

OpenCL Introduction

Scott Grauer-Gray

Readme for project updated

- OpenCL instructions should work now
- OpenACC instructions added

Simple Program A

- Initialize two arrays of size N
- Add values element-by-element
- Place output summations in another array
(of size N)

Simple Program A: C code

```
int main()
{
    float inArrayA[N];
    float inArrayB[N];
    float outArrayC[N];

    //function to set values in array in some manner
    initializeArray(inArrayA);
    initializeArray(inArrayB);

    for (int i=0; i < N; i++)
    {
        outArrayC[i] = inArrayA[i] + inArrayB[i];
    }

    //function to do "stuff" with the output data
    processOutputArray(outArrayC);

    return 0;
}
```

Program Execution

- By default
 - Performed on CPU using single core
 - May be possible to use additional resources to speed up program (specifically, more CPU cores or GPUs)
- Embarrassingly parallel problem
 - Loop iterations independent
 - No dependencies
 - Possible for each loop iteration to run simultaneously

Embarrassingly Parallel: Wikipedia

Embarrassingly parallel

From Wikipedia, the free encyclopedia



This article includes a [list of references](#), but **its sources remain unclear because it has insufficient [inline citations](#)**. Please help to [improve](#) this article by [introducing](#) more precise citations. *(May 2008)*

In [parallel computing](#), an **embarrassingly parallel** workload (or *embarrassingly parallel problem*) is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks.^[1]

Embarrassingly parallel problems tend to require little or no communication of results between tasks, and are thus different from [distributed computing](#) problems that require communication between tasks, especially communication of intermediate results. They are easy to perform on [server farms](#) which do not have any of the special infrastructure used in a true [supercomputer](#) cluster. They are thus well suited to large, internet based distributed platforms such as [BOINC](#).

A common example of an embarrassingly parallel problem lies within [graphics processing units](#) (GPUs) for the task of [3D projection](#), where each pixel on the screen may be rendered independently.

Contents [\[show\]](#)

Examples

[\[edit\]](#)

Some examples of embarrassingly parallel problems include:

- Distributed relational database queries using [distributed set processing](#) [↗](#)
- Serving static files on a webserver to multiple users at once.
- The [Mandelbrot set](#) and other fractal calculations, where each point can be calculated independently.
- [Rendering of computer graphics](#). In [ray tracing](#), each [pixel](#) may be [rendered](#) independently. In [computer animation](#), each [frame](#) may be rendered independently (see [parallel rendering](#)).^[*dubious – discuss*]
- [Brute-force searches](#) in [cryptography](#). A notable real-world example is [distributed.net](#).
- [BLAST searches](#) in [bioinformatics](#) for multiple queries (but not for individual large queries) ^[2]
- Large scale [face recognition](#) that involves comparing thousands of arbitrary acquired faces (e.g. a security or surveillance video via [closed-circuit television](#)) with similarly large number of previously stored faces (e.g. a "[rogues gallery](#)" or similar [watch list](#)) ^[3]

Acceleration Methods

- OpenMP (already presented)
 - Often used for multi-core CPUs
- MPI (already presented)
 - Often used for clusters with many nodes
- OpenCL (focus of this lecture)
 - Can be used for multi-core CPUs, GPUs, and other accelerators

Advantages to OpenCL

- Can run on many architectures
 - Relatively easy to port between multicore CPUs / GPUs / other accelerators
 - Supported by many vendors
 - NVIDIA and AMD GPUs
 - Vendors can add their own extensions

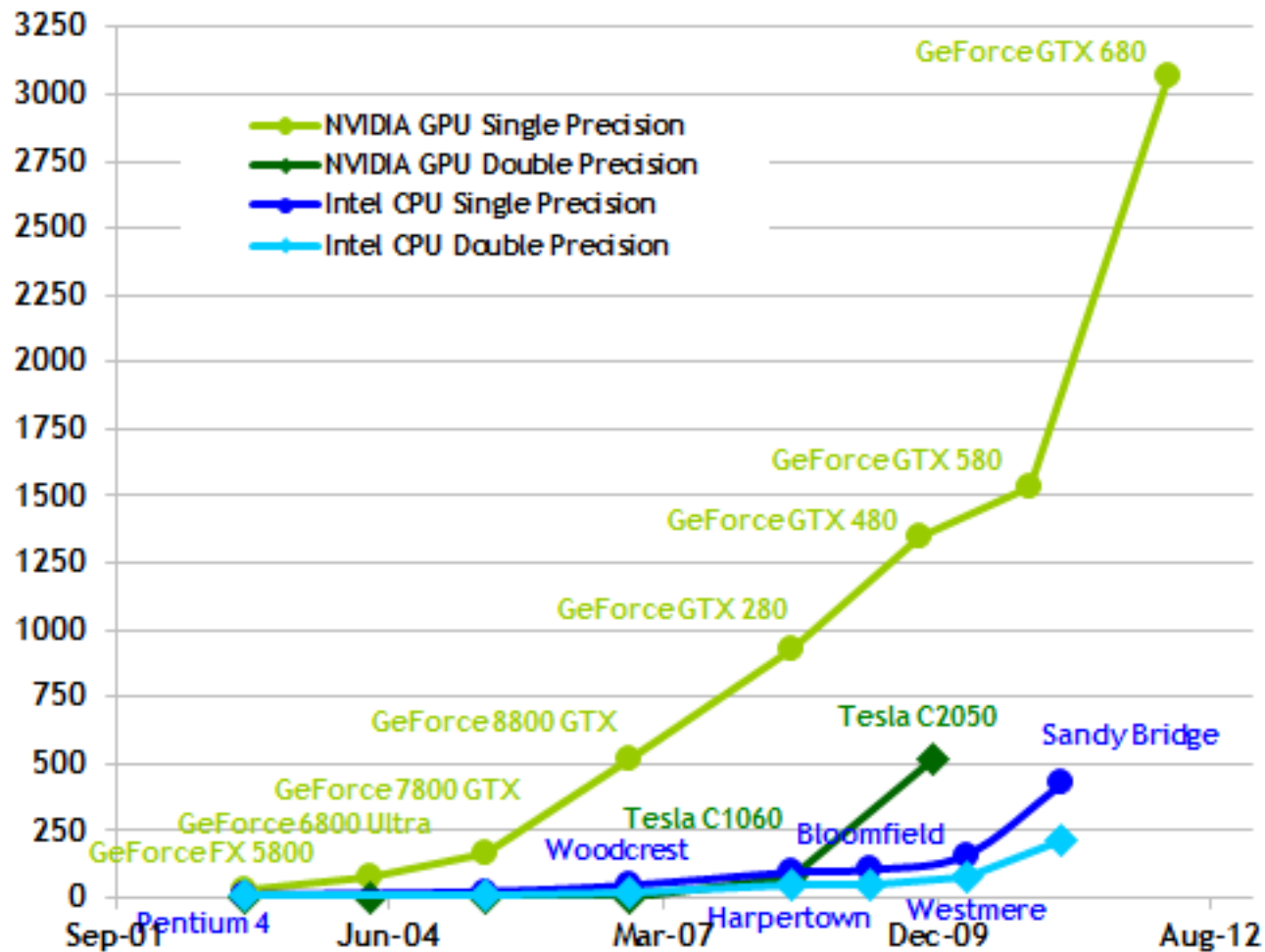
GPGPU

- GPGPU: General-purpose programming on the GPU
 - Take advantage of GPU with hundreds of simple cores
 - Typically use OpenCL or CUDA
 - Relatively new area of computing
 - Large speedup over CPU on certain applications
 - "Free" speedup with new architectures

GPGPU

Processing power of GPU vs CPU

Theoretical
GFLOP/s





AS YOU CAN SEE,
THE CENTRAL ROLE HERE
IS PLAYED BY THE
INTEL XEON
CPU...

INTEL

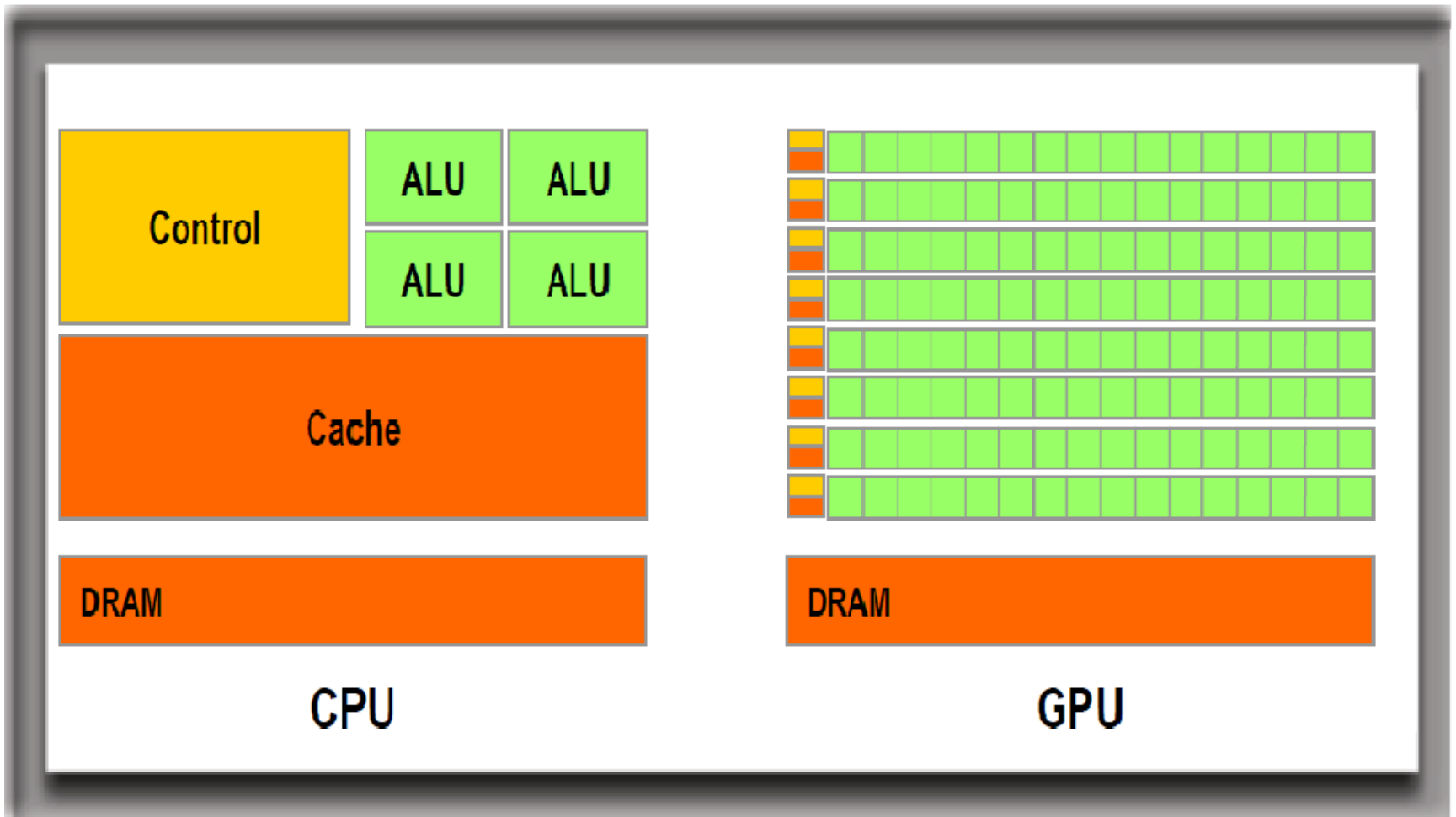
NVIDIA
GPU

INTEL
XEON
CPU

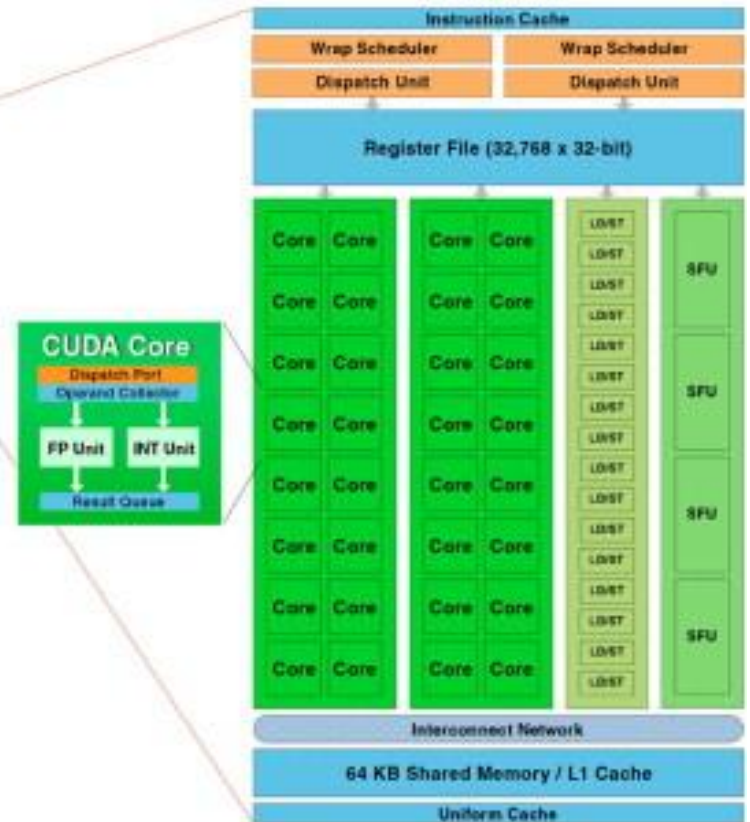
SUPERCOMPUTER

CPU Vs. GPU

Most transistors on GPU for computation



GPU Architecture: NVIDIA Fermi (2010)



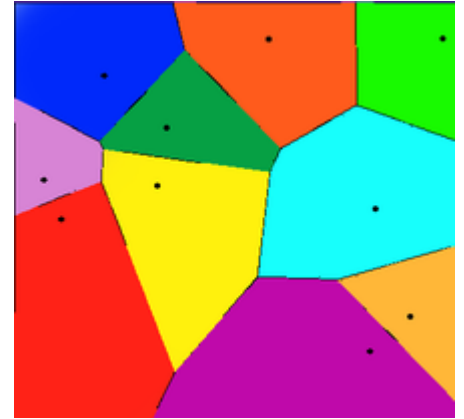
History of GPGPU

- GPGPU existed before creation of CUDA / OpenCL
 - Needed to use graphics API to take advantage of GPU
 - "Trick" GPU into thinking it was doing graphics instead of general-purpose computing
 - Fragment shader somewhat analogous to CUDA / OpenCL kernel
 - Major limitation: no scatter operation for output data
 - Still able to get speedup on some applications

History of GPGPU

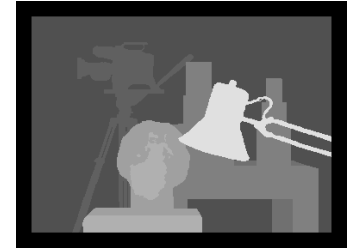
Notable early GPGPU Work (1999):

- GPGPU on Voronoi diagrams
 - Voronoi diagram: way of dividing space into regions
 - Set of specified "seeds" in region
 - For each seed there is a region of all points closest to that seed
 - **1999 SIGGRAPH PAPER:** "Fast computation of generalized Voronoi diagrams using graphics hardware" by Hoff et. al.
 - Implementation in OpenGL
 - Before programmable shaders on GPU
 - Takes advantage of z-buffer and rasterization capabilities of GPU
 - Been cited 508 times (according to google scholar)



History of GPGPU

Notable early GPGPU Work (2006):



● Stereo Vision

- "Belief propagation on the GPU for stereo vision" by A. Brunton, et. al.
- Belief propagation: Global stereo vision algorithm
 - Iterative message computation/passing step takes most computation time
 - Performed on half of image pixels in parallel in each iteration
 - Embarrassingly parallel
- Implementation in OpenGL (2006...pre-CUDA/OpenCL)
 - Use fragment shader to run message computation step in parallel
 - Data is stored as textures on the GPU
 - Results claim a 2x speedup over CPU runtime

History of GPGPU

Motivations for early GPGPU work

- Potential speedup over CPU
- May be processing data on GPU
 - Better to process data on GPU than transfer the data to the CPU for processing then back to the GPU for display
- Vendors not oblivious to interest in GPGPU
 - Added extensions to OpenGL to aid GPGPU
 - Addition of programmable vector/fragment shaders significant for graphics and GPGPU
 - OpenGL went from only supporting 8-bit textures to supporting a wide variety of data types, including 32-bit floats
 - Eventually developed ways for GPGPU without needing to use graphics API

History of GPGPU



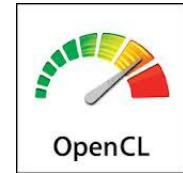
- CUDA

- Specifically for GPGPU
 - Removes "no scatter operation" limitation
- Introduced in February 2007
- Works on NVIDIA GPUs

- Close to Metal (CTM)


- Introduced by ATI/AMD for their GPUs
- ATI/AMD later switched to Stream SDK
- AMD now focused on OpenCL

OpenCL



- Open Standard for parallel programming of heterogeneous systems
 - Maintained by **KRONOS**
GROUP
CONNECTING SOFTWARE TO SILICON
- History of OpenCL
 - Initially developed by Apple
 - Became a collaboration between Apple, AMD, IBM, Intel, and NVIDIA
 - Specification approved for public release in December 2008
 - AMD, NVIDIA, and Apple released OpenCL implementations in 2009

OpenCL: Apple


 **Developer**

[Technologies](#) [Resources](#) [Programs](#) [Support](#) [Member Center](#)


- Overview
- Developer Tools
- iOS
- OS X**
- What's New in Mountain Lion
- Features in Lion
- Core Technologies
- Accessibility
- Audio and Video
- Cocoa
- Data Management
- Graphics and Animation
- Networking and Internet
- Features

OS X Core Technologies


OS X incorporates advanced core technologies that improve performance throughout the system. As a developer, you can use these advanced technologies in your app to make it faster, more responsive, and able to take advantage of the latest Mac hardware.



Grand Central Dispatch
A better way to do multicore
[Read more ↓](#)



64-Bit Throughout
The next big step
[Read more ↓](#)



OpenCL
Taking the graphics processor beyond graphics
[Read more ↓](#)



OpenCL

Harnessing the Power of the GPU for Your Application

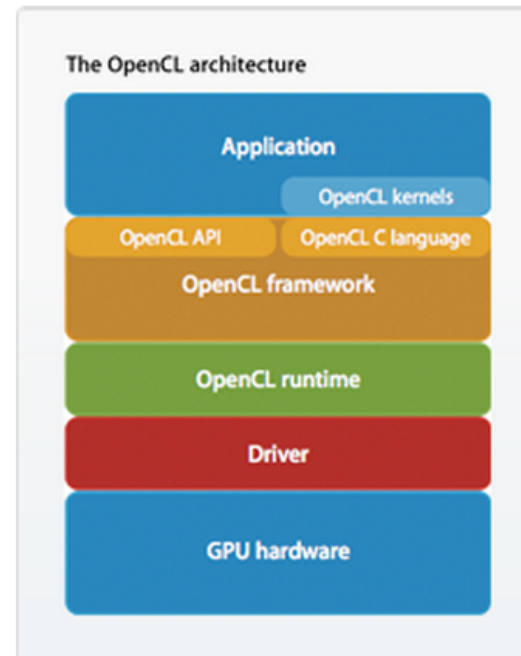
OpenCL dramatically accelerates your application by giving you the ability to access the amazing parallel computing power of the GPU. OpenCL also opens up a new range of computationally complex algorithms for use in your application. Possibilities include: Using OpenCL to bring sophisticated financial modeling techniques to accounting applications, performing cutting-edge analysis on large media files, and incorporating accurate physics and AI simulation into entertainment software.

Optimization Is Automatic

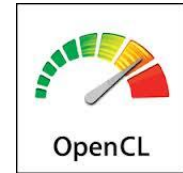
OpenCL uses runtime compilation to optimize for the kind of graphics processor in the Mac - automatically adjusting itself to the available processing power. OpenCL also rigorously defines numerical precision and accuracy, providing for consistent results across a wide-variety of GPU hardware.

Just Change the Code You Need

OpenCL stands for Open Computing Language. It uses an approachable C99-based language and a flexible API for managing parallel computation. To accelerate your application, simply use Xcode to rewrite the most performance-intensive parts of your code for OpenCL. The vast majority of your application's code can be left unchanged.



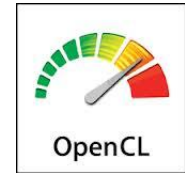
OpenCL



Steps for program:

1. Obtain OpenCL platform
2. Obtain device id for at least one device (accelerator)
3. Create context for device
4. Create accelerator program from source code
5. Build the program
6. Create kernel(s) from program functions
7. Create command queue for target device
8. Allocate device memory / move input data to device memory
9. Associate arguments to kernel with kernel object
10. Deploy kernel for device execution
11. Move output data to host memory
12. Release context/program/kernels/memory

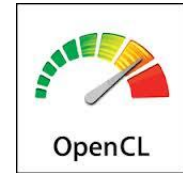
OpenCL



Step 1

- Obtain platform
 - Platform id identifies vendor installation of OpenCL
 - `clGetPlatformIDs(1, &platform, NULL);`
 - Functions often used twice, first to get the number of platforms and then for allocation

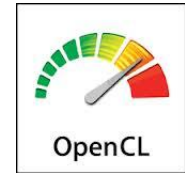
OpenCL



Step 2

- Obtain device id for at least one device (accelerator)
 - Use platform to get ID for device
 - `clGetDeviceIds(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);`
 - device id is stored in "device" variable
 - Functions often used twice, first to get the number of devices available and then for allocation

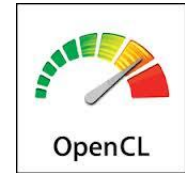
OpenCL



Step 3

- Create context for device
 - Context - abstract container attached to device
 - Contains program kernels, memory objects, etc
 - Holds command queue used for program execution
 - `context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);`

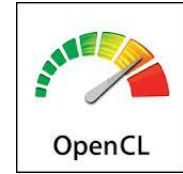
OpenCL



Step 4

- Create accelerator program from source code
 - Recommended to have a .cl file that contains kernels to run on accelerator
 - Read .cl file into a string on host
 - Create cl_program attached to context
 - ```
program = clCreateProgramWithSource
(context, 1, (const char**)
&program_buffer, &program_size, &err);
```

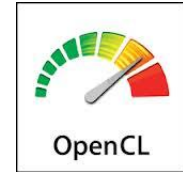
# OpenCL



## Step 5

- Build the program
  - OpenCL accelerator code is compiled at run-time
    - Host code will compile even if there are errors in accelerator code
    - Need to check for errors during run-time compilation
    - `clBuildProgram(program, 0, ...)`
    - Compilation error determined by error value returned from `clBuildProgram`
    - Calling `clGetProgramBuildInfo()` with the program object and the parameter `CL_PROGRAM_BUILD_STATUS` returns a string with the compiler output

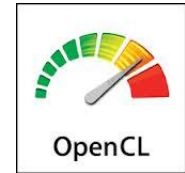
# OpenCL



## Step 6

- Create `cl_kernel(s)` from program functions
  - Use (now built) program as parameter to create kernel
  - `kernel = clCreateKernel(program, "kernel_name", &err)`
  - "kernel\_name" is the name of the kernel function to be run in parallel

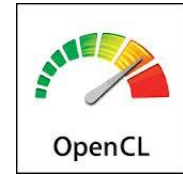
# OpenCL



## Step 7

- Create command queue for kernel dispatch
  - Command queue is attached to specific device
    - Mechanism for request that action be performed by device
    - Requests include memory transfer, begin executing kernel, etc
  - Can support out-of-order execution and profiling
  - `queue = clCreateCommandQueue(context, device, 0, &err)`

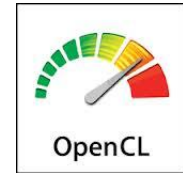
# OpenCL



## Step 8

- Allocate device memory / move input data to device
  - `memObject = clCreateBuffer (context, NULL, SIZE_N, NULL, &err)`
  - `clEnqueueWriteBuffer(command_queue, memObject, ..., TOTAL_SIZE, hostPointer, ...)`
  - Memory objects can be buffers or images
    - Focus on buffers
    - Contiguous memory chunks on GPU (global memory)
    - Read/write capable

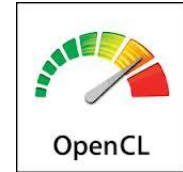
# OpenCL



## Step 9

- Associate arguments to kernel with kernel object
  - `cl_int clSetKernelArg (kernel, arg_index, arg_size, *arg_value)`
  - `arg_index` is index of argument in function signature (0 if first argument into function, etc)
  - Argument value is pointer to memory object if input parameter is array (buffer on GPU)
  - Argument value is pointer to primitive if input parameter is primitive value (such as a char, int, float, etc)

# OpenCL

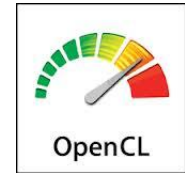


## Step 10

- Deploy kernel for device execution
  - Using `command_queue`, kernel object, and global and local (workgroup) sizes
    - `global_size = TOTAL_NUM_THREADS;`
    - `local_size = WORKGROUP_SIZE;`
      - All threads in workgroup execute on same compute unit
      - Access to fast local memory (shared within workgroup)
      - Can synchronize between threads in workgroup
    - `clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL);`



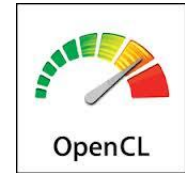
# OpenCL



## Step 11

- Write output device data back to host
  - `clEnqueueReadBuffer(command_queue, memObject, blocking_read, offset, TOTAL_SIZE, hostPointer, 0, NULL, NULL)`
    - Notable parameters
      - `command_queue`
      - `memObject`
      - `buffer size`
      - `target pointer on host`

# OpenCL



## Step 12

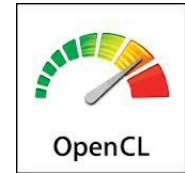
- **Release context/program/kernels/memory**
  - `clReleaseMemObject (memObject)`
  - `clReleaseKernel (kernel)`
  - `clReleaseProgram (program)`
  - `clReleaseContext (context)`

# OpenCL Kernel

## Steps for OpenCL kernel

- Assuming embarrassingly parallel problem
  - Each thread performs single loop iteration
    - Ideal on GPU
1. Retrieve ID corresponding to thread
  2. Make sure ID is within computation bounds
  3. Perform instruction(s) in loop body using thread ID

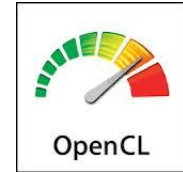
# OpenCL



## Step 1: Kernel

- Retrieve ID corresponding to thread
  - **threadId = get\_global\_id** (curr\_dimension)
  - If parallelizing single loop, dimension will be 0

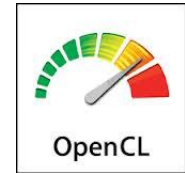
# OpenCL



## Step 2: Kernel

- Make sure ID is within computation bounds
  - **Assume loop iterates from  $i=0$  to  $N-1$**
  - `if ((threadId >= 0) && (threadId < N))`
    - Perform computation

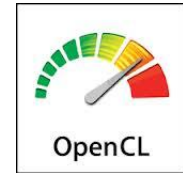
# OpenCL



## Step 3: Kernel

- Perform instruction(s) in loop body using thread ID
  - **Using Simple Program A**
    - `outArrayC[threadId] = inArrayA[threadId] + inArrayB[threadId];`

# OpenCL



## OpenCL Kernel for Simple Program A

```
__kernel void addArrays(__global float* inArrayA, __global
float* inArrayB, __global float* outArrayC, int nVal)
{
 int threadId = get_global_id (0);
 if ((threadId >= 0) && (threadId < nVal))
 {
 outArrayC[threadId] = inArrayA[threadId] + inArray
[threadId];
 }
}
```

- **Note that input/output arrays in global memory space**
- **GPU arrays are memory objects on host**
- **"int nVal" parameter is a primitive type input**

# OpenCL Memory Spaces

## **\_\_global**

Memory in global address space (DRAM on GPU)

## **\_\_constant**

Special type of read-only memory (may be faster)

## **\_\_local**

Memory shared within work-group of kernels

May be on-chip and much faster than global

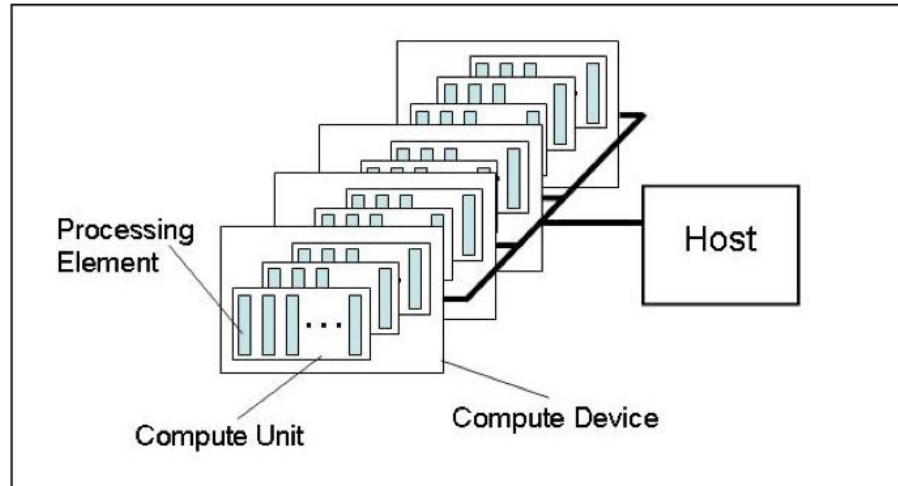
## **\_\_private**

Private per work-item (thread)



# OpenCL Device

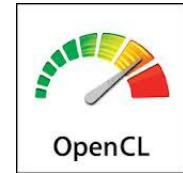
- Can be CPU, GPU, or other accelerator
- Contains global memory
- Number of compute units (cores on CPU, streaming multiprocessors on GPU)
  - Each compute unit contains processing elements and (potentially fast) local memory
  - All threads within a work-group execute on same compute unit
    - Allows synchronization and local memory sharing within work-group



# Determining Best Workgroup Size

- Depends on device
- Likely higher on GPU than CPU
  - More processing elements per compute unit on GPU
- Intel recommends workgroup size of 64-128
- Often 128 is minimum to get good performance on GPU
  - On NVIDIA Fermi, workgroup size must be at least 192 for full utilization of cores
  - If using a lot of registers or local memory, may be necessary/optimal to use smaller workgroup sizes
  - Something to experiment with
  - Optimal workgroup size differs across applications

# OpenCL



## Steps for Host:

1. Obtain OpenCL platform
2. Obtain device id for at least one device (accelerator)
3. Create context for device
4. Create accelerator program from source code
5. Build the program
6. Create kernel object(s) from program functions
7. Create command queue for target device
8. Allocate device memory / move input data to device memory
9. Associate arguments to kernel with kernel object
10. Deploy kernel for device execution
11. Move output data to host memory
12. Release context/program/kernels/memory

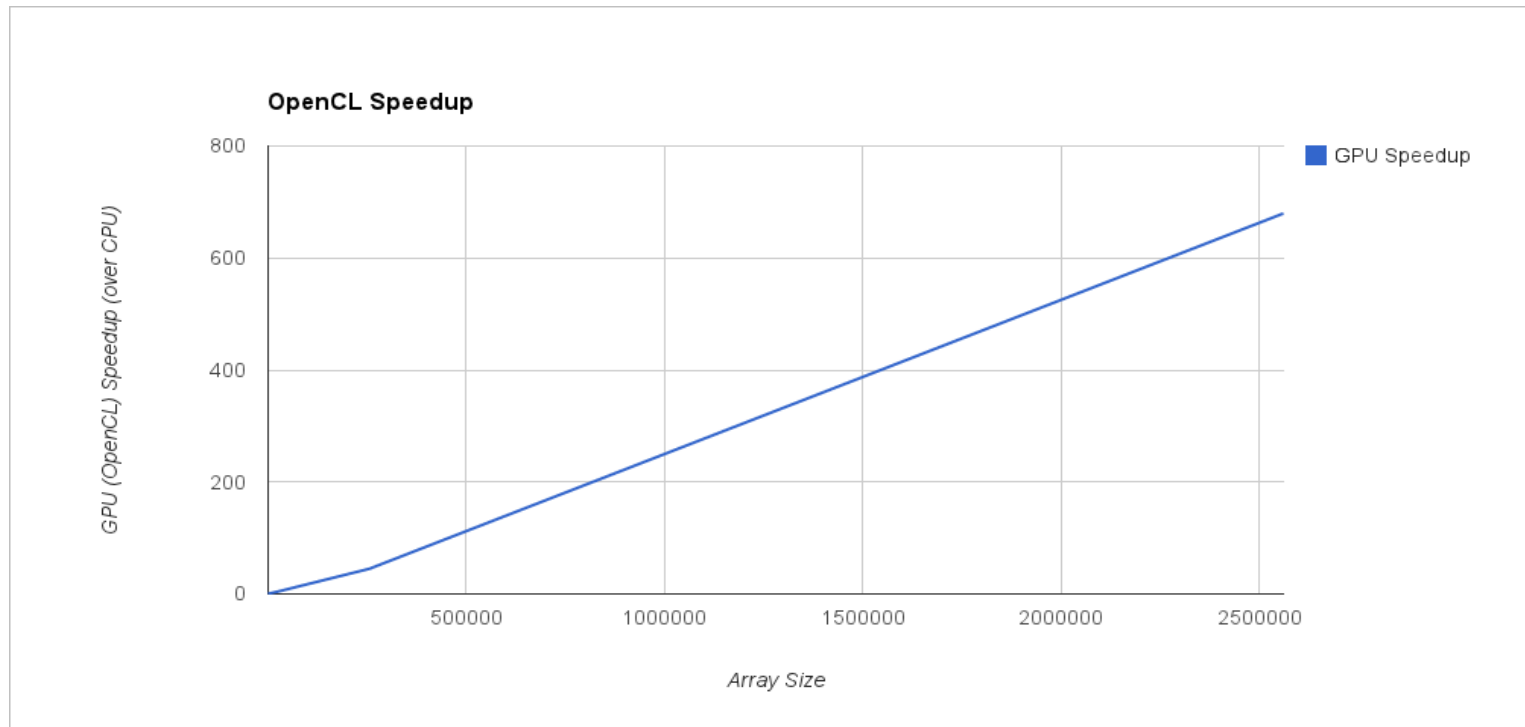
# OpenCL

## Simple Program A demo

- Code will be posted
- Should be able to use as template for project 1
  - Need to adjust code for reading .cl file (step 4) and timing (unless using Windows)

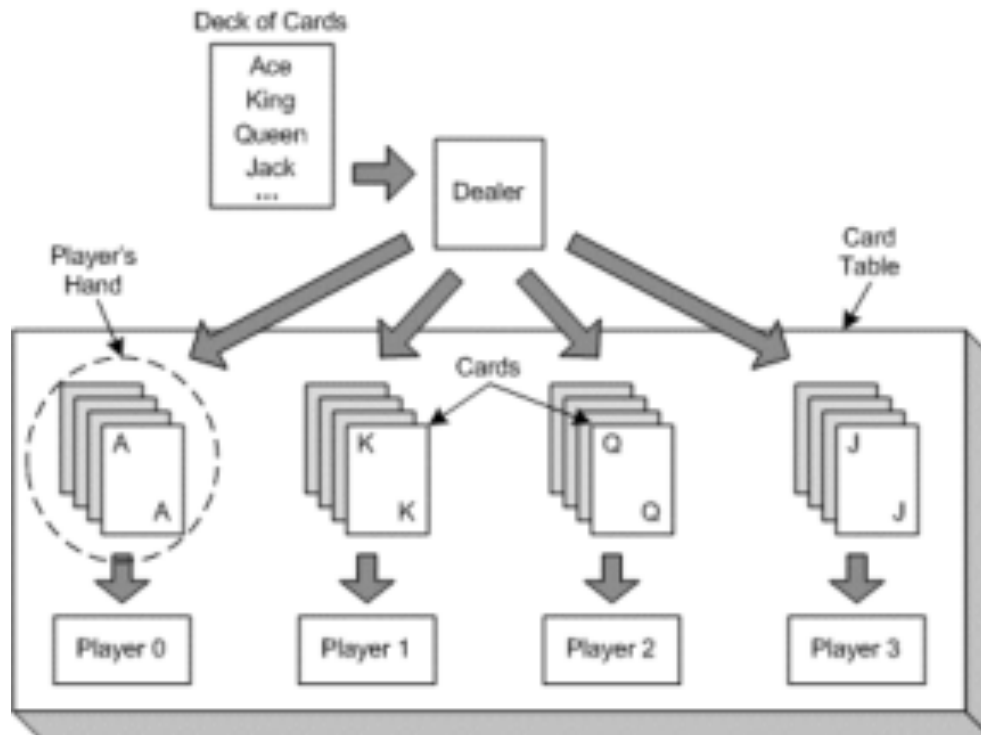
# OpenCL Speedup on Simple Program A

- GPU: 660M
  - 384 CUDA cores
  - Compared to single-core CPU
  - Speedup over 500x with array size of over 2 million



# Analogy for OpenCL environment

- From Dr. Dobb's site: A Gentle Introduction to OpenCL (by Matthew Sharpino)
- Card game analogy



# Analogy for OpenCL environment

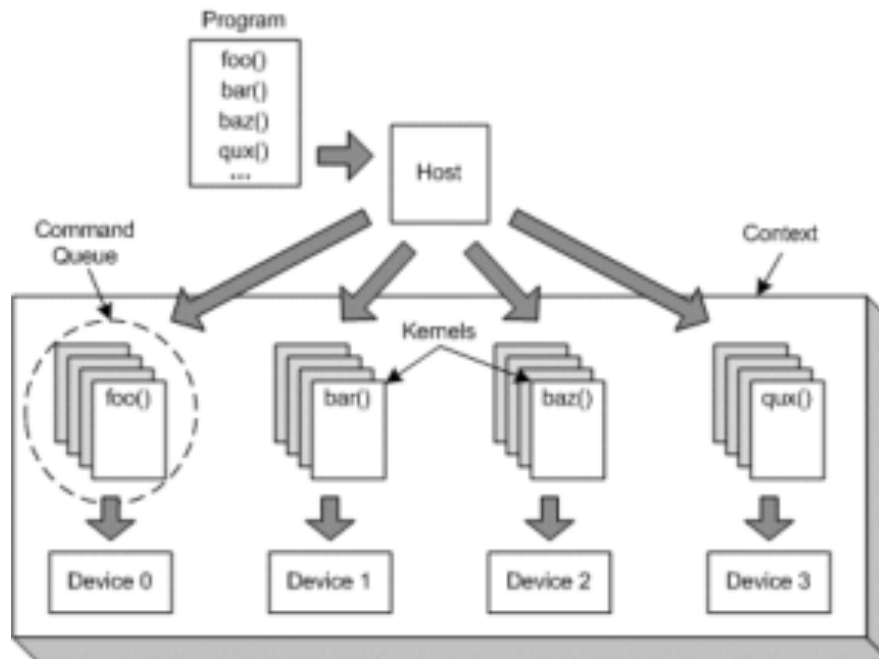
**Host - card dealer**

**OpenCL devices - card players**

Player receives cards from dealer <--> device receives kernels from host

**OpenCL Kernels - cards**

Dealer distributes cards to players <--> Host distributes kernels to devices



# Analogy for OpenCL environment

## OpenCL Program - deck of cards

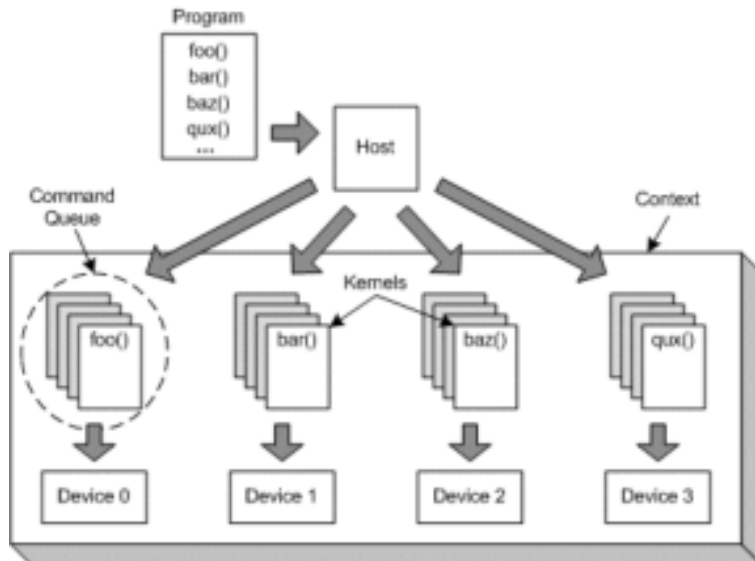
Dealer selects cards from a deck <--> Host selects kernels from a program

## Command Queue - player's hand

Each player receives cards as part of a hand <--> Each device receives kernels through command queue

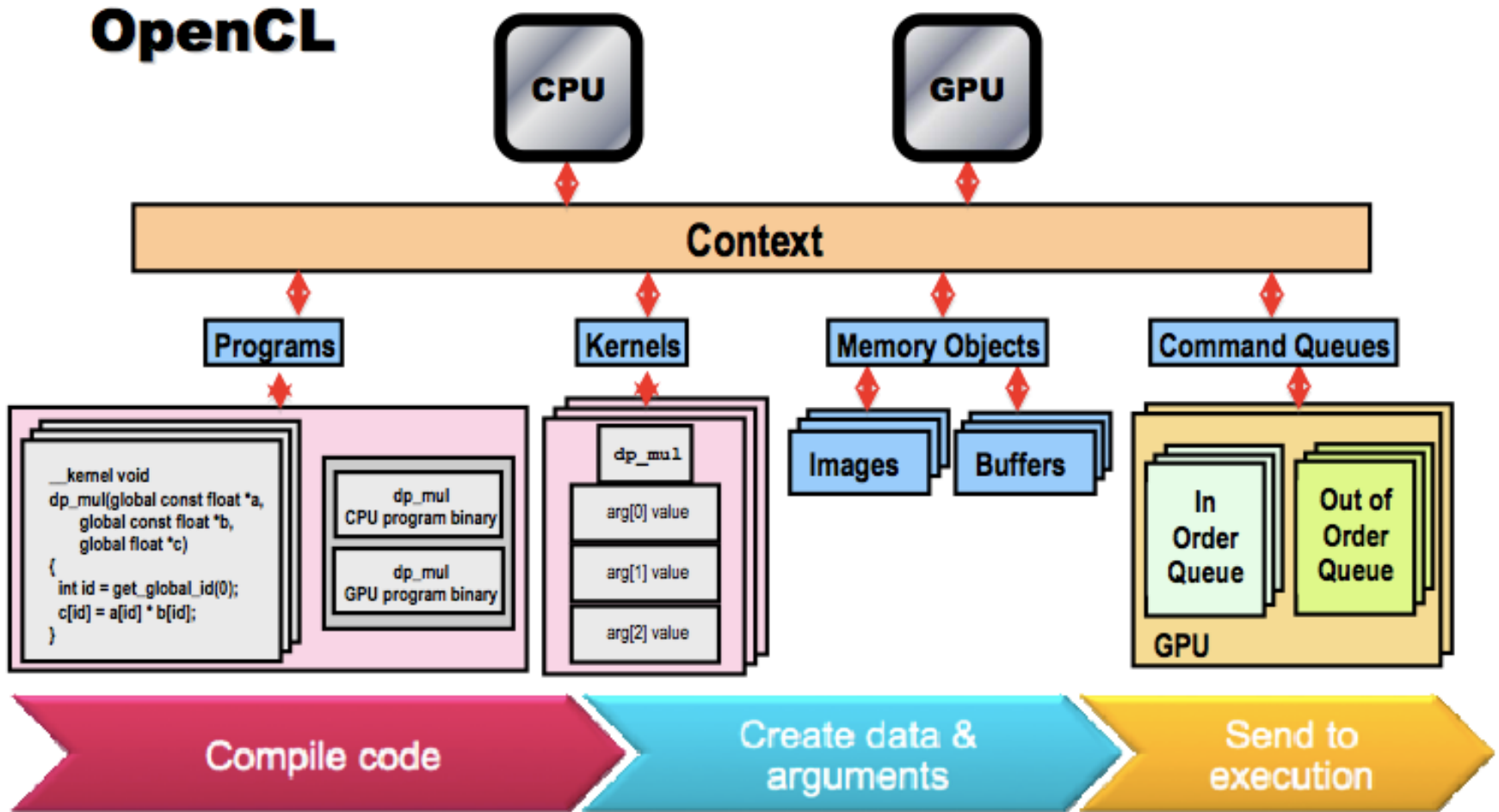
## OpenCL Context - card table

Card table makes it possible for players to transfer cards to each other <--> OpenCL Context allows devices to receive kernels and transfer data





# OpenCL Illustration



# OpenCL

- Entire OpenCL specification available at <http://www.khronos.org>
  - Contains detailed information about each function
- Additional resources
  - Cavazos' slides from class last year (<http://www.eecis.udel.edu/~cavazos/cisc879-spring2012/>)
  - NVIDIA SDK code samples (in CUDA 4.0-4.2)
  - AMD APP code samples
  - Other online OpenCL documentation