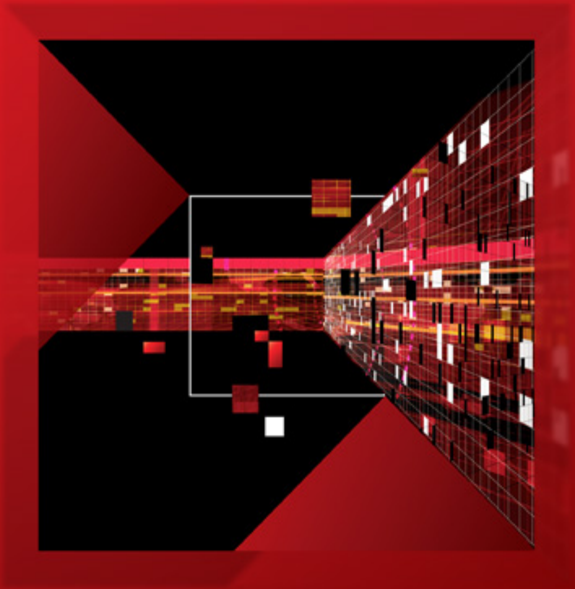


GPU COMPUTING RESEARCH WITH OPENCL

Studying Future Workloads and Devices

Perhaad Mistry, Dana Schaa, Enqiang Sun,
Rafael Ubal, Yash Ukidave, David Kaeli

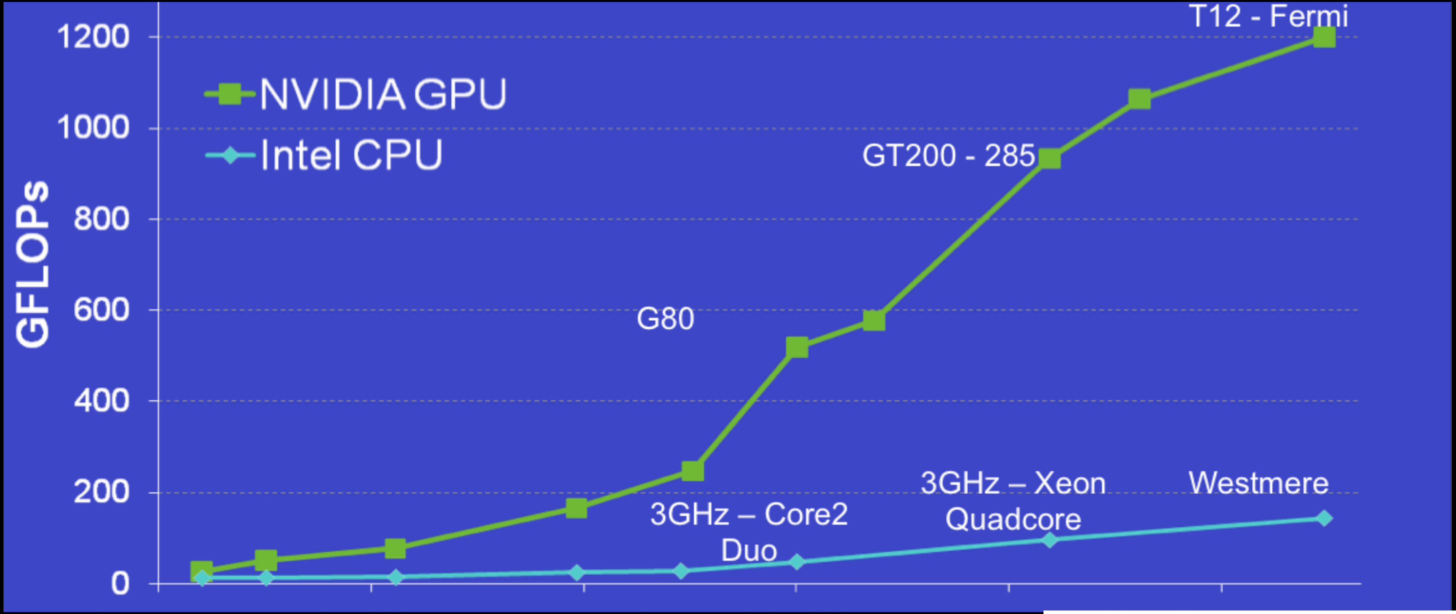
Dept of Electrical and Computer Engineering
Northeastern University

CCIS Class - CS 6240

TOPICS

- Introduction to OpenCL and GPU Computing
- Speeded Up Robust Features
- HAPTIC - OpenCL Heterogeneous Application Profiling & Introspection Capabilities

MOTIVATION TO STUDY GPU COMPUTING

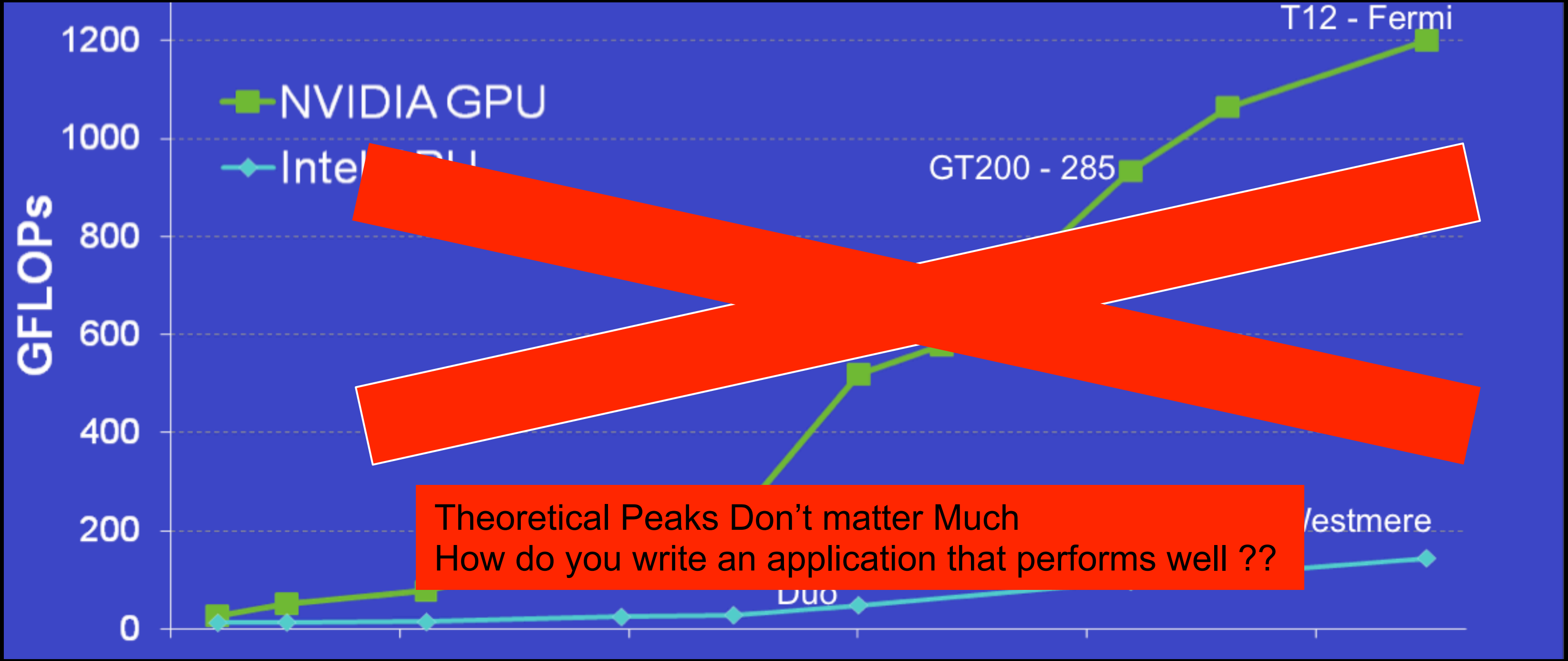


More than 65% of Americans played a video game in 2009 – economies of scale

Manufacturers include NVIDIA, AMD/ATI, IBM-Cell

Very competitive commodities market

MOTIVATION TO STUDY GPU COMPUTING

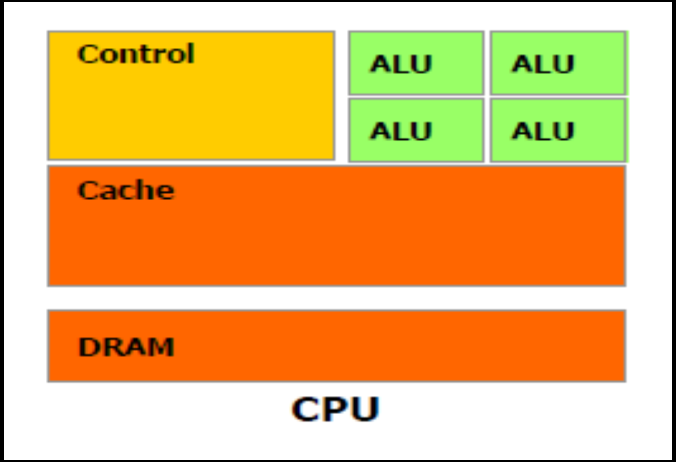


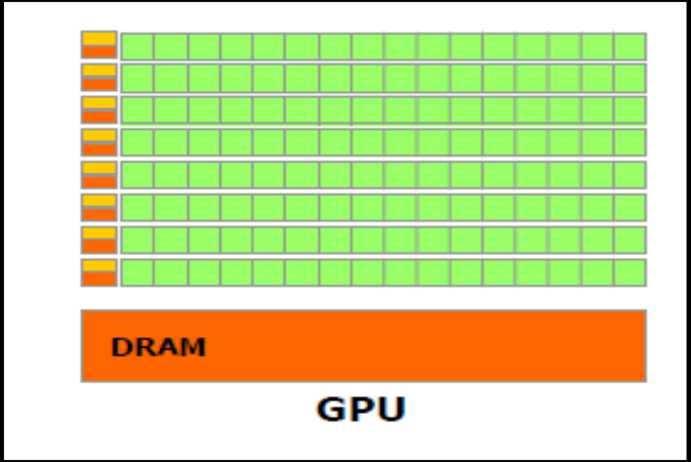
Theoretical Peaks Don't matter Much
How do you write an application that performs well ??

GPU COMPUTING - A wide range of GPU applications

- 3D image analysis
- Adaptive radiation therapy
- Acoustics
- Astronomy
- Audio
- Automobile vision
- Bioinfomatics
- Biological simulation
- Broadcast
- Cellular automata
- Fluid dynamics
- Computer vision
- Cryptography
- CT reconstruction
- Data mining
- Digital cinema / projections
- Electromagnetic simulation
- Equity training
- Film
- Financial
- Languages
- GIS
- Holographics cinema
- Machine learning
- Mathematics research
- Military
- Mine planning
- Molecular dynamics
- MRI reconstruction
- Multispectral imaging
- N-body simulation
- Network processing
- Neural network
- Oceanographic research
- Optical inspection
- Particle physics
- Protein folding
- Quantum chemistry
- Ray tracing
- Radar
- Reservoir simulation
- Robotic vision / AI
- Robotic surgery
- Satellite data analysis
- Seismic imaging
- Surgery simulation
- Surveillance
- Ultrasound
- Video conferencing
- Telescope
- Video
- Visualization
- Wireless
- X-Ray

CPU VS GPU ARCHITECTURES



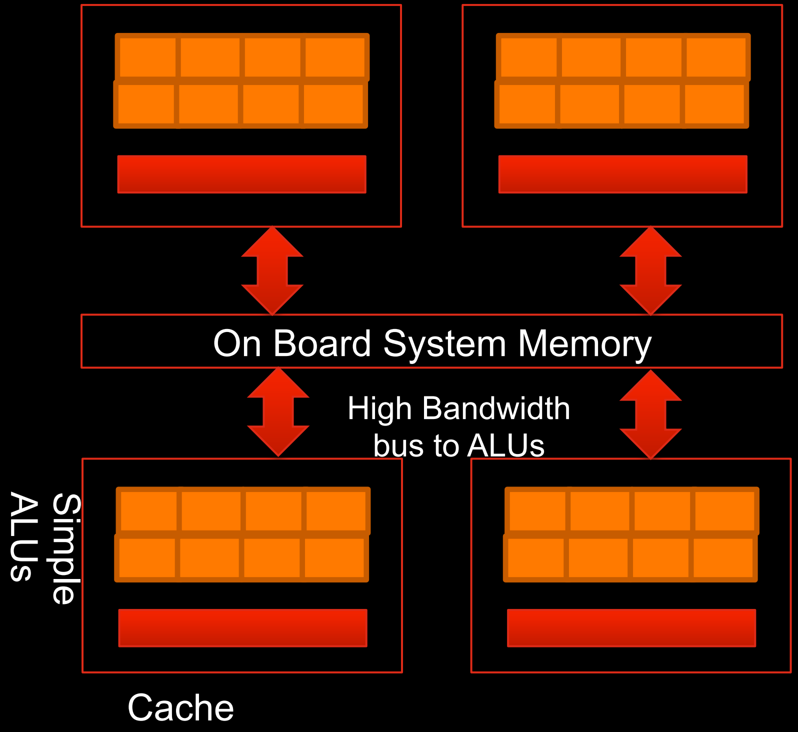


Irregular data accesses
Focus on per thread performance
Space devoted to control logic instead of ALU
Efficiently handle control flow intensive workloads
Multi level caches used to hide latency

Regular data accesses
More ALUs and massively parallel
Throughput oriented

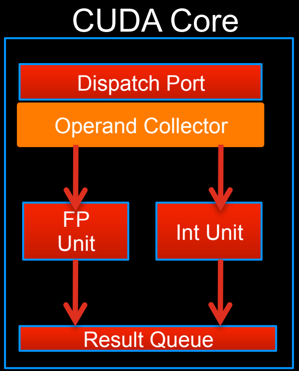
MODERN GPGPU ARCHITECTURE

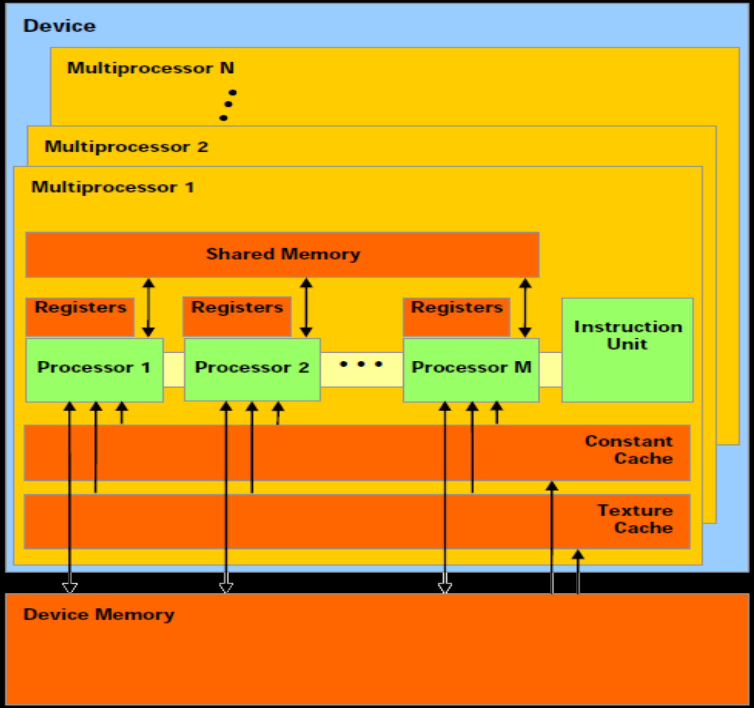
- Generic many core GPU
  - Less space devoted to control logic and caches
  - Large register files to support multiple thread contexts
- Low latency hardware managed thread switching
- Large number of ALU per “core” with small user managed cache per core
- Memory bus optimized for bandwidth
  - ~150 GBPS bandwidth allows us to service a large number of ALUs simultaneously



NVIDIA GPU COMPUTE ARCHITECTURE

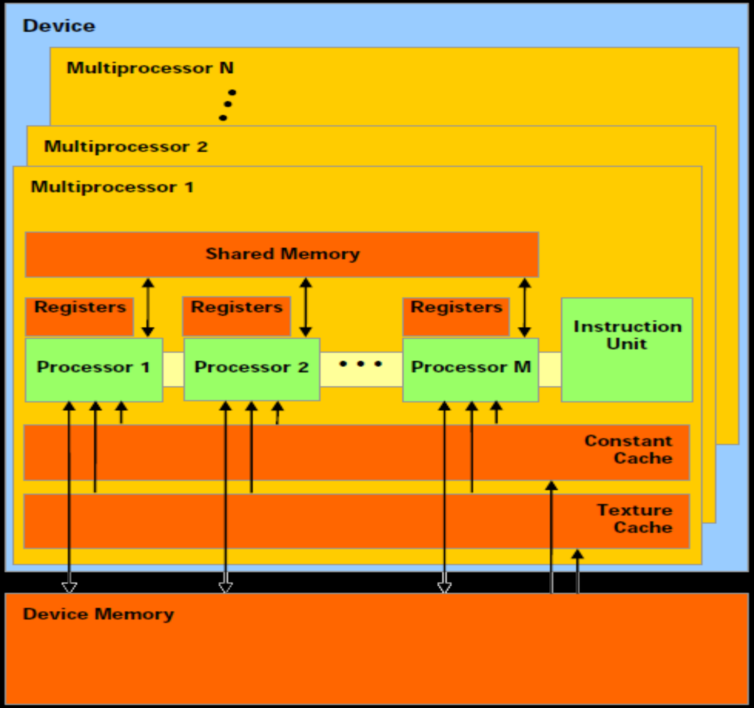
- Compute Unified Device Architecture
- Hierarchical architecture
- A device contains many multiprocessors
- Scalar “cuda cores” per multiprocessor
  - 32 for Fermi
- Single instruction issue unit per multiprocessor
- Many memory spaces
- GTX 480 - Compute 2.0 capability
  - 15 Streaming Multiprocessors (SMs)
  - 1 SM features 32 CUDA processors
  - 480 CUDA processors





GPU MEMORY ARCHITECTURE

- Device Memory (GDDR)
  - Large memory with a high bandwidth link to multiprocessor
- Registers on chip (~16k)
  - Large number of registers enable low overhead context switching and massive multithreading
- Shared memory (on chip)
  - Shared between scalar cores
  - Low latency and banked
- Constant and texture memory
  - Read only and cached



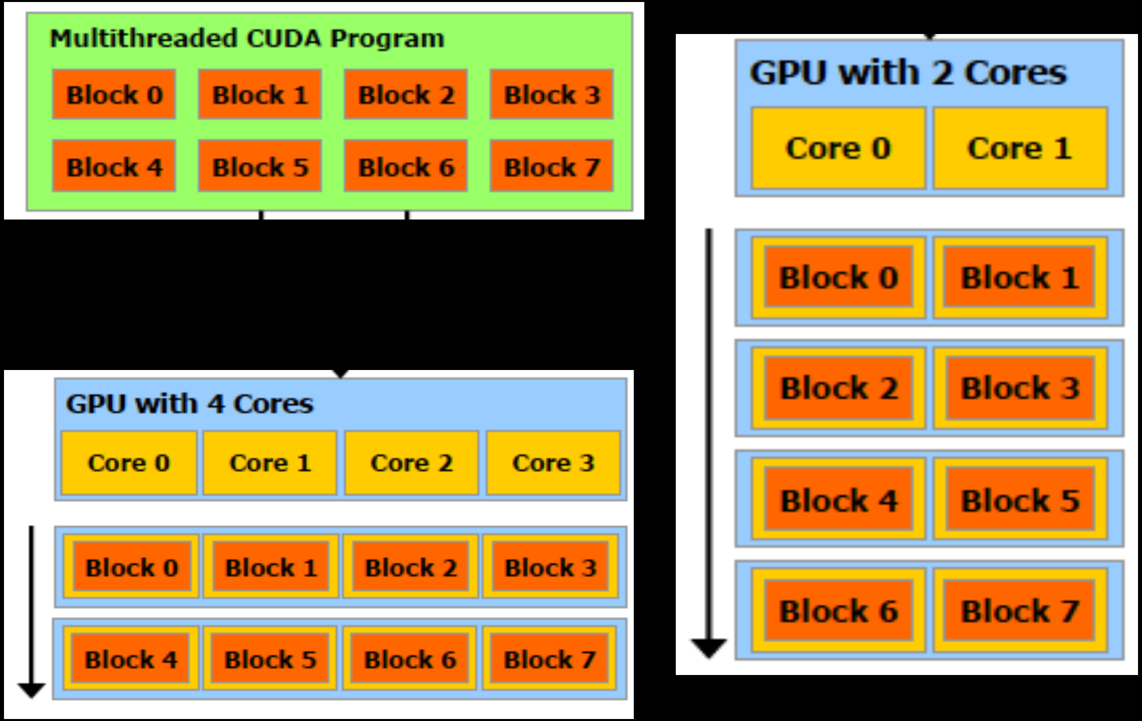
A “TRANSPARENTLY” SCALABLE ARCHITECTURE

The programming model maps easily to underlying architecture

Scalable programming model

Program consists of independent blocks of threads

Same program will be scalable across devices

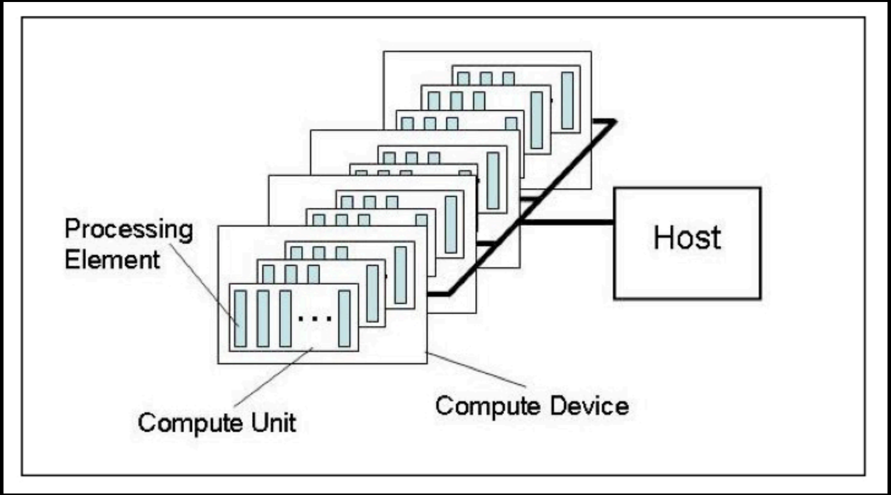


AN OPTIMAL GPGPU PROGRAM

- From the discussion on hardware we see that an ideal kernel for a GPU:
  - Has thousands of independent pieces of work
    - Uses all available compute units
    - Allows interleaving for latency hiding
  - Is amenable to instruction stream sharing
    - Maps to SIMD execution by preventing divergence between work items
  - Has high arithmetic intensity
    - Ratio of math operations to memory access is high
    - Not limited by memory bandwidth
- Note that these caveats apply to all GPUs

OPENCL – THE FUTURE FOR MANY-CORE COMPUTING

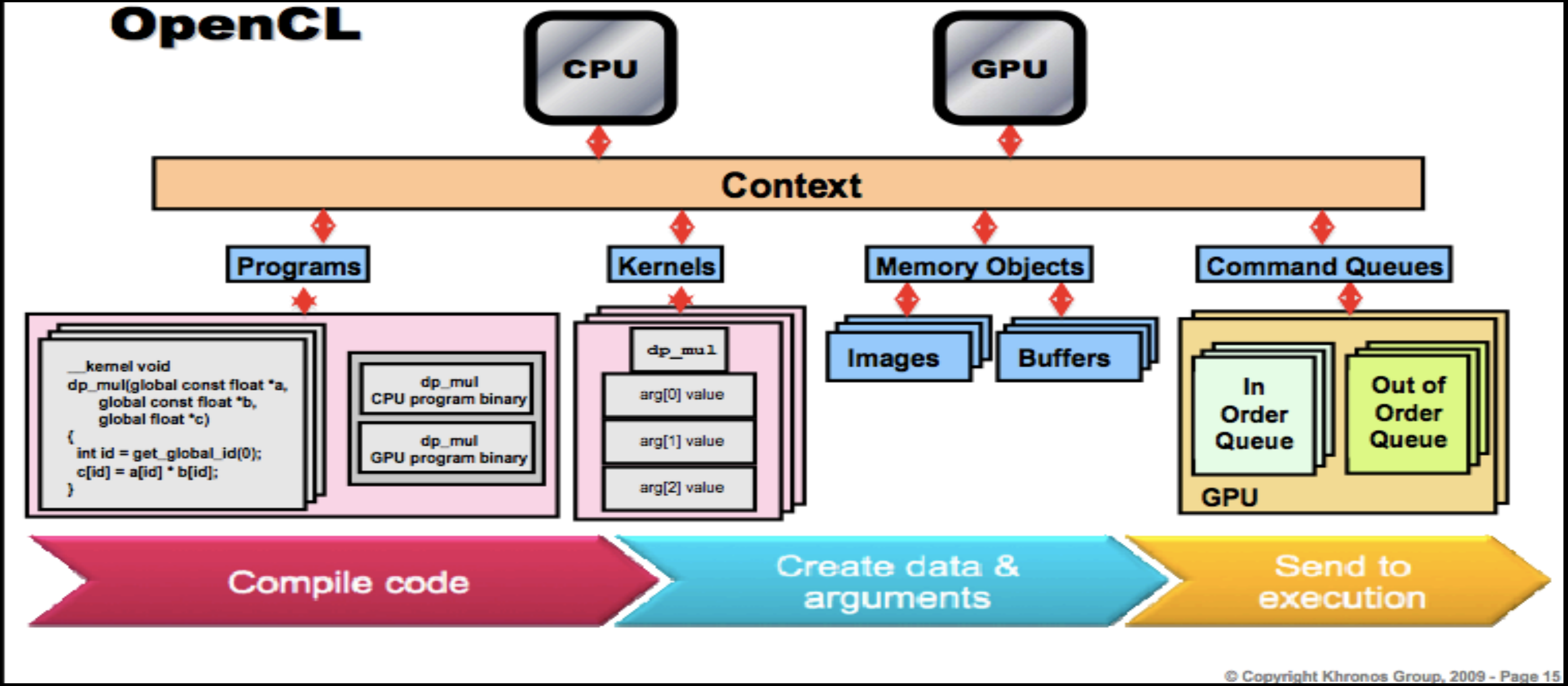
- OpenCL (Open Computing Language) released in 2008
  - Developed by Khronos Group – a non-profit
- A framework similar to CUDA for writing programs that execute on heterogeneous systems
- Allows CPU and GPU to work together for faster and more efficient processing
- Modeled as four parts:
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model
- Kernels — execute on heterogeneous devices
  - Same kernel on multiple devices such as CPUs, GPUs, DSPs, FPGAs, etc



OPENCL – CONFORMANT COMPANIES

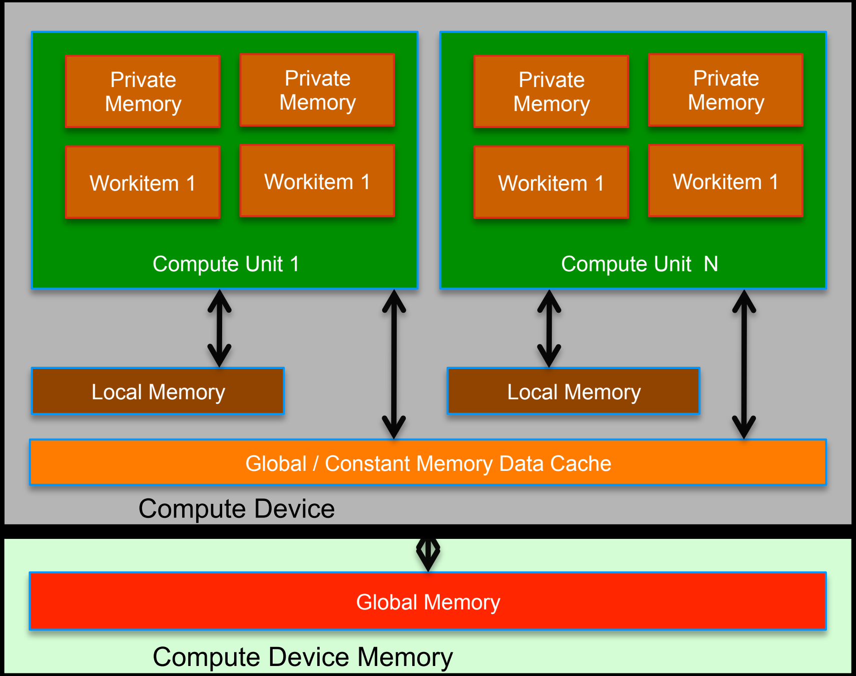
Over 300+ OpenCL 1.1 Compliant Devices
Altera, TI coming up…
OpenCL 1.2 announced at SC 11

OPENCL - THE BIG PICTURE



```
__kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

GPU MEMORY MODEL IN OPENCL



- For both AMD, Nvidia GPUs a subset of hardware memory exposed in OpenCL
- Configurable shared memory is usable as local memory
  - Local memory used to share data between items of a work group at lower latency than global memory
- Private memory utilizes registers per work item

OPENCL EXAMPLE - BASIC MATRIX MULTIPLICATION

- Non-blocking matrix multiplication
  - Doesn't use local memory
    - Each element of matrix reads its own data independently
- Serial matrix multiplication

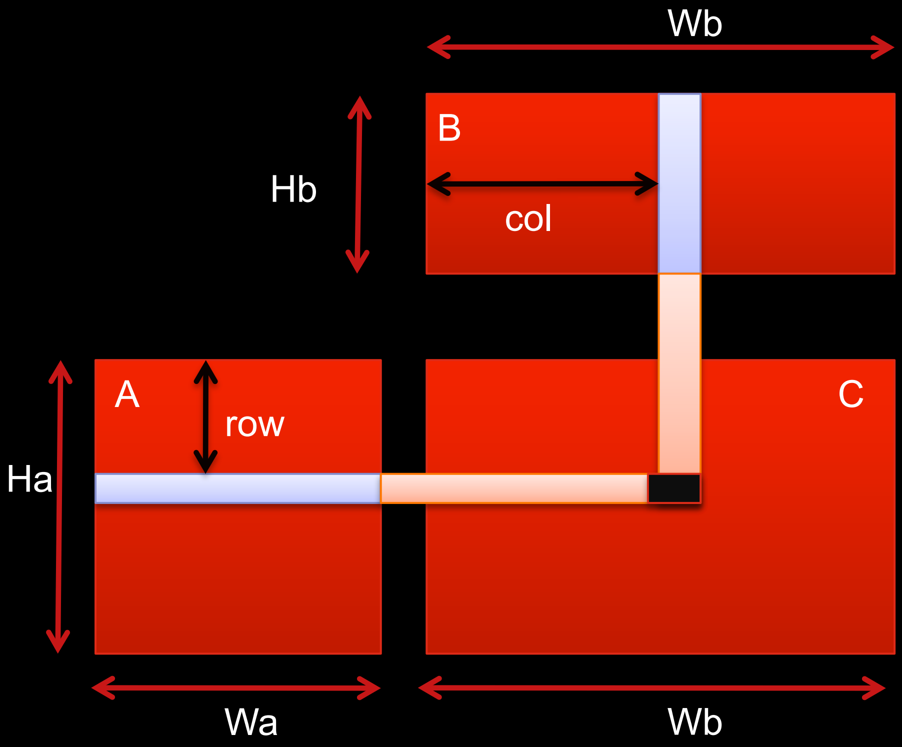
```
for(int i = 0; i < Ha; i++)
        for(int j = 0; j < Wb; j++){
                c[i][j] = 0;
                for(int k = 0; k < Wa; k++)
                        c[i][j] +=  a[i][k] + b[k][j]
        }
```

- Reuse code from image rotation
  - Create context, command queues and compile program
  - Only need one more input memory object for 2nd matrix

SIMPLE MATRIX MULTIPLICATION

```
__kernel void simpleMultiply(
__global float* c,
int Wa, int Wb,
__global float* a, __global float* b)
{

    //Get global position in Y direction
    int row = get_global_id(1);
    //Get global position in X direction
    int col   = get_global_id(0);
    float sum = 0.0f;
    //Calculate result of one element
    for (int i = 0; i < Wa; i++)  {
        sum += a[row*Wa+i] * b[i*Wb+col];
    }
    c[row*Wb+col] = sum;
}
```



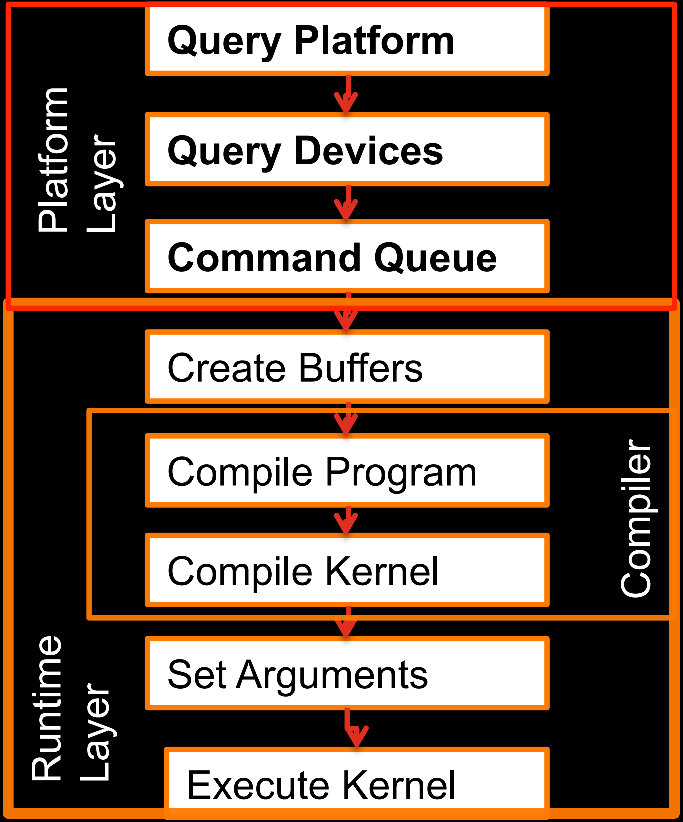
STEP0: INITIALIZE DEVICE

- Declare context
- Choose a device from context
- Using device and context create a command queue

```
cl_context myctx = clCreateContextFromType (
                 0, CL_DEVICE_TYPE_GPU,
                 NULL, NULL, &ciErrNum);
```

```
ciErrNum = clGetDeviceIDs (0,
                 CL_DEVICE_TYPE_GPU,
                 1, &device,  cl_uint *num_devices)
```

```
cl_commandqueue myqueue ;
myqueue = clCreateCommandQueue(
                 myctx, device, 0, &ciErrNum);
```



STEP1: CREATE BUFFERS

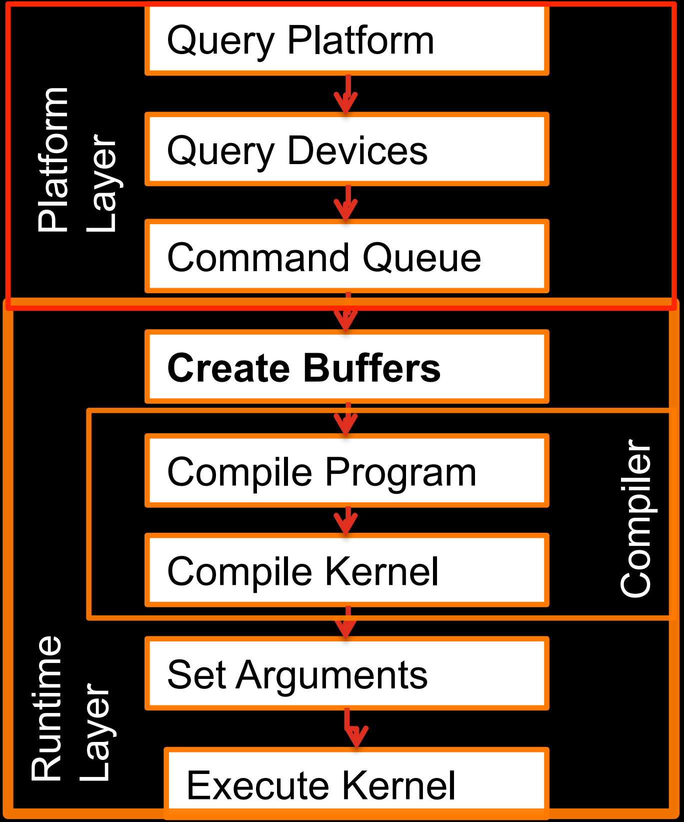
- Create buffers on device
- Input data is read-only
- Output data is write-only

```
cl_mem d_a = clCreateBuffer( myctx,
                CL_MEM_READ_ONLY, mem_size,
                NULL, &ciErrNum);
```

```
cl_mem d_c = clCreateBuffer( myctx,
                CL_MEM_WRITE_ONLY, mem_size,
                NULL, &ciErrNum);
```

- Transfer input data to the device

```
ciErrNum = clEnqueueWriteBuffer (
                myqueue , d_a, CL_TRUE, 0, mem_size,
                (void *)src_image, 0, NULL,  NULL)
```

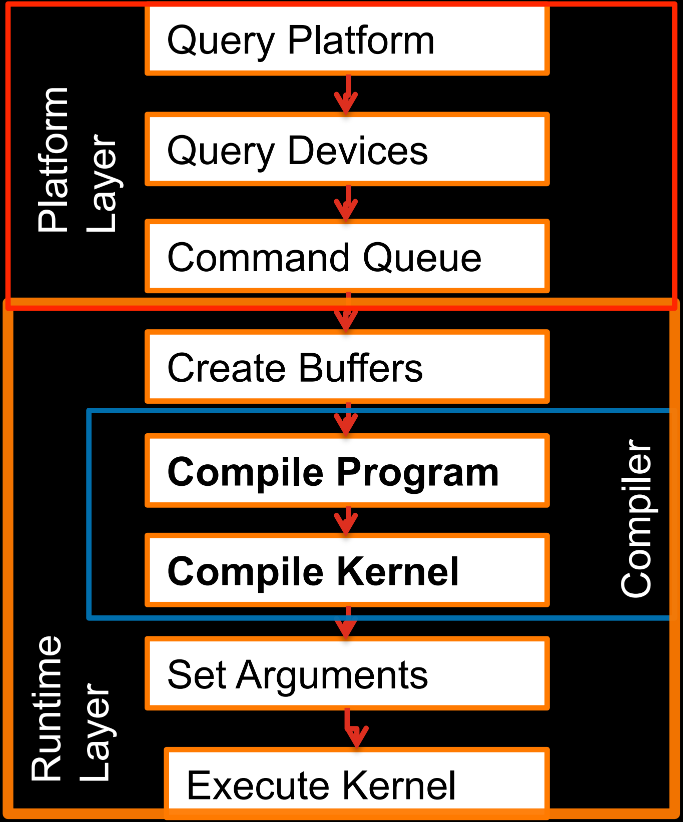


STEP2: BUILD PROGRAM, SELECT KERNEL

```
// create the program
cl_program myprog = clCreateProgramWithSource
              ( myctx,1, (const char **)&source,
              &program_length, &ciErrNum);
```

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

```
//Use the “image_rotate” function as the kernel
cl_kernel mykernel = clCreateKernel (
              myprog , “image_rotate” ,  error_code)
```

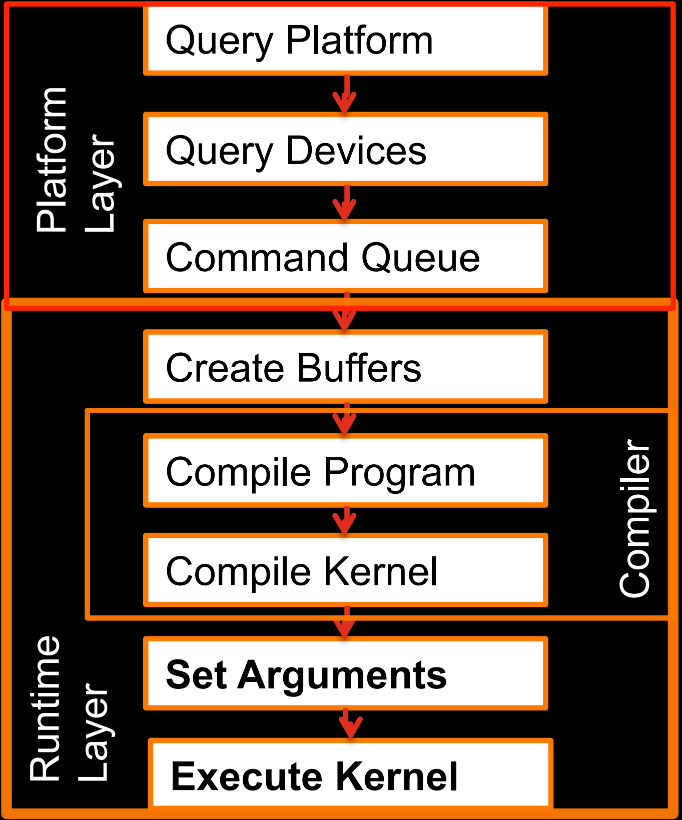


STEP3: SET ARGUMENTS, ENQUEUE KERNEL

```
// Set Arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem), (void *)&d_a);
clSetKernelArg(mykernel, 1, sizeof(cl_mem),  (void *)&d_b);
clSetKernelArg(mykernel, 2, sizeof(cl_int),  (void *)&W);
…
```

```
//Set local and global workgroup sizes
size_t localws[2] = {16,16} ;
size_t globalws[2] = {W, H};//Assume divisible by 16
```

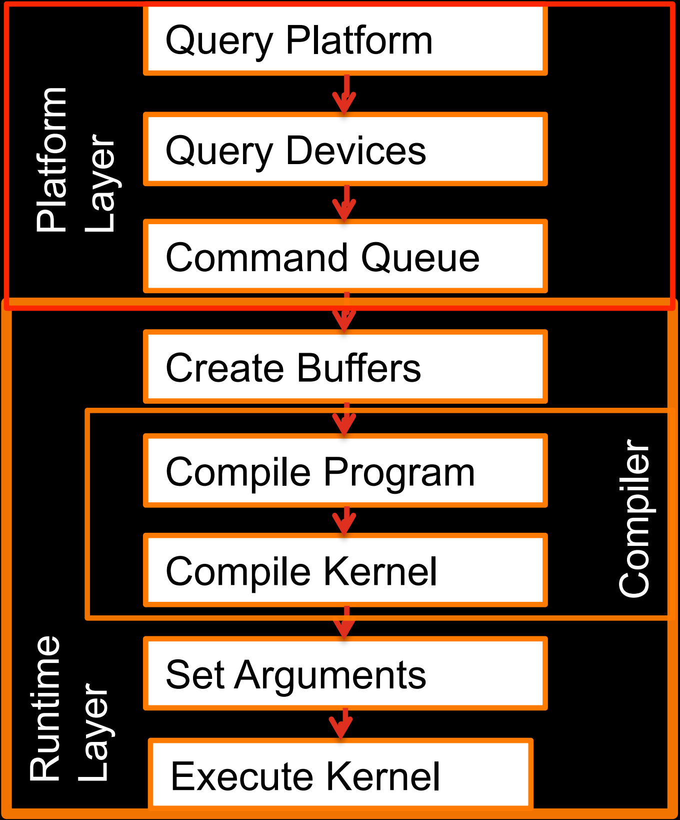
```
// execute kernel
clEnqueueNDRangeKernel(
                  myqueue , myKernel, 2, 0, globalws, localws,
                0, NULL, NULL);
```



STEP4: READ BACK RESULT

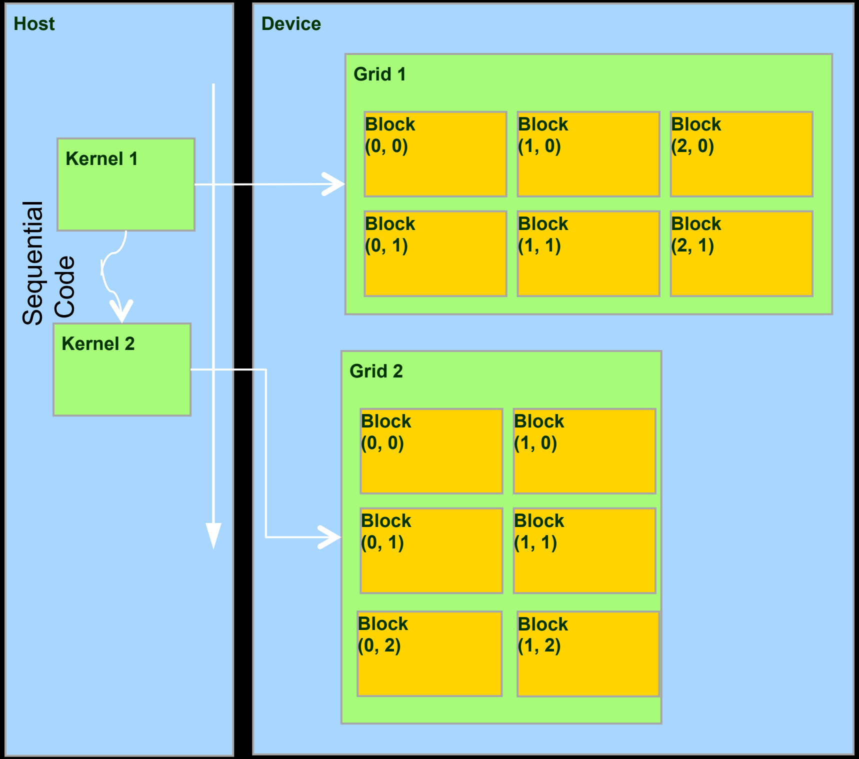
- Only necessary for data required on the host
- Data output from one kernel can be reused for another kernel
  - Avoid redundant host-device IO

```
// copy results from device back to host
clEnqueueReadBuffer(
                myctx, d_op,
                CL_TRUE,                //Blocking Read Back
                0, mem_size,  (void *) op_data,
                NULL, NULL, NULL);
```



SUMMARY - STEPS PORTING TO OPENCL

- Create standalone C / C++ version
- Multi-threaded CPU version (debugging, partitioning)
- Simple OpenCL version
- Optimize OpenCL version for underlying hardware
- No reason why an application should have only 1 kernel
- Use the right processor for the job



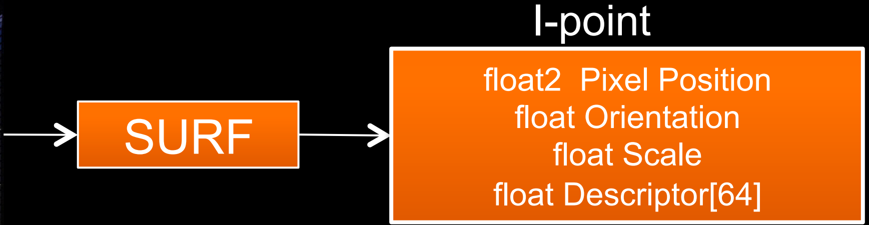
SPEEDED UP ROBUST FEATURES

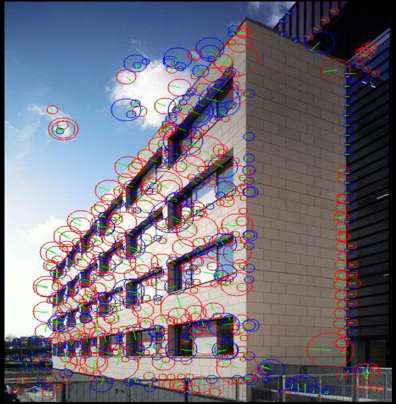
Computer Vision Applications

Perhaad Mistry, Dana Schaa, Enqiang Sun, David Kaeli
Northeastern University

SPEEDED UP ROBUST FEATURES (SURF)

- "Summarize" an image into a number of "interest points"
  - Robust features - Simple to compute, small
  - More insensitive to changes in image like scale, rotation
  - Open source – Highly optimized
- Applications: Object recognition, tracking , image stitching etc
- http://code.google.com/p/clsurf/

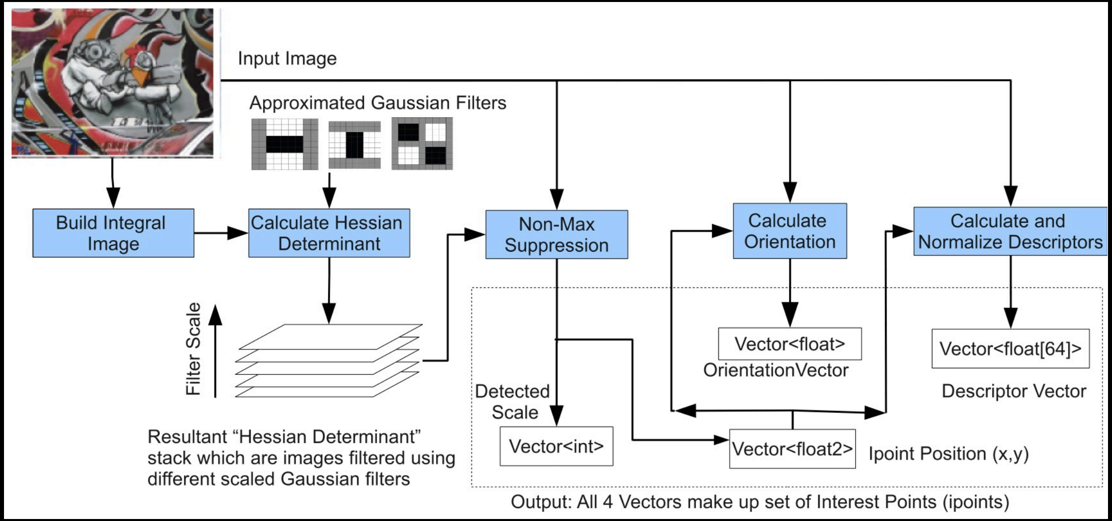




Speeded-Up Robust Features (SURF), Herbert Bay et. al.

SPEEDED UP ROBUST FEATURES (SURF)

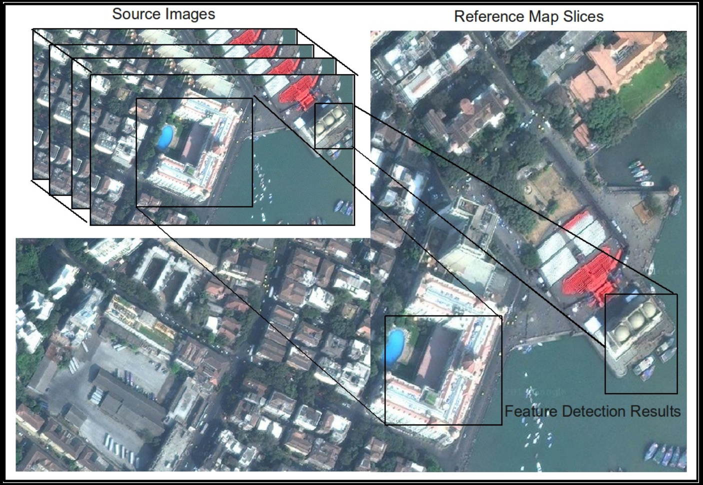
- Integral image: (2 kernels) 4 calls
  - Scan, transpose in 2 dimensions
- Hessian: (2 Kernels) 8 calls
  - Groups of convolutions
- Non max suppression: (1 kernel) 5 calls
  - Maxima and minima from convolution
- Orientation: (2 kernels) 2 calls
  - Local intensity gradients for rotation invariance
- Descriptors: (2 kernels) 2 calls
  - Haar descriptors around each i-point

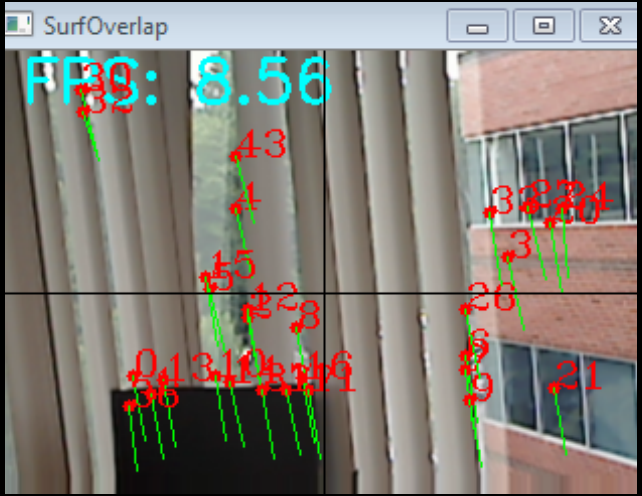

Input Image
Approximated Gaussian Filters
Build Integral Image
Calculate Hessian Determinant
Non-Max Suppression
Calculate Orientation
Calculate and Normalize Descriptors
Filter Scale
Resultant "Hessian Determinant" stack which are images filtered using different scaled Gaussian filters
Detected Scale
Vector<int>
Vector<float>
OrientationVector
Vector<float[64]>
Descriptor Vector
Vector<float2>
Ipoint Position (x,y)
Output: All 4 Vectors make up set of Interest Points (ipoints)

SURF is a multi-kernel pipeline where each stage contributes a part of each feature

SURF APPLICATIONS

- Applications using SURF's generated features
- Image Search - Compare descriptors of different features using simple Euclidean distance
- Video Stabilization - Compare orientation values of different features





PERFORMANCE CHARACTERISTICS OF SURF

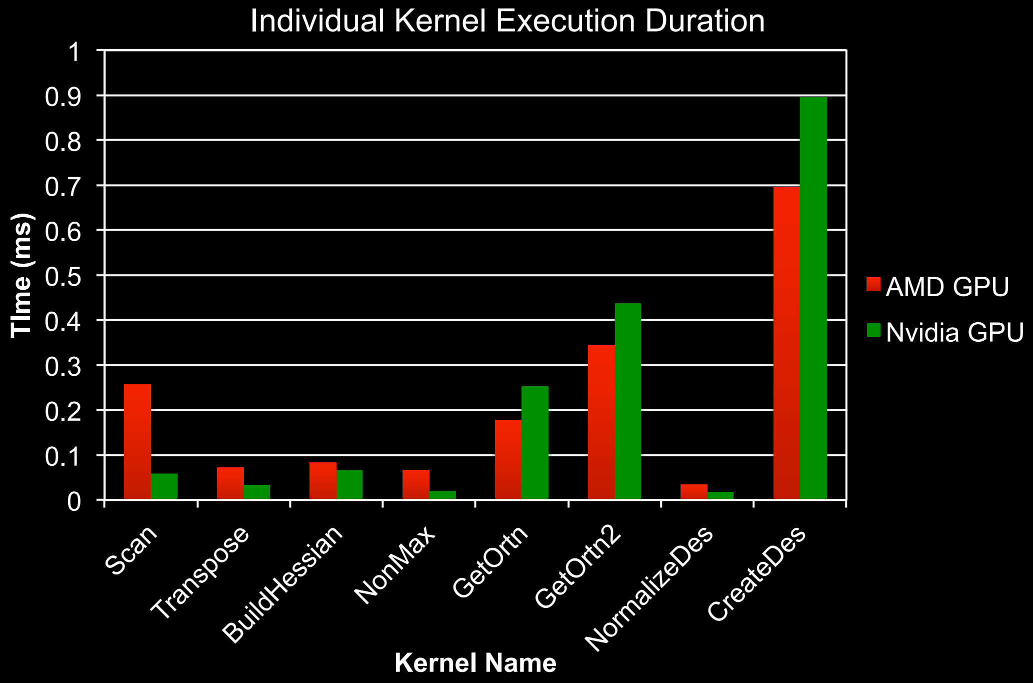
- Performance is hard to predict because of variable feature counts
  - Feature count decides the workgroup sizes down the pipeline

- We aim to study SURF’s performance when embedded into applications
  - Not always as clean as embedding a spmv kernel in a solver

- Many OpenCL kernels of varying complexity
  - 10 kernels varying from 5 lines to 280 lines
  - Kernels called multiple times
    - Number of kernel calls unknown until runtime

INDIVIDUAL KERNEL PERFORMANCE

- Optimization steps for kernels
  - Timing of each kernel across frames
- Events show a consistent view across devices
  - Individual timings are not representative
  - Createdescriptors is longest kernel
  - However BuildHessian is called more
  - Hard to find without profiling
- Reducing the number of kernel calls may be as beneficial as applying platform specific optimization
- Profiling allows us to pursue feedback-driven optimization



WHY ARE WE TALKING ABOUT SURF ?

- Especially since we haven’t seen any OpenCL kernels or host code
- Performance Characteristics of SURF
  - Data driven performance necessitates profiling at runtime
  - Input arguments threshold determine performance
- Commonly used as a algorithm kernel within an application
  - Applications include stabilization of a video, image searching, motion tracking, etc.
  - Same algorithm used for different applications with different input parameters
    - Number of convolutions
    - Thresholds
- Improve the state of the art in performance analysis tools for interesting workloads
  - Improve performance for complex and irregular applications and algorithms

HAPTIC

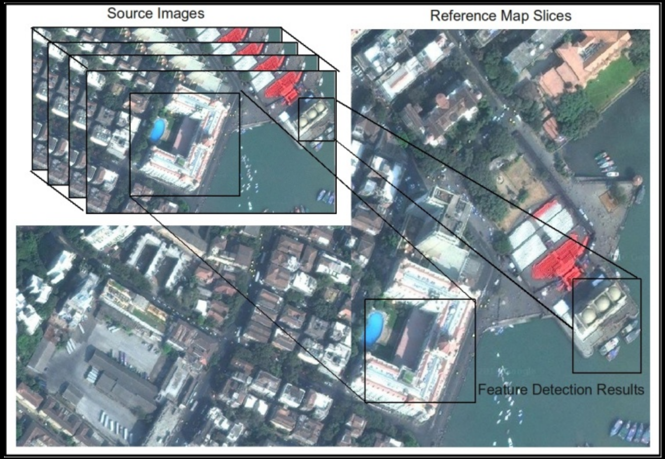
OpenCL Heterogeneous Application Profiling & Introspection Capabilities

Perhaad Mistry, David Kaeli

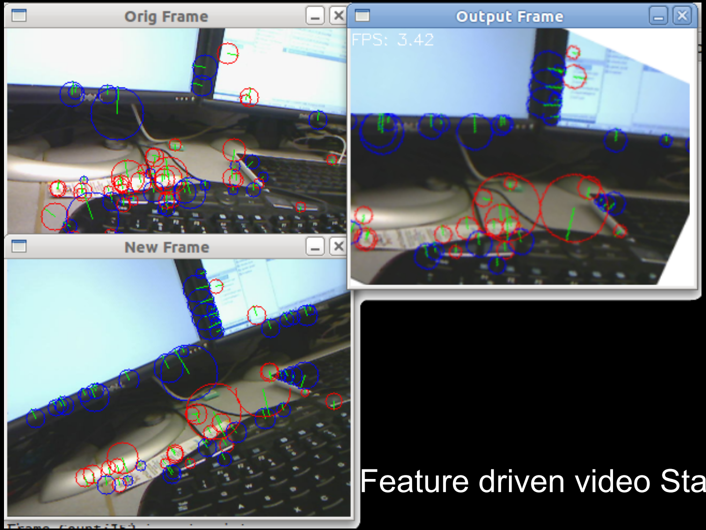
Department of Electrical and Computer Engineering
Northeastern University

MOTIVATION FOR PROFILING CAPABILITIES WITHIN A HETEROGENEOUS APPLICATION

- Library developer cannot predict all applications where his/her library will be used
- Algorithms whose performance is dependent on factors other than “data size”
  - Analysis is required at runtime by the library to learn about the application



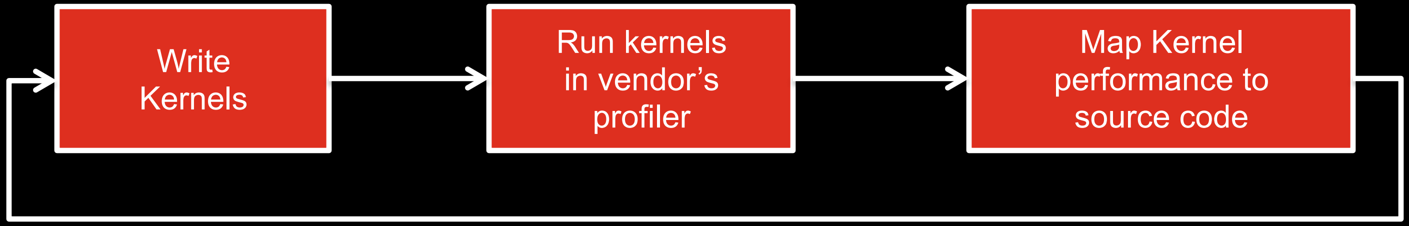
Feature Based Image Search



Feature driven video Stabilization

PERFORMANCE OPTIMIZATION STEPS IN OPENCL TODAY

- A continuous process, restricted to development stage for OpenCL / CUDA
- Kernel writer needs to know about how his kernel will be used which leads to over-conservative assumptions while coding
  - Types of algorithms where you don’t know OP characteristics
  - Decides format and location of OP data structures
  - Simple example bucket sort, where each bucket has to be a big size and the number of buckets
  - Data driven performance problems are hard to catch
- Once the kernel is written, no framework exists that monitors performance of the kernel


Write Kernels
Run kernels in vendor’s profiler
Map Kernel performance to source code

Repeat till you grow old / change project

OPENCL EVENTS

- OpenCL provides not only cross platform applications, but also mechanisms to create tools for parallel computing
- ***Events*** are an interface to understanding OpenCL performance
  - Event objects (`cl_event`) used to determine command status
- OpenCL enqueue methods return event objects
  - Provides for command level control and synchronization

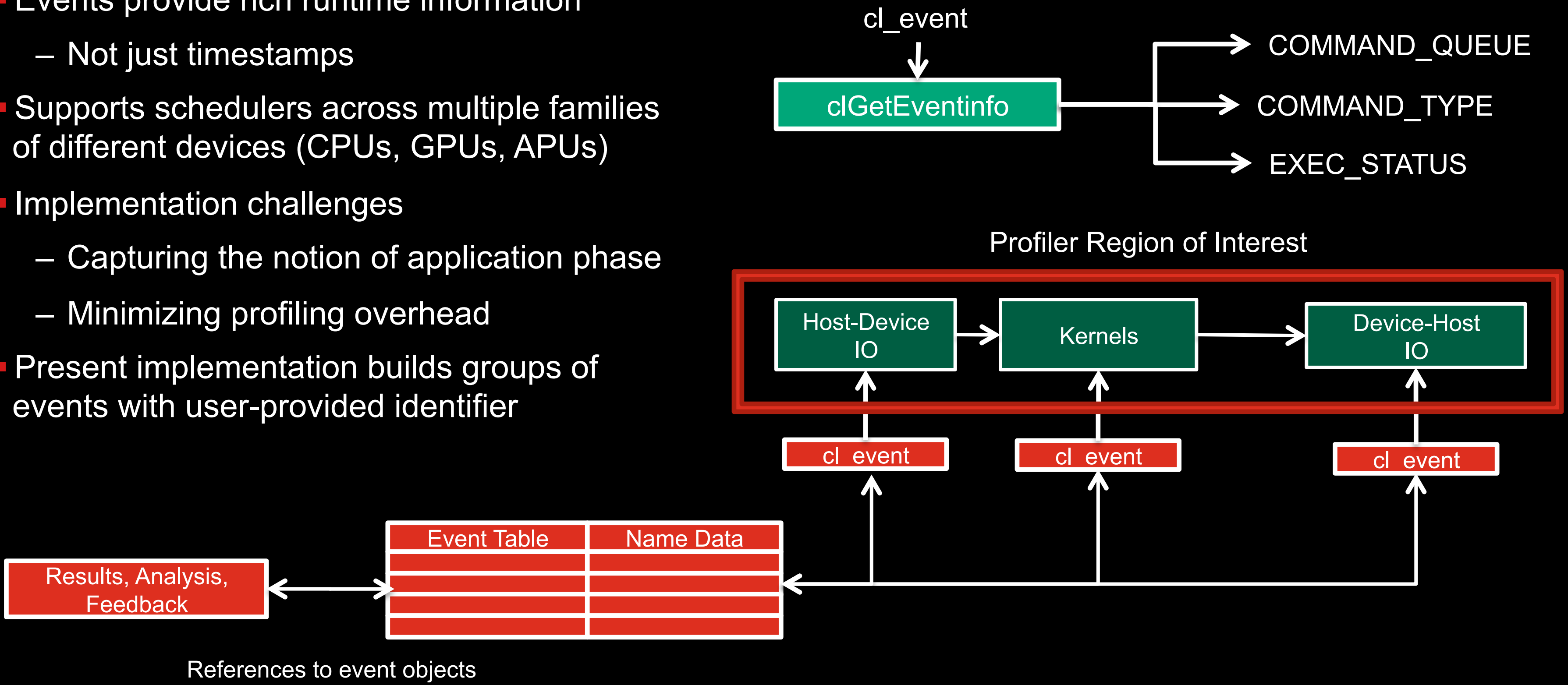
Command State	Description
CL_QUEUED	Command is in a queue
CL_SUBMITTED	Command has been submitted to device
CL_RUNNING	Command is currently executing on device
CL_COMPLETE	Command has finished execution

Command states as visible from OpenCL events

```
cl_int clEnqueueNDRangeKernel (
                cl_command_queue queue,
                cl_kernel kernel, cl_uint work_dim,
                const size_t *global_work_offset,
                const size_t *global_work_size,
                const size_t *local_work_size,
                cl_uint num_events_in_wait_list,
                const cl_event *event_wait_list,
                cl_event *event)
```

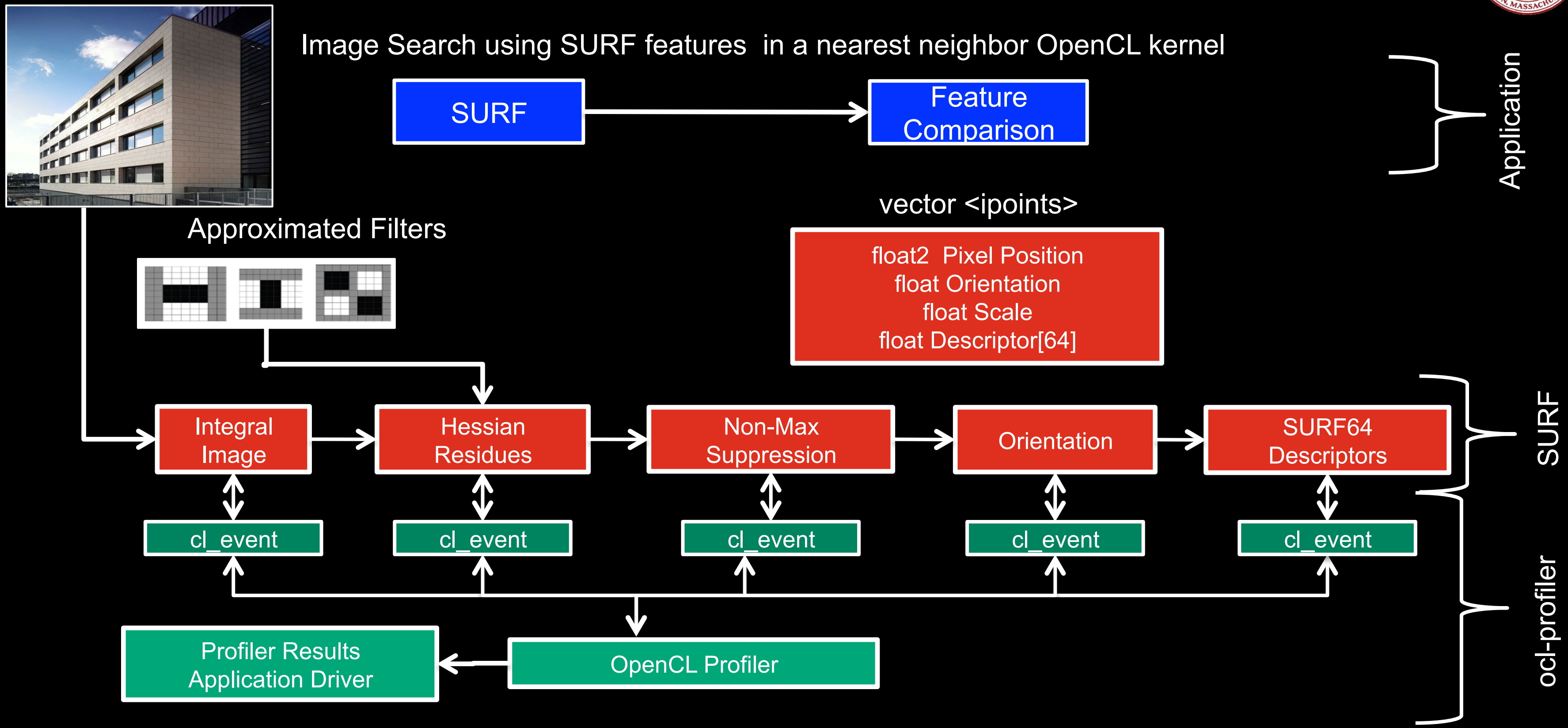
OPENCL PROFILING

- Events provide rich runtime information
  - Not just timestamps
- Supports schedulers across multiple families of different devices (CPUs, GPUs, APUs)
- Implementation challenges
  - Capturing the notion of application phase
  - Minimizing profiling overhead
- Present implementation builds groups of events with user-provided identifier



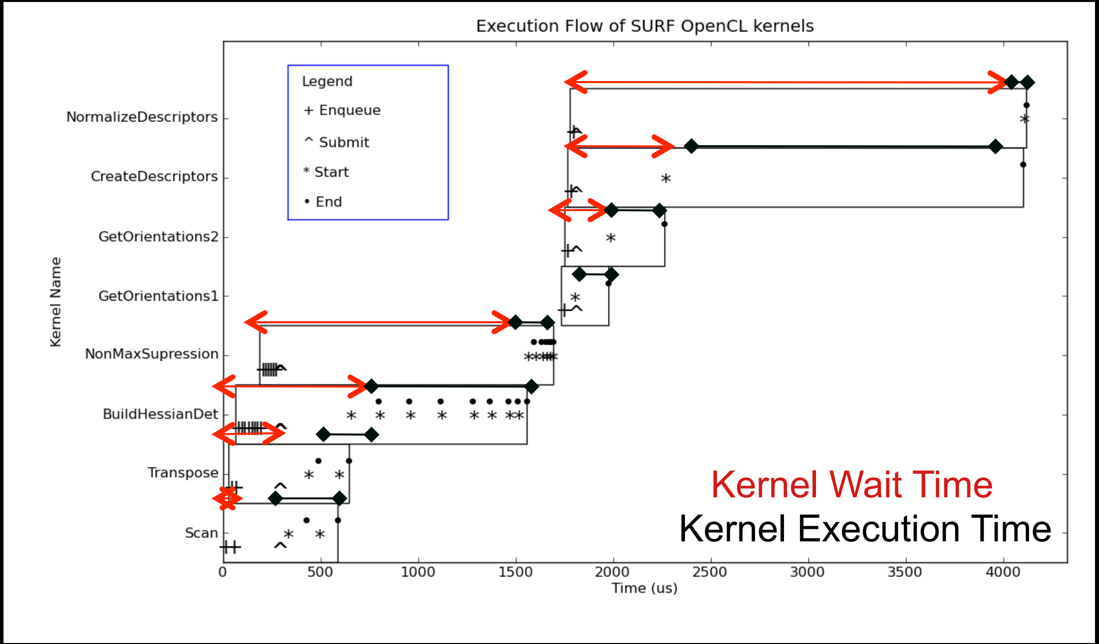
OPENCL PROFILER IN SURF APPLICATION

Image Search using SURF features in a nearest neighbor OpenCL kernel



KERNEL TIMELINE IN SURF

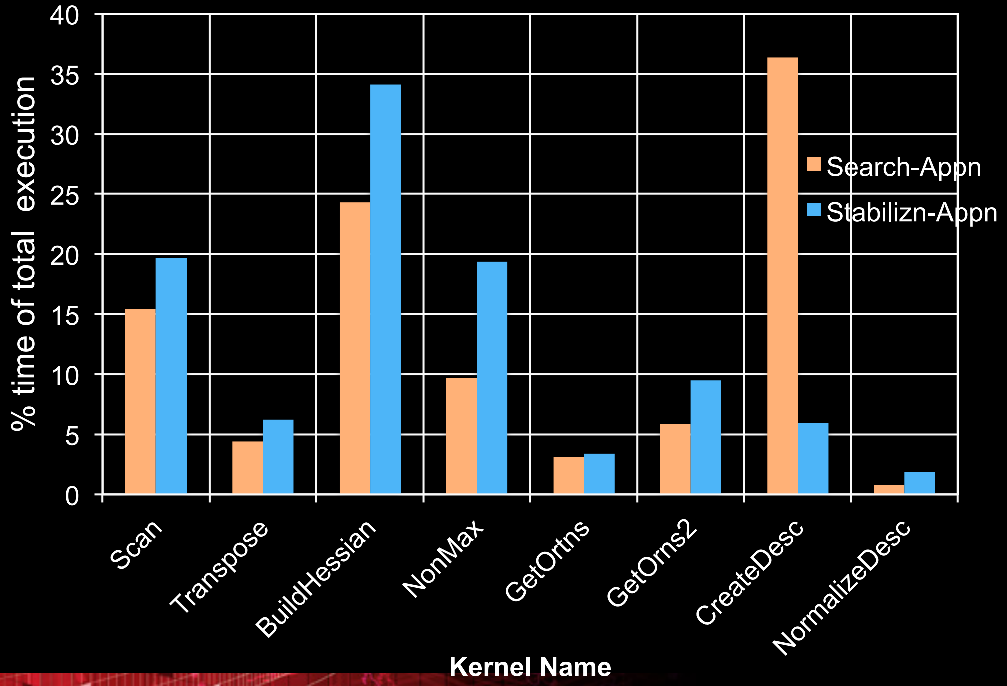
- Application view of SURF
  - Kernel pipelined over data set
  - Averaged event time stamps for a data set
- Exposes optimization opportunities
  - Cumulative time of small kernel
  - High kernel call count
  - Device – host IO duration is insignificant in pipeline
- Used to estimate host idle time once kernels are enqueued



SURF PERFORMANCE FOR DIFFERENT APPLICATIONS

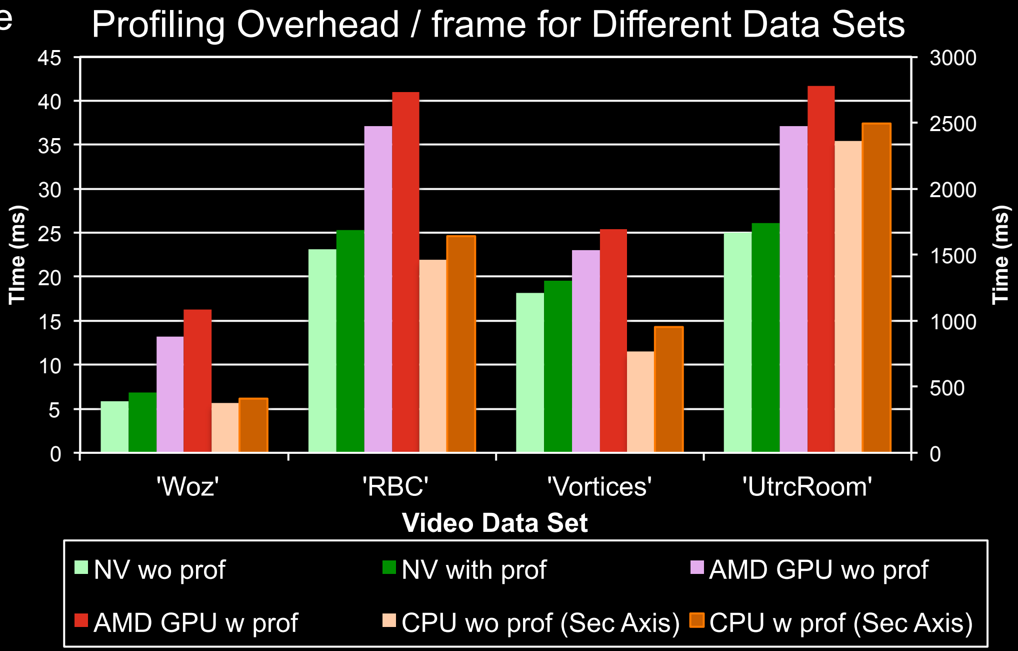
- Different applications on top of SURF
  - Stabilization
  - Image Search
- Search Application:
  - Create-Descriptor is the bottleneck
  - **Split kernel on multiple devices**
- Stabilization Application:
  - Build-Hessian is the bottleneck
  - Reduce the number of kernel calls

Percentage time of each kernel of SURF (AMD 5870)



PROFILER OVERHEAD

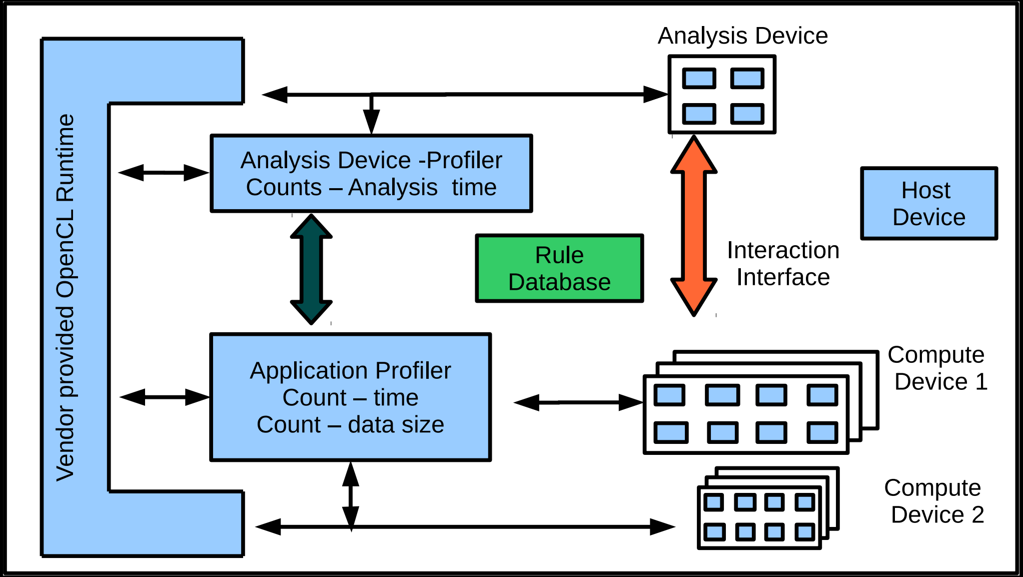
- Baseline: profiling disabled in command queue
  - Overhead for different videos
- Simple techniques to minimize overhead
  - Grow event list once and reuse data structures
- Query events after frame
  - Allows for variable granularity of performance measurement
- We show the worst case overhead for SURF
  - Profiling all kernels for every frame



Consistent overhead seen - per platform

ANALYSIS DEVICES - PRELIMINARY WORK

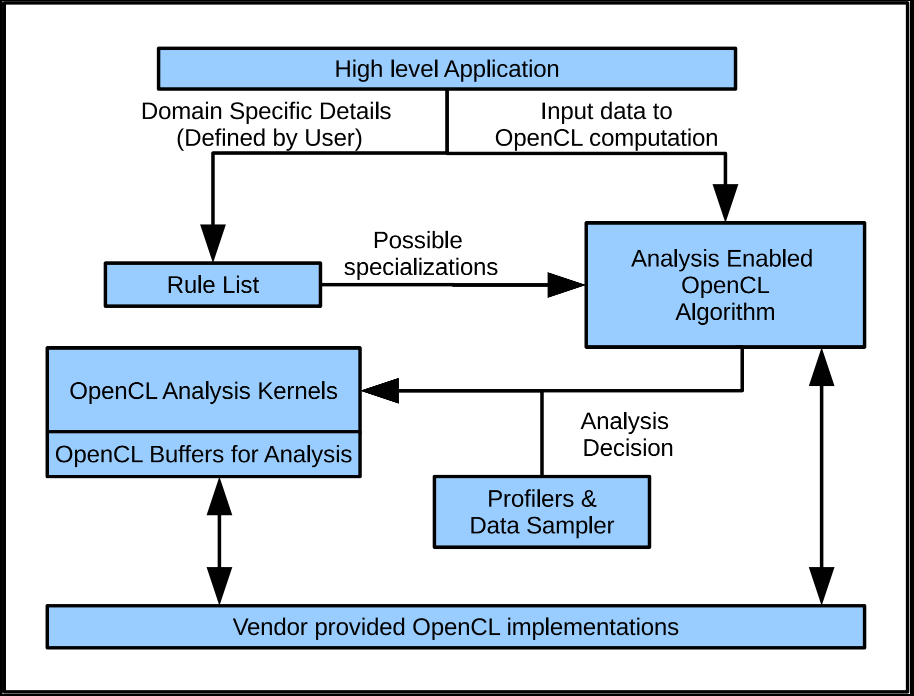
- Motivated by the fact that the GPU is rapidly disappearing under libraries and frameworks
  - A core library writer doesn't know each high level application
- Specialization of an underlying OpenCL system based on domain specific information
  - A specialized compute device known as a "Analysis Device"
- Exploit extra OpenCL devices to work on computation that can help performance
  - Preprocessing passes
  - Data transformation
  - Data value monitoring



The system consists OpenCL profilers (discussed previously) which monitor application performance on the compute device
Present granularity limited to on a OpenCL kernel basis

ANALYSIS DEVICES – PROGRAMMING VIEW

- Test applications developed use SURF as a example underlying computational kernel pipeline whose behavior is configurable
- Rules are prewritten OpenCL kernels whose execution could improve the application
- Example Specializations – for SURF
  - Turn ON / OFF pipeline stages
  - Change frequency of SURF calls for invariant data
  - Change thresholds of SURF which changes the number of features
- Can be used to hide access to source code and deep architectural optimization details
  - While providing knobs to specialize a computational pipeline to an application


High level Application
Domain Specific Details (Defined by User)
Input data to OpenCL computation
Rule List
Possible specializations
Analysis Enabled OpenCL Algorithm
OpenCL Analysis Kernels
OpenCL Buffers for Analysis
Analysis Decision
Profilers & Data Sampler
Vendor provided OpenCL implementations

SUMMARY

- Most of this work motivated by an interesting case of data dependent parallelism performance (clSURF)
- clSURF currently runs on CPUs, GPUs and APUs
- Profiling plays an increasingly important role in heterogeneous environments
- The OpenCL specification provides a useful interface to understand application performance
- Similar information provided for different devices
- Compliments existing tools such as the APP Profiler and Nvidia OpenCL Profiler
- Language extensions provide a path to high performance
- Enables static and dynamic profiling and feedback directed optimization

EXTRA HOMEWORK FOR NO REWARD

- clSURF code download
  - http://code.google.com/p/clsurf

- Haptic Download
  - http://code.google.com/p/clhaptic

- For more information about GPU research in NUCAR
  - www.ece.neu.edu/groups/nucar/GPU/

WE ARE THANKFUL FOR OUR GENEROUS SPONSORS ☺

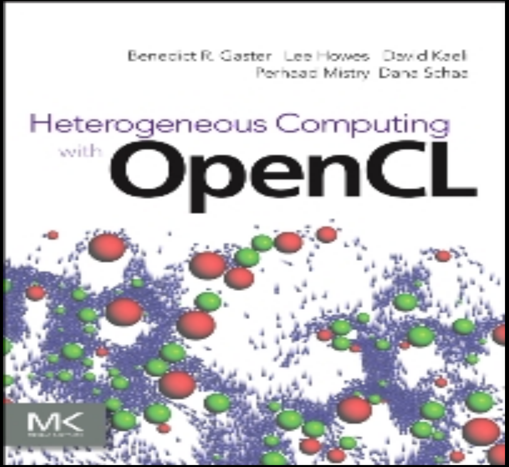
NORTHEASTERN UNIVERSITY LVX VERITAS VIRTVS FOUNDED A.D. 1898 BOSTON, MASSACHUSETTS

AMD

NSF

Google™

SAMSUNG


Benedict R. Gaster Lee Howes David Kaeli
Perhaad Mistry Dana Schaa

Heterogeneous Computing with OpenCL

MK

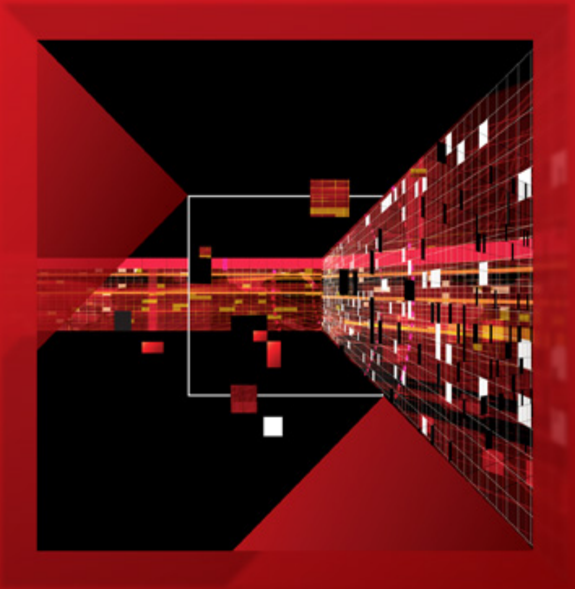
NVIDIA. CUDA™ RESEARCH CENTER

LOOKING INTO HIDDEN WORLDS GORDON CenSSIS

ANALOGIC®

ANALOG DEVICES

BK Medical



Thank You !
Questions or Comments ?

Perhaad Mistry

pmistry@ece.neu.edu

INFORMATION AND REFERENCES

- http://developer.amd.com/zones/OpenCLZone/universities/Pages/default.aspx
- General Programming
  - Beyond Programmable Shading – David Leubke
  - Decomposition Techniques for Parallel Programming – Vivek Sarkar
  - CUDA Textures & Image Registration - Richard Ansorge
  - Setting up CUDA within Windows Visual Studio
  - http://www.ademiller.com/blogs/tech/2011/03/using-cuda-and-thrust-with-visual-studio-2010/
  - SDK examples: Histogram64, Matmul, SimpleTextures
- SURF Related
  - http://code.google.com/p/clsurf/
  - http://www.chrisevansdev.com/computer-vision-opensurf.html
  - http://developer.amd.com/afds/assets/presentations/2123_final.pdf