FENCE);

    if (i >= WARP_SIZE) shData[i] += shDataT[(i/WARP_SIZE)-1];
    barrier(CLK_LOCAL_MEM_FENCE);
}

kernel void scan(global const int* pInput, global int* pOutput, const int count)
{
    local int shData[SH_MEM_SIZE+WARP_SIZE];
    int i = get_local_id(0);
    int id = get_global_id(0);
    shData[i] = (id < count)?pInput[id]:0;
    bscan(i, shData, &shData[SH_MEM_SIZE]);
    if (id < count) pOutput[id] = shData[i];
}
```

```
kernel void gather(global const int* pInput, global int* pOutput, const int start, const int count)
{
    local int shData[SH_MEM_SIZE+WARP_SIZE];
    int i = get_local_id(0);
    int id = start+(get_local_size(0)*get_local_id(0));
    shData[i] = (id < count)?pInput[id]:0;
    bscan(i, shData, &shData[SH_MEM_SIZE]);
    pOutput[i] = shData[i];
}

kernel void add(global const int* pInput, global int* pOutput, const int start, const int count)
{
    local int shData;
    if (get_local_id(0) == 0) shData = pInput[get_group_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);
    const int id = start+get_global_id(0);
    if (id < count) pOutput[id] += shData;
}
```


Prefix Sum on a binary (0 or 1) sequence

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Input	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0
Inc. Scan	0	0	0	0	1	1	2	2	2	2	2	3	3	3	3	4	4	4	5	5
Ex. Scan	0	0	0	0	0	0	1	1	1	1	1	2	2	2	2	3	3	3	4	4

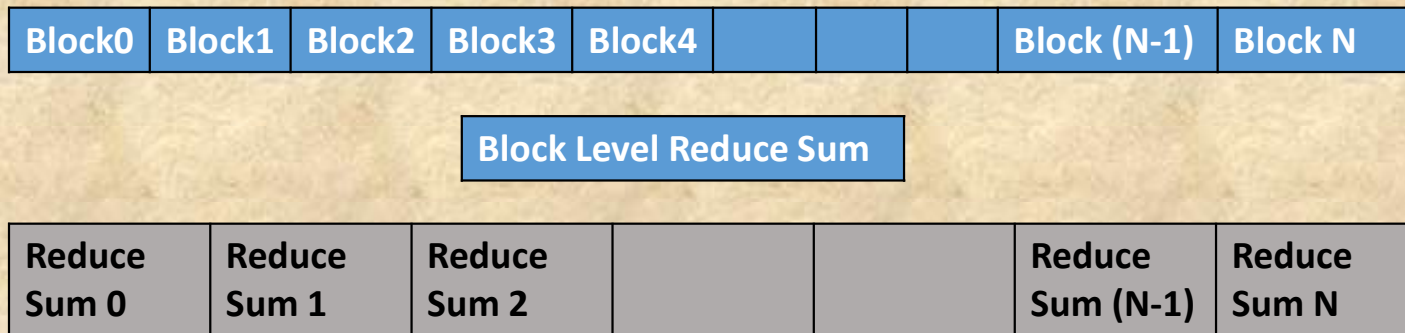
Indexes with 1's	Inc. Scan Value	Ex. Scan Value
4	1	0
6	2	1
11	3	2
15	4	3
18	5	4

Prefix Sum Applications

- Stream Compaction
- Radix-sort
- Parallel Quick-sort
- Integral Image
- Polynomial Evaluation
- Histogram
- Lexical Analysis
- Solving recurrences

Faster Stream Compaction

1. Do block level reduce sum's on the input and store the outputs on global memory.



1. Do an Exclusive Prefix Sum on the block level reduce sum's
2. Add the **results** in Step2 block wise to all elements in the block.
(add the value at index 0 to all elements in block 0, add the value at index 1 to all elements in block1, add the value at index N to all elements in blockN)

Coding Example 8:

Harris Corner Detection

1. Compute I_x (gradient x) and I_y (gradient y) at every pixel
2. Compute the product of derivative at every pixel
 1. $I_x^2 = I_x * I_x$
 2. $I_y^2 = I_y * I_y$
 3. $I_{xy} = I_x * I_y$
3. Apply Gaussian smoothing on I_x^2 , I_y^2 , I_{xy} and compute $G_{I_x^2}$, $G_{I_y^2}$, and $G_{I_{xy}}$
4. Define at each pixel Matrix H

$G_{I_x^2}$	$G_{I_{xy}}$
$G_{I_{xy}}$	$G_{I_y^2}$

5. Calculate $R = \det(H) - k * \text{trace}(H) * \text{trace}(H)$
6. Threshold on value of R and due non max suppression
7. Do stream compaction

OpenCL 2.0

- Dynamic parallelism (Nested parallelism)
 - Device side kernels capable of launching other kernels
- Built-in work group functions
 - Scan (Inclusive, Exclusive)
 - Reduce (Min, Max, Sum etc)
- Shared virtual memory
 - Map Free access
 - Fine Grained Coherent access
 - Fine Grained Synchronization
 - Complex Data structures with Pointers (eg:- BTree, Graphs etc)

```
cl::Context context(CL_DEVICE_TYPE_GPU);
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
cl::CommandQueue queue(context, devices[0], CL_QUEUE_PROFILING_ENABLE);

cl_queue_properties qprop[] = { CL_QUEUE_PROPERTIES,
    (cl_command_queue_properties)(CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE|CL_QUEUE_ON_DEVICE|
    CL_QUEUE_ON_DEVICE_DEFAULT|CL_QUEUE_PROFILING_ENABLE), 0 };

int err = 0;
cl_command_queue dev_q = clCreateCommandQueueWithProperties(context(), devices[0](), qprop, &err);

.....

std::ostringstream options;
options << "-cl-std=CL2.0";

cl::Program program(context, sSource);
program.build(options.str().c_str());
```


Coding Example 9: Prefix Sum using OpenCL 2.0

```
kernel void scan(global int* pdata, global int* tempData, int count)
{
    clk_event_t event1;
    clk_event_t event2;
    int ncount = count/4;
    int ncount4 = ncount*4;
    int gcount = ((ncount/BLK_SIZE)+(((ncount%
    const int BLK_SIZE = 256;
    const int BLK_SIZE2 = (BLK_SIZE*BLK_SIZE); BLK_SIZE) == 0)?0:1))*BLK_SIZE;

    enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT,
        ndrange_1D(gcount, BLK_SIZE), 0, 0, &event1, ^{ child_scan(pdata, ncount); });

    for (int i = (4*BLK_SIZE); i < ncount4; i += (16*BLK_SIZE2))
    {
        enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(BLK_SIZE, BLK_SIZE), 1,
            &event1, &event2, ^{ gather_scan(pdata, tempData, i, ncount4); });
        enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D((4*BLK_SIZE2), BLK_SIZE), 1,
            &event2, &event1, ^{ add_data(pdata, tempData, i, ncount); });
    }
    enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(1, 1), 1,
        &event1, &event2, ^{ add_remainders(&pdata[ncount4], count%4); });
    release_event(event2);
}
```

Coding Example 10:

Harris Corner Detection using OpenCL 2.0

Radix Sort

Input Data

Data	Binary
6	0110
13	1101
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

**Re-ordering based on bit position 0
(Stream compaction with the mask
b'0001)**

Data	Binary
6	0110
2	0010
4	0100
6	0110
8	1000
13	1101
3	0011
5	0101
7	0111
9	1001

**Re-ordering based on bit position 1
(Stream compaction with the mask
b'0010)**

Data	Binary
4	0100
8	1000
13	1101
5	0101
9	1001
6	0110
2	0010
6	0110
3	0011
7	0111

**Re-ordering based on bit position 2
(Stream compaction with the mask
b'0100)**

Data	Binary
8	1000
9	1001
2	0010
3	0011
4	0100
13	1101
5	0101
6	0110
6	0110
7	0111

**Re-ordering based on bit position 3
(Stream compaction with the mask
b'1000)**

Data	Binary
2	0010
3	0011
4	0100
5	0101
6	0110
6	0110
7	0111
8	1000
9	1001
13	1101

Coding Example 11: Radix Sort

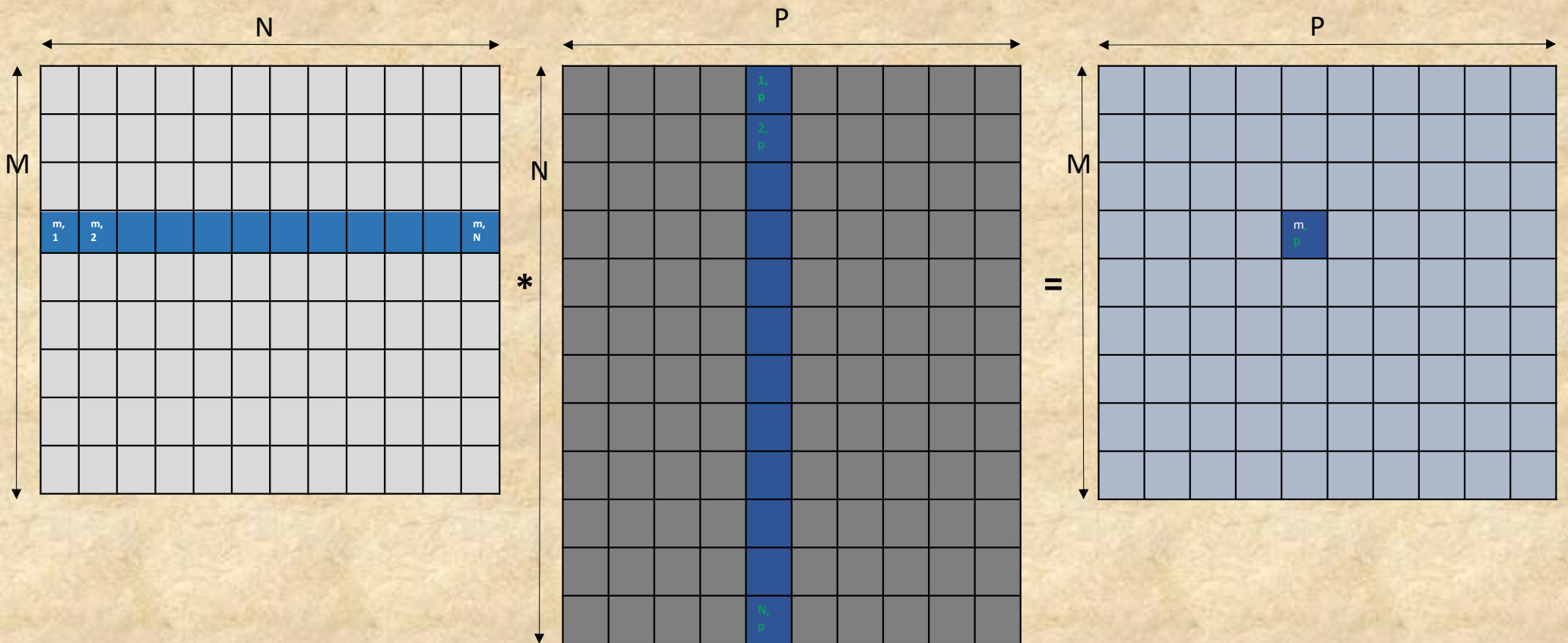
```
kernel void radix_sort(global int* p_in_data, global int* p_temp_data, int count)
{
    clk_event_t event[4];
    const int BLK_SIZE = 256;
    int size = 4+(count/256);
    int temp_buff_size = 1024*((size/1024)+((size%1024)==0?0:1));

    global int* p_odd_blk_count = p_temp_data + temp_buff_size;
    global int* p_even_blk_count = p_odd_blk_count + size;
    global int* p_out_data = p_even_blk_count + size;

    unsigned int mask = 0x0001;
    for (int i = 0; mask != 0; i++)
    {
        global int* p_input_data = ((i%2) == 0)?p_in_data:p_out_data;
        global int* p_output_data = ((i%2) == 0)?p_out_data:p_in_data;

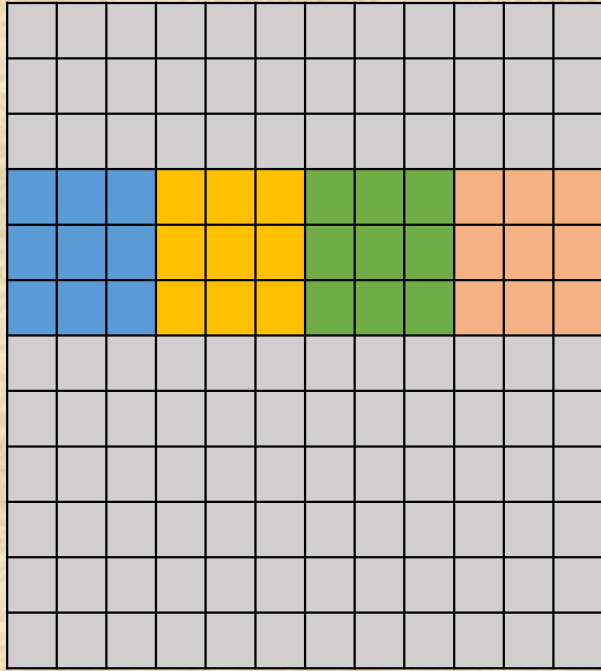
        enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(count, BLK_SIZE), (i==0)?0:1,
            (i==0)?0:&event[3], &event[0], ^{ block_even_odd_count(p_input_data, mask, p_even_blk_count, p_odd_blk_count); });
        enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(1, 1), 1,
            &event[0], &event[1], ^{ scan(p_even_blk_count, p_temp_data, size); });
        enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(1, 1), 1,
            &event[0], &event[2], ^{ scan(p_odd_blk_count, p_temp_data, size); });
        enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(count, BLK_SIZE), 2,
            &event[1], &event[3], ^{ compact(p_input_data, mask, p_even_blk_count, p_odd_blk_count, p_output_data); });
        mask = mask << 1;
    }
    release_event(event[3]);
}
```

Matrix Multiplication

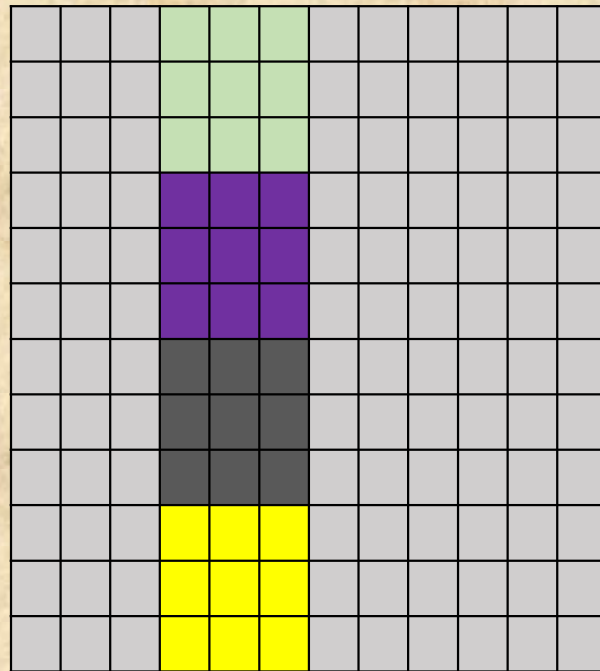



```
kernel void multiply(global float* pMatC, global const float* pMatA, global const float* pMatB, int M, int N, int P)
{
    int m = get_global_id(0);
    int p = get_global_id(1);
    float result = 0.0;
    for (int n = 0; n < N; n++)
    {
        result += (pMatA[(m*N)+n]*pMatB[(n*P)+p]);
    }
    pMatC[(P*m)+p] = result;
}
```

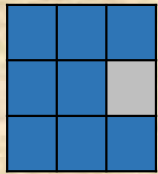
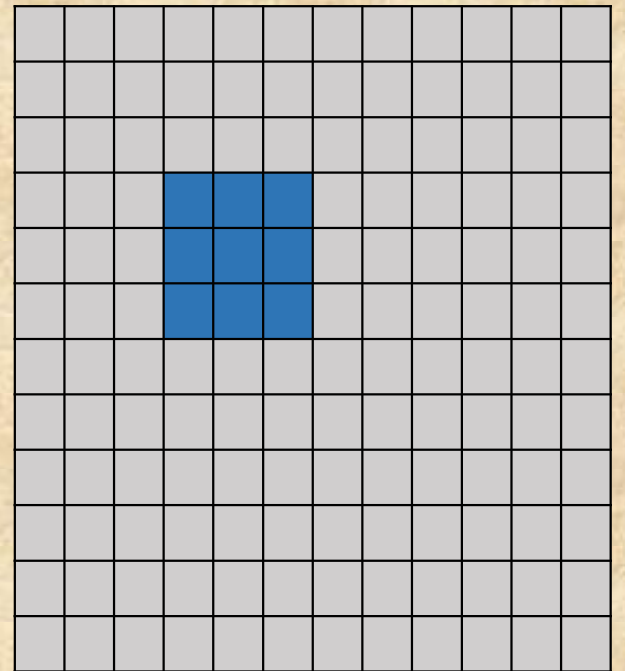
- Too much global memory access
- Is it possible to reduce the global memory acces?



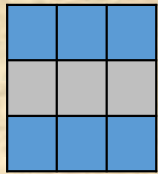
*



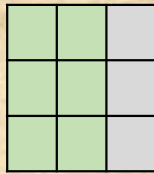
=



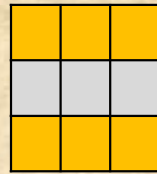
=



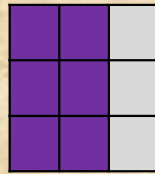
*



+



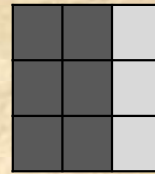
*



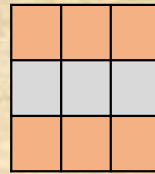
+



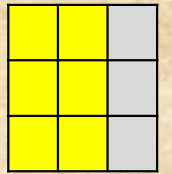
*



+



*



Coding Example 12: Matrix Multiplication

```
kernel void multiply(global float* pMatC, global const float* pMatA, global const float* pMatB, int M, int N, int P)
{
    local float shA[16][16];
    local float shB[16][16];

    int m = get_global_id(0);
    int p = get_global_id(1);
    int pc = (get_group_id(1)<<4)+get_local_id(0);

    float result = 0.0;
    for (int n = get_local_id(1); n < N; n += 16)
    {
        shA[get_local_id(0)][get_local_id(1)] = pMatA[(N*m)+n];
        shB[get_local_id(0)][get_local_id(1)] = pMatB[(P*n)+pc];
        barrier(CLK_LOCAL_MEM_FENCE);

        for (int i = 0; i < 16; i++)
        {
            result += (shA[get_local_id(0)][i]*shB[get_local_id(1)][i]);
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    pMatC[(P*m)+p] = result;
}
```


Optimization tips

- Algorithm (Parallelization ?)
- Memory
 - Cached access
 - Use of local memory
 - Bandwidth
- Parallel patterns
- Inline assembly ???
 - Many different GPU architectures (Nvidia, AMD, Intel, ARM, Qualcomm, Imagination)
 - Instruction set open for Nvidia and AMD

Debugging

- Intel OpenCL SDK
 - Breakpoints (Remote)
 - Single stepping (Remote)
 - Profiling
- Nvidia Nsight
 - Profiling support for OpenCL
 - Breakpoints (Only for CUDA apps)
 - Single stepping (Only for CUDA apps)
- Visual Studio supports OpenCL CPU device debugging

CUDA vs OPENCL

CUDA	OpenCL
- Proprietary Framework (Only Nvidia GPUs)	+ Open (Available on most of the Platforms) <small>OpenCL 1.2 is supported by Intel, AMD, Nvidia GPUs</small>
+ Supports C++ (Templates are very useful)	- Supports only C (C++ support available in OpenCL 2.1)
+ CUDA preprocessor (Allows mixing of device and host code)	- No preprocessor
+ CUDA compiler optimizations much better	- Not as good as CUDA
+ Lots of optimized 3 rd party libraries available	- Not as good as CUDA
+ Excellent tools for debugging, profiling etc.	- Not good

Reference

- Books

- OpenCL Programming Guide [Aftab Munshi, Benedict R. Gaster etc]
- Structured Parallel Programming Patterns for Efficient Computation [Michael McCool, Arch D.Robinson, James Reinders]
- The CUDA Handbook [Nicholas Wilt]
- GPGPU Gems [Nvidia]

- Useful Libraries

- Boost compute
- clBLAS
- Intel clDNN (Neural Network inference)

Exercise

1. Compute and display RGB histogram for a real time camera feed?
 - a) with atomics
 - b) without atomics

Compare the performance of both options.

THANK YOU