INTRODUCTION TO OPENCL

HAIBO XIE, PH.D.
haibo.xie@amd.com

AGENDA

- What's OpenCL
- Fundamentals for OpenCL programming
- OpenCL programming basics
- OpenCL programming tools
- Examples & demos

WHAT IS OPENCL

Open Computing Language (**OpenCL**) is a framework

- For writing parallel computing programs that execute across heterogeneous platforms

OpenCL is a programming model

- To fulfill parallel computing thought in the Heterogeneous Computing era

OpenCL includes

- Language for writing Kernels
- APIs to use and control the platform
- Compilers for cross-platform binary generation

OpenCL is an open standard

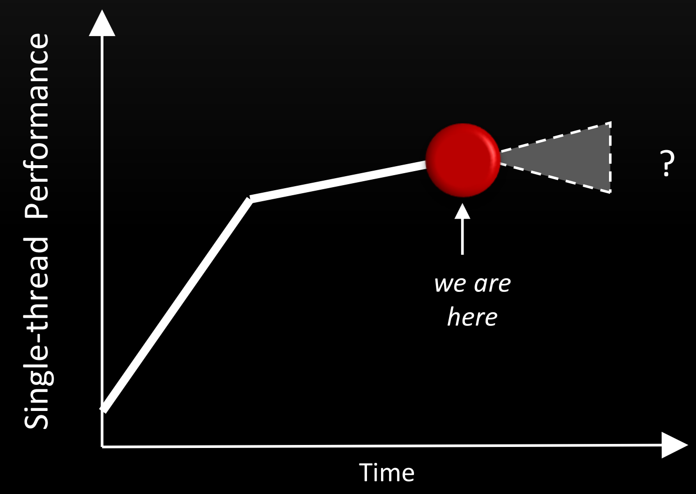
ARCHITECTURE EVOLUTION

AMD

Single-Core Era

Enabled by:
- ✓ Moore's Law
- ✓ Voltage Scaling

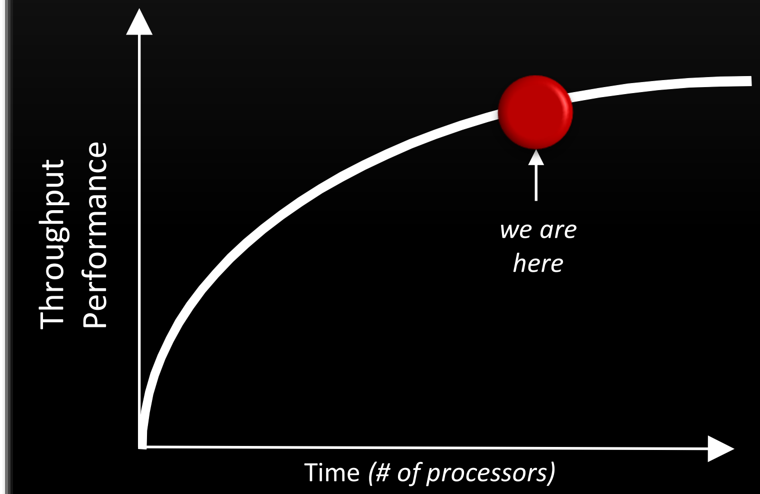
Constrained by:
- ✗ Power
- ✗ Complexity



Multi-Core Era

Enabled by:
- ✓ Moore's Law
- ✓ SMP architecture

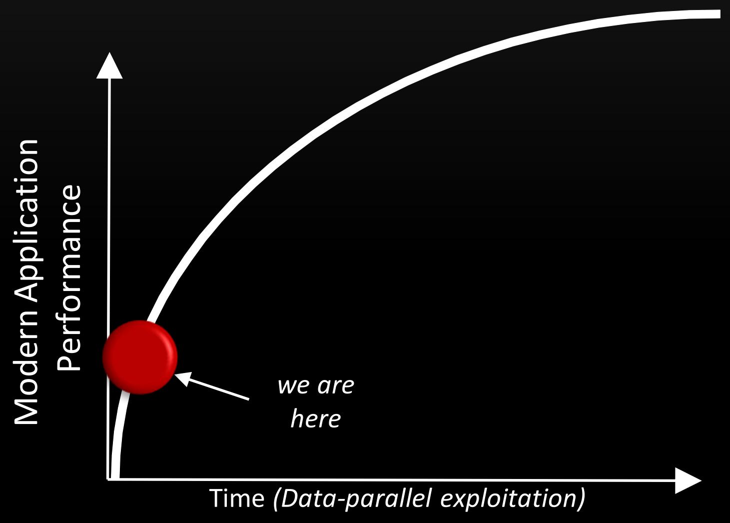
Constrained by:
- ✗ Power
- ✗ Parallel SW
- ✗ Scalability



Heterogeneous Systems Era

Enabled by:
- ✓ Abundant data parallelism
- ✓ Power efficient GPUs

Temporarily
Constrained by:
- ✗ Programming models
- ✗ Comm.overhead



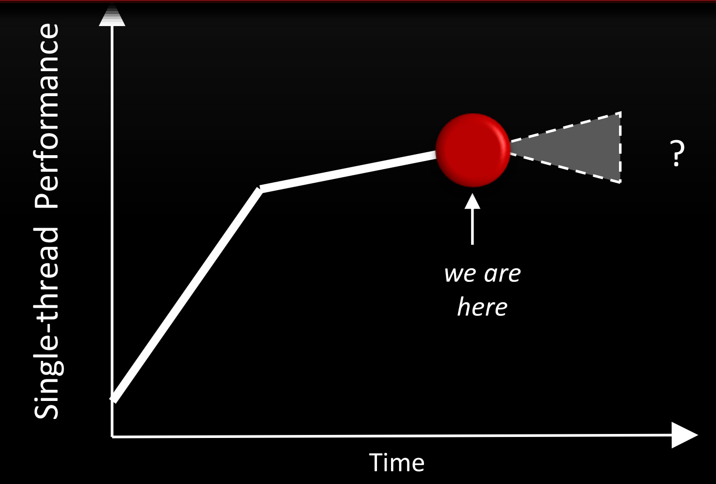
PROGRAMMING MODEL EVOLUTION

Single-Core Era

Enabled by:
- ✓ Moore's Law
- ✓ Voltage Scaling

Constrained by:
- ✗ Power
- ✗ Complexity

Assembly ➡ C/C++ ➡ Java …

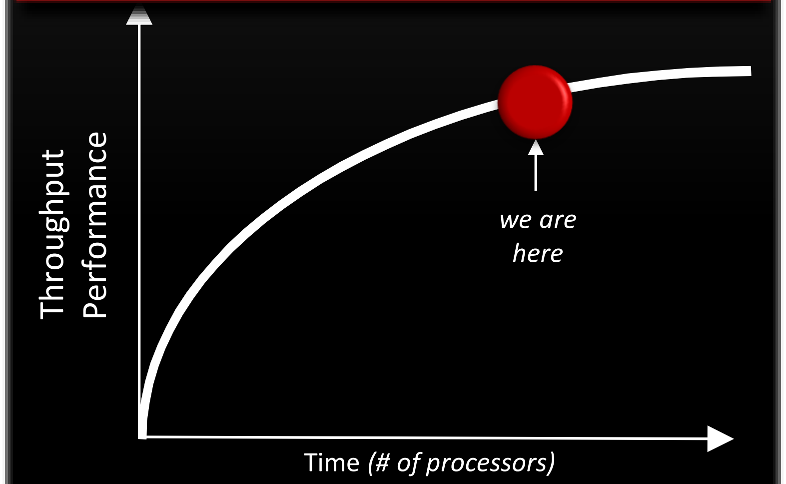


Multi-Core Era

Enabled by:
- ✓ Moore's Law
- ✓ SMP architecture

Constrained by:
- ✗ Power
- ✗ Parallel SW
- ✗ Scalability

pthreads ➡ OpenMP / TBB …

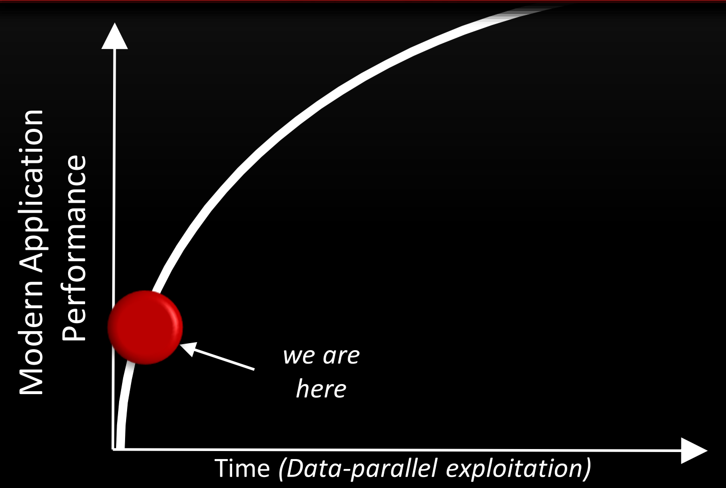


Heterogeneous Systems Era

Enabled by:
- ✓ Abundant data parallelism
- ✓ Power efficient GPUs

Temporarily
Constrained by:
- ✗ Programming models
- ✗ Comm.overhead

Shader ➡ CUDA ➡ OpenCL ➡ C++ AMP ➡ Java



WHAT'S HETEROGENEOUS COMPUTING

Heterogeneous computing systems refer to electronic systems that use a variety of different types of computational units with different instruction set architectures (ISAs).

Compute units are:

General-purpose processor

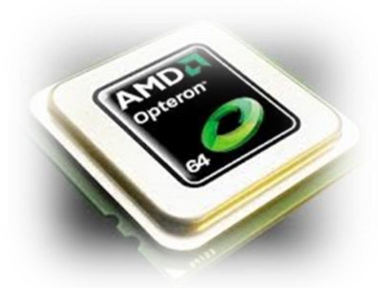
- Multi-core CPUs

Special-purpose processor

- Graphics Processing Unit (GPU)
- Digital Signal Processor (DSP)
- Field-Programmable Gate Array (FPGA)
- Custom acceleration logic (application-specific integrated circuit (ASIC))

WIKIPEDIA
The Free Encyclopedia

TYPICAL HETEROGENEOUS SYSTEM – CPU + dGPU

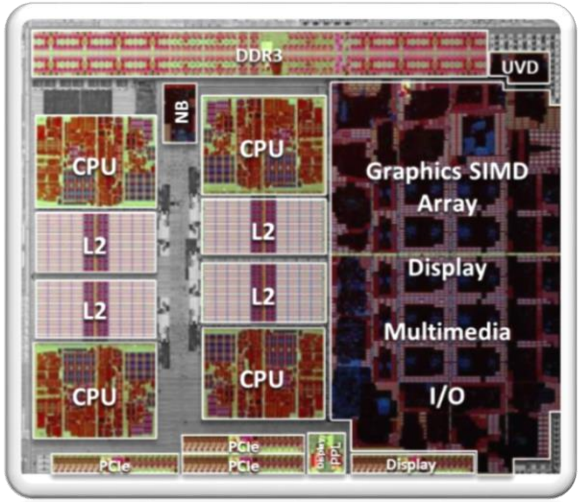


CPU + dGPU

Common form factor of recent GPGPU
2-16 x86 cores
1-4 GPU cards
Tens of TFLOPS

Distributed memory system between CPU and GPU
PCI-E communication as a bottleneck
Very fine granularity parallelism needed
Expert programmer but better learning curve than Cell B.E
Kinds of programming model supported, CG/CUDA/OpenCL/C++ AMP

TYPICAL HETEROGENEOUS SYSTEM – AMD HSA APU



AMD APU, codename Kevari

Third generation APU chip
Up to 4 x86 general purpose core
Combine GPU into the single die
More than 1TFLOPS single precision float operation

Unified memory system between CPU and GPU
Industry standard programming model – OpenCL
Kinds of high level programming languages support, C/C++/Java, etc
Way to future Full HSA enablement.

EVOLUTION OF HETEROGENEOUS COMPUTING

AMD

Excellent

Poor

Proprietary Drivers Era

Graphics & Proprietary Driver-based APIs

- "Adventurous" programmers
- Exploit early programmable "shader cores" in the GPU
- Make your program look like "graphics" to the GPU
- CUDA™, Brook+, etc

2002 - 2008

Standards Drivers Era

OpenCL™, DirectCompute Driver-based APIs

- Expert programmers
- C and C++ subsets
- Compute centric APIs , data types
- Multiple address spaces with explicit data movement
- Specialized work queue based structures
- Kernel mode dispatch

2009 - 2011

Architected Era

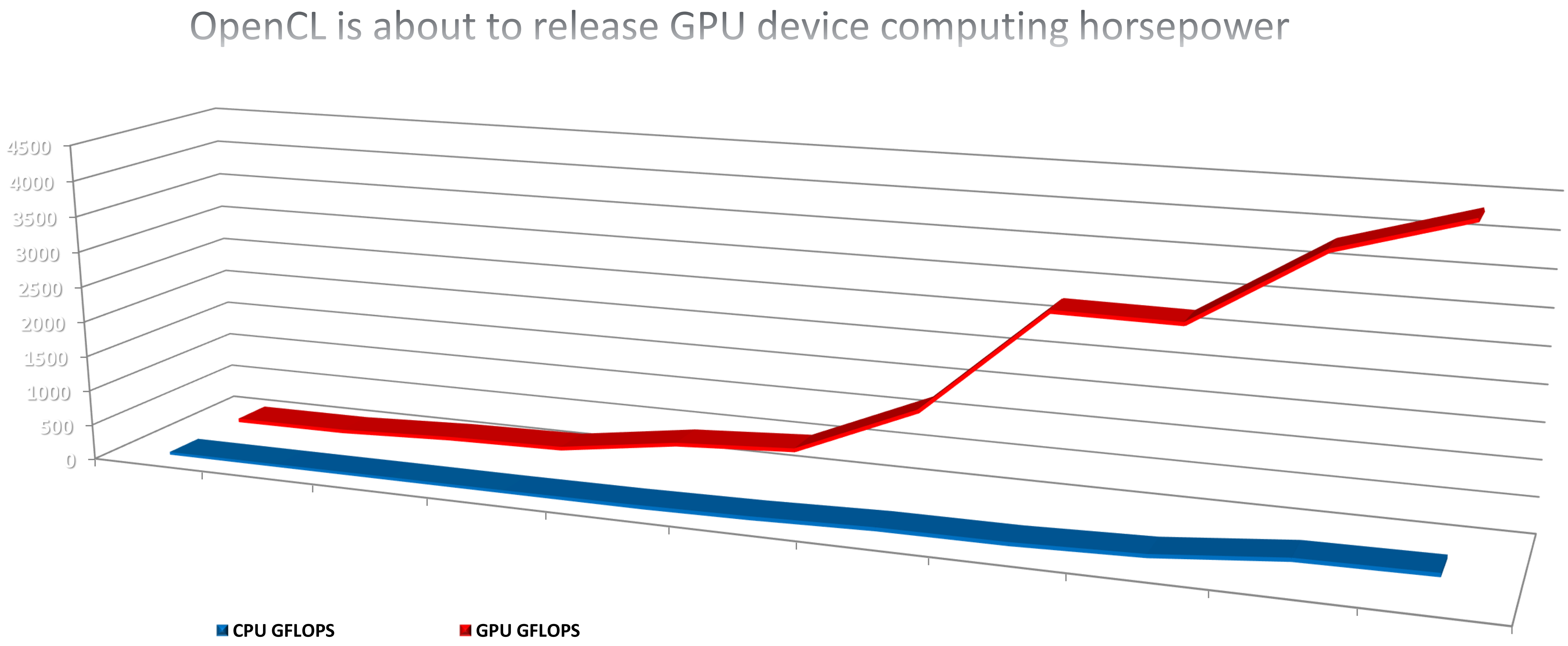
Heterogeneous System Architecture GPU Peer Processor

- Mainstream programmers
- Full C++
- GPU as a co-processor
- Unified coherent address space
- Task parallel runtimes
- Nested Data Parallel programs
- User mode dispatch
- Pre-emption and context switching

2012 - 2020

GPU COMPUTE CAPABILITY IS MORE THAN 10X THAT OF THE CPU

AMD



AN OPENCL STANDARD

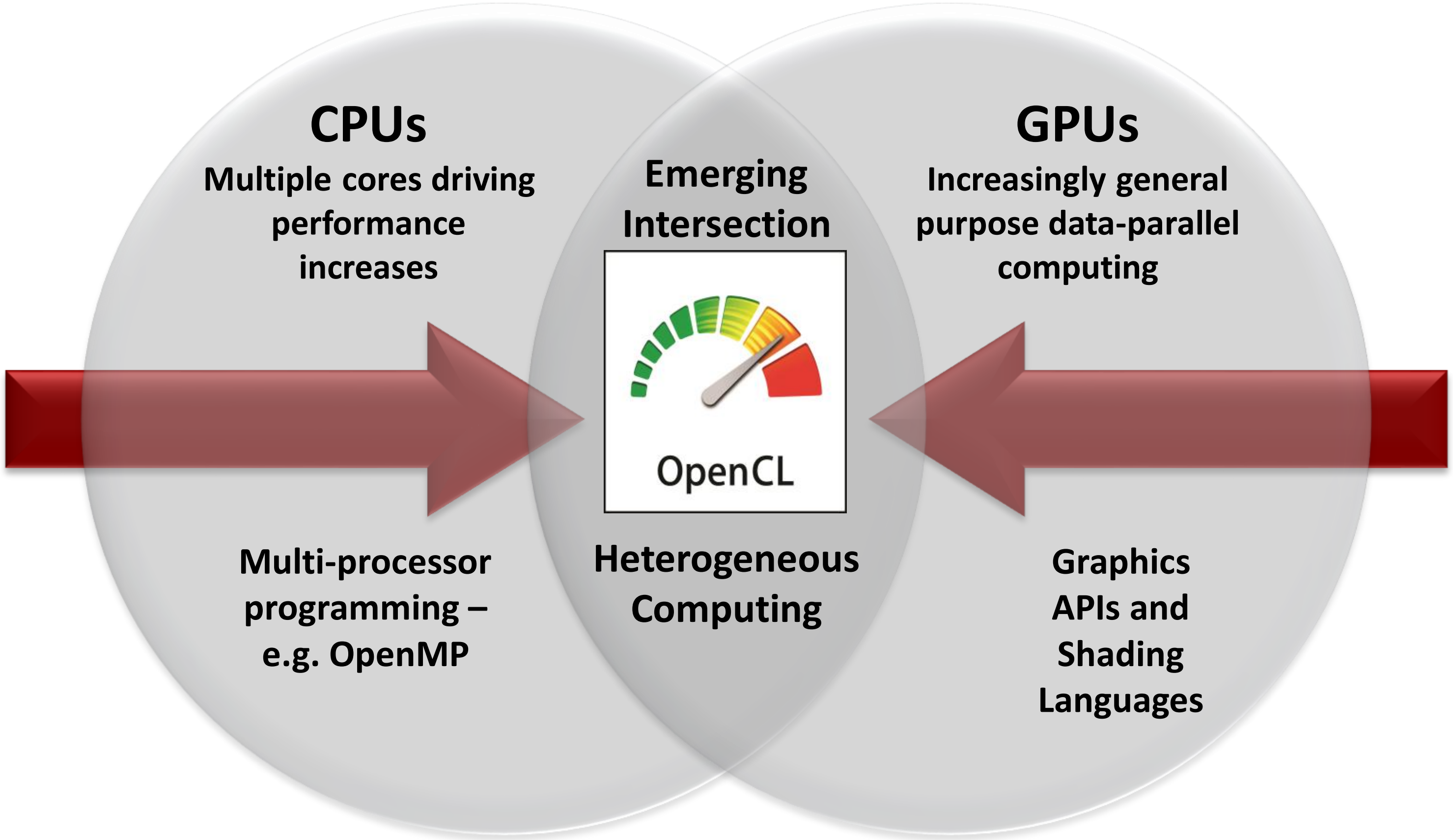
Open Standard

Cross Platform

Multi-Vendor

Royalty Free

Broad ISV Support



OpenCL™ is a programming framework for heterogeneous compute resources

AN OPENCL STANDARD

OPENCL GAINING MOMENTUM

AMD

N.America

APIs for Current Multi-Threaded Development

The most popular multi-threaded development API used by developers in th survey is OpenMP (Open Multi-Processing), which supports multi-platform share memory multiprocessing programming in C, C++, and FORTRAN. OpenMP currently used by 31% of respondents. OpenCL (Open Computing Language), framework for writing programs that execute across various processor platform follows at 28%. Another 25% use Intel Threading Building Blocks, a C++ templat library that leverages Intel's multi-core processors.

Which of the following do you program with today?	Count	Percent of Responses	Percent of Cases
OpenMP	91	13.9	31.1
OpenCL	81	12.4	27.6
Intel Threading Building Blocks	72	11.0	24.6
Intel Parallel Building Blocks	65	10.0	22.2
CUDA	59	9.0	20.1
Intel Cilk Plus	56	8.6	19.1
MPI	50	7.7	17.1
Co Array Fortran	34	5.2	11.6
Other	145	22.2	49.5
	----	----	----
Total Responses	653	100	222.9

North American Development Survey: Vol. I, © 2011 Evans Data Corp.

Note that this multiple response question allowed the developers to select as many responses as they wished, and thus the total number of cases will not come to 100%. The response column shows the percent of total responses, while the case column shows the percent of actual developers (cases) who responded.

APAC

APIs for Current Multi-Threaded Development

Developers regularly rely on libraries and APIs to make it easier to accomplish difficult tasks. This maxim is especially true of threading applications, since keeping track of every thread in ones application would be difficult and provide opportunity for errors. With poor threading, applications may crash.

The most popular APIs for multi-threaded development currently are Intel's Threading Building Blocks, OpenMP, and OpenCL.

Intel TBB is a C++ template library that adds parallel programming for C++ programmers. The open source library includes algorithms, highly concurrent containers, locks and atomic operations, a task scheduler and a scalable memory allocator.

Which of the following do you program with today?	Count	Percent of Responses	Percent of Cases
Intel Threading Building Blocks	143	18.4	43.7
OpenMP	114	14.7	34.9
OpenCL	95	12.2	29.1
Intel Parallel Building Blocks	79	10.2	24.2
MPI	71	9.1	21.7
Intel Cilk Plus	63	8.1	19.3
CUDA	58	7.5	17.7
Co Array Fortran	42	5.4	12.8
Other	111	14.3	33.9
	----	----	----
Total Responses	776	100	237.3

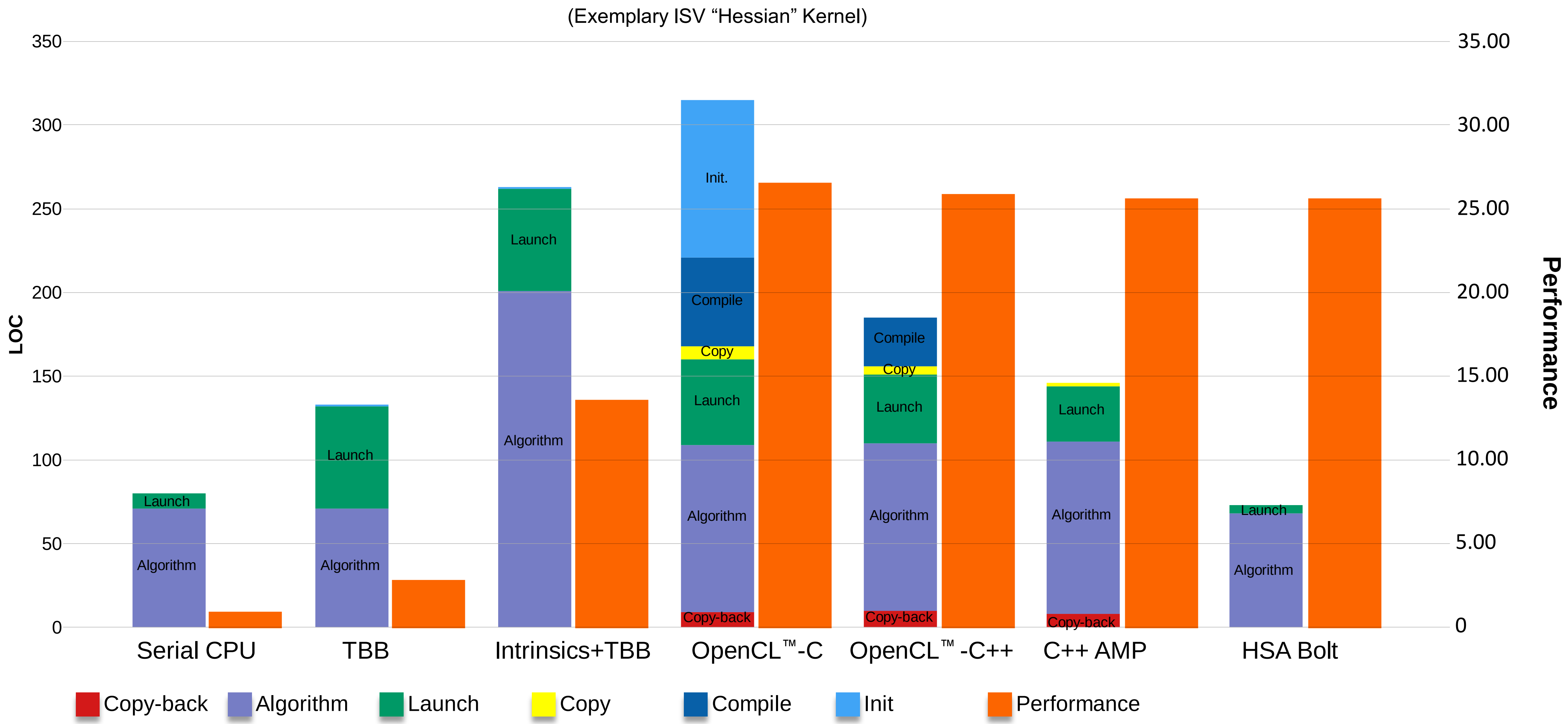
APAC Development Survey: Vol. I, © 2011 Evans Data Corp.

Note that this multiple response question allowed the developers to select as many responses as they wished, and thus the total number of cases will not come to 100%. The response column shows the percent of total responses, while the case column shows the percent of actual developers (cases) who responded.

Market data provided by Evans Data Corporation | June 2011

LINES-OF-CODE AND PERFORMANCE

WITH DIFFERENT PROGRAMMING MODEL



AMD A10-5800K APU with Radeon™ HD Graphics – CPU: 4 cores, 3800MHz (4200MHz Turbo); GPU: AMD Radeon HD 7660D, 6 compute units, 800MHz; 4GB RAM.
Software – Windows 7 Professional SP1 (64-bit OS); AMD OpenCL™ 1.2 AMD-APP (937.2); Microsoft Visual Studio 11 Beta

AGENDA

- What's OpenCL
- Fundamentals for OpenCL programming
- OpenCL programming basics
- OpenCL programming tools
- Demos

FUNDAMENTALS FOR OPENCL PROGRAMMING

- Parallel computing thinking
  - Parallel computing thinking is a must-have for OpenCL programming on GPU devices which work as a many-core computing device

- Knowledge of GPU architecture
  - GPU has a quite different architectural philosophy against CPU

- Ideas of controlling and cooperating heterogeneous devices
  - Heterogeneous Computing is not like parallel computing on a SMP device
  - Developers should carefully control the different part of this system
  - And coordinate them smoothly
  - Will covered by OpenCL programming basics section

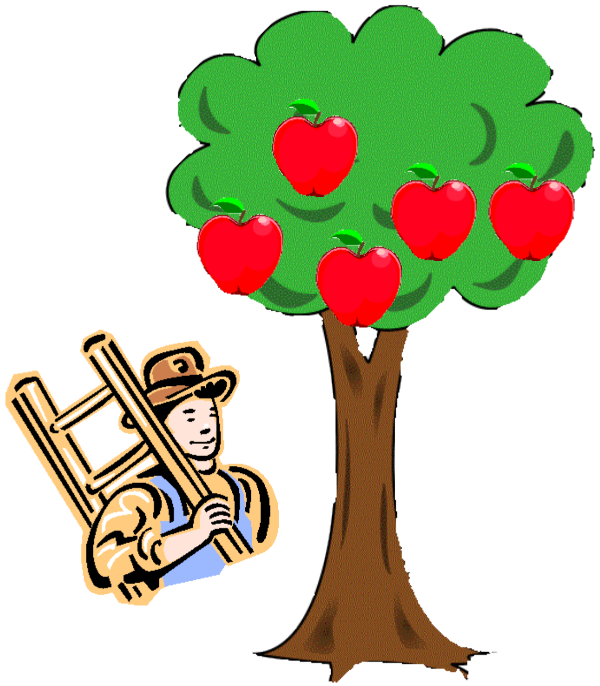
PARALLELISM

AMD

- *Parallelism* describes the potential to complete multiple parts of a problem at the same time
- In order to exploit parallelism, we have to have the physical resources (i.e. hardware) to work on more than one thing at a time
- There are different types of parallelism that are important for GPU computing:
  - *Task parallelism* – the ability to execute different tasks within a problem at the same time
  - *Data parallelism* – the ability to execute parts of the same task (i.e. different data) at the same time

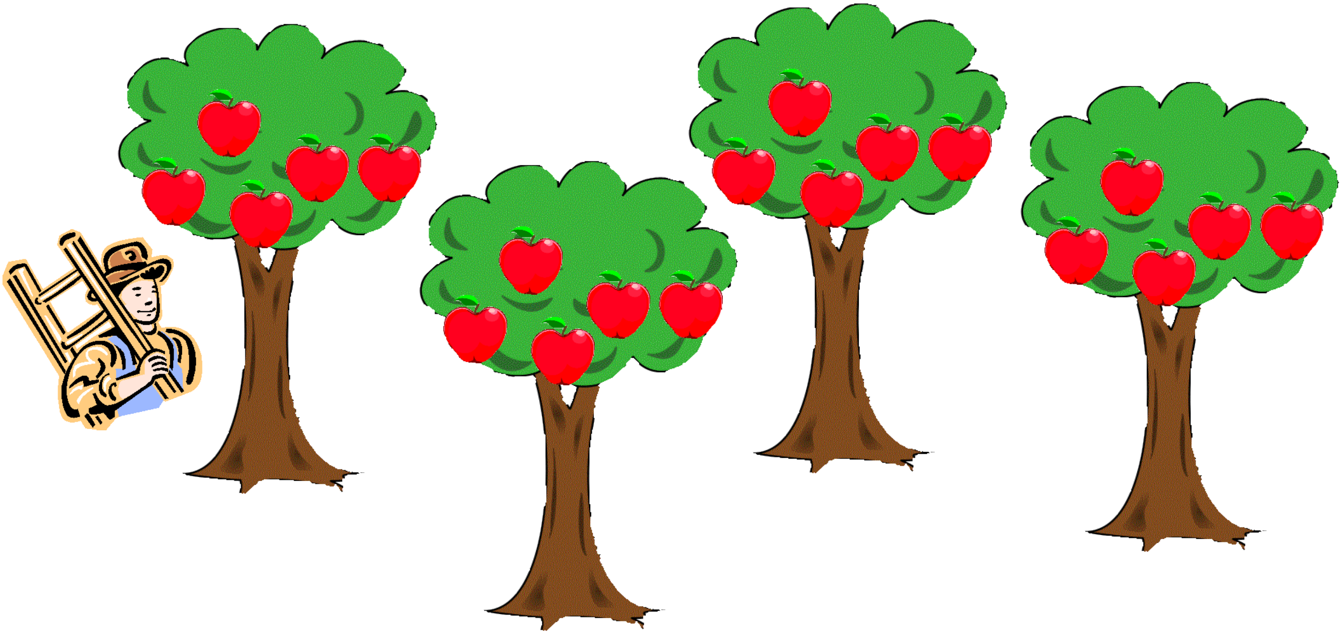
PARALLELISM

- As an analogy, think about a farmer who hires workers to pick apples from an orchard of trees
  - The workers that do the apple picking are the (hardware) processing elements
  - The trees are the tasks to be executed
  - The apples are the data to be operated on



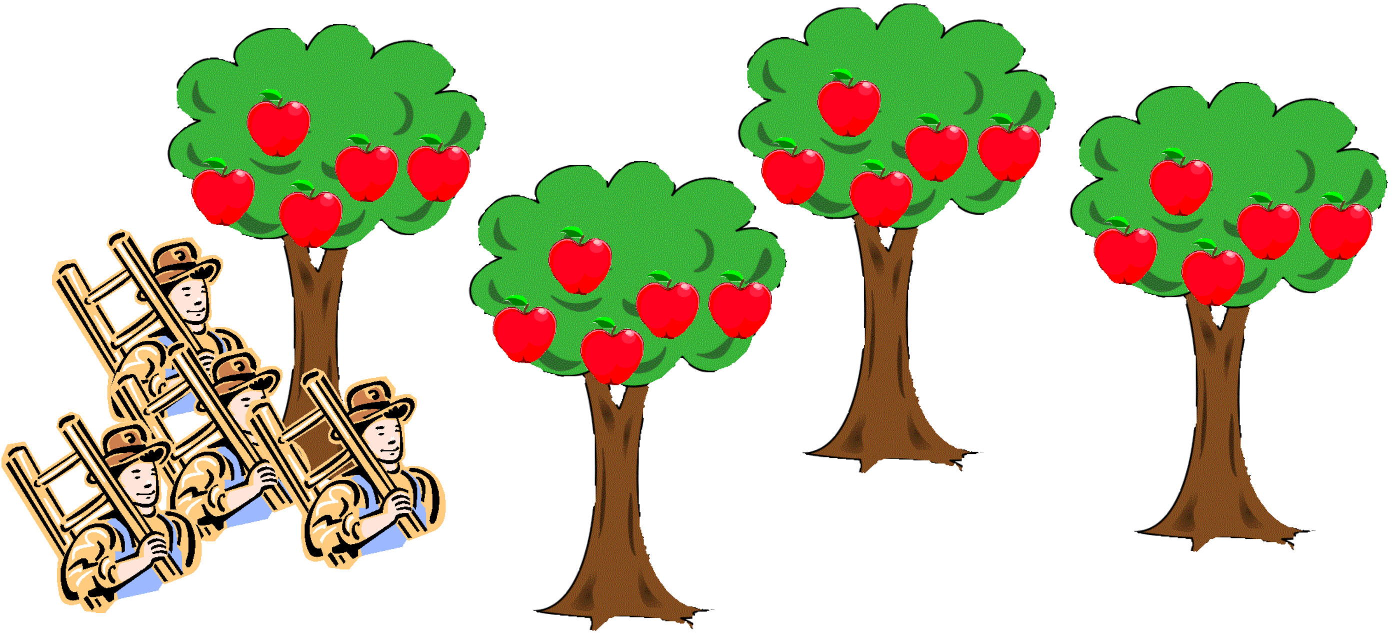
PARALLELISM

- The *serial* approach would be to have one worker pick all of the apples from each tree
  - After one tree is completely picked, the worker moves on to the next tree and completes it as well



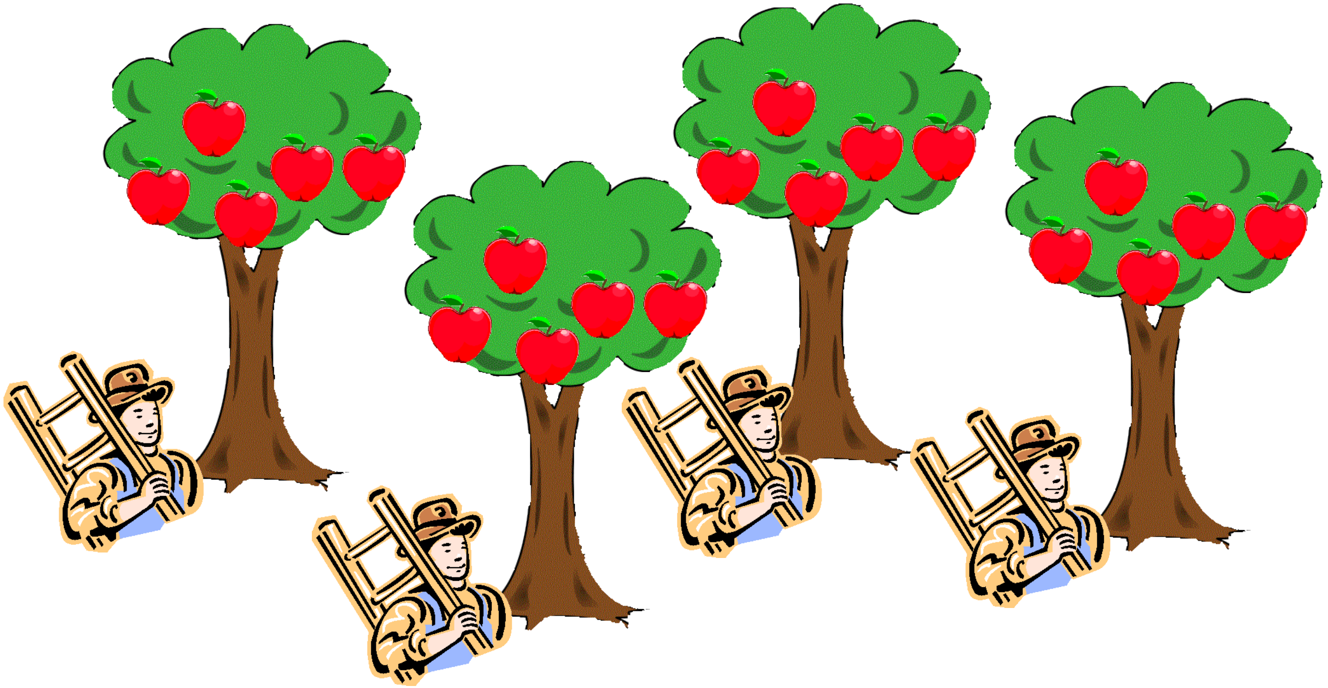
PARALLELISM

- If the workers uses both of his arms to pick apples, he can grab two at once
  - This represents data parallel hardware, and would allow each task to be completed quicker
  - A worker with more than two arms could pick even more apples



PARALLELISM

- If more workers were hired, each worker could pick apples from a different tree
  - This represents task parallelism, and although each task takes the same time as in the serial version, many are accomplished in parallel

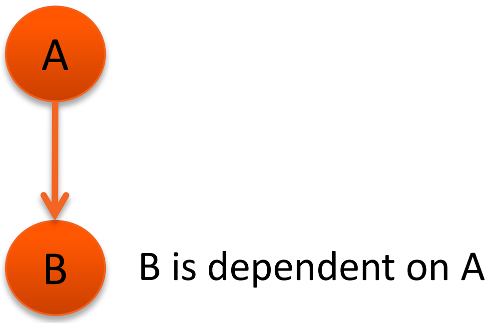


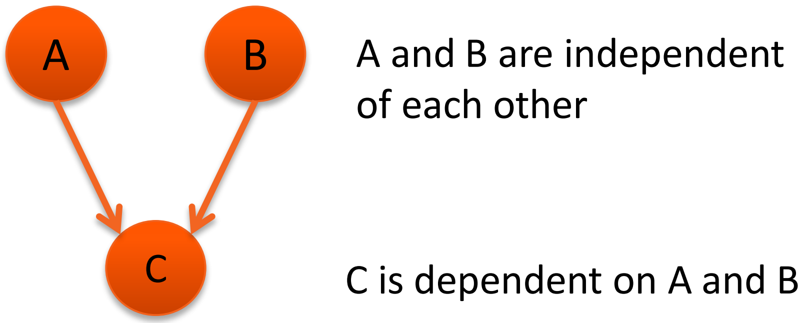
DECOMPOSITION

- For non-trivial problems, it helps to have more formal concepts for determining parallelism
- When we think about how to parallelize a program we use the concepts of decomposition:
  - *Task decomposition*: dividing the algorithm into individual tasks (don’t focus on data)
    - In the previous example the goal is to pick apples from trees, so clearing a tree would be a task
  - *Data decomposition*: dividing a data set into discrete chunks that can be operated on in parallel
    - In the previous example we can pick a different apple from the tree until it is cleared, so apples are the unit of data

TASK DECOMPOSITION

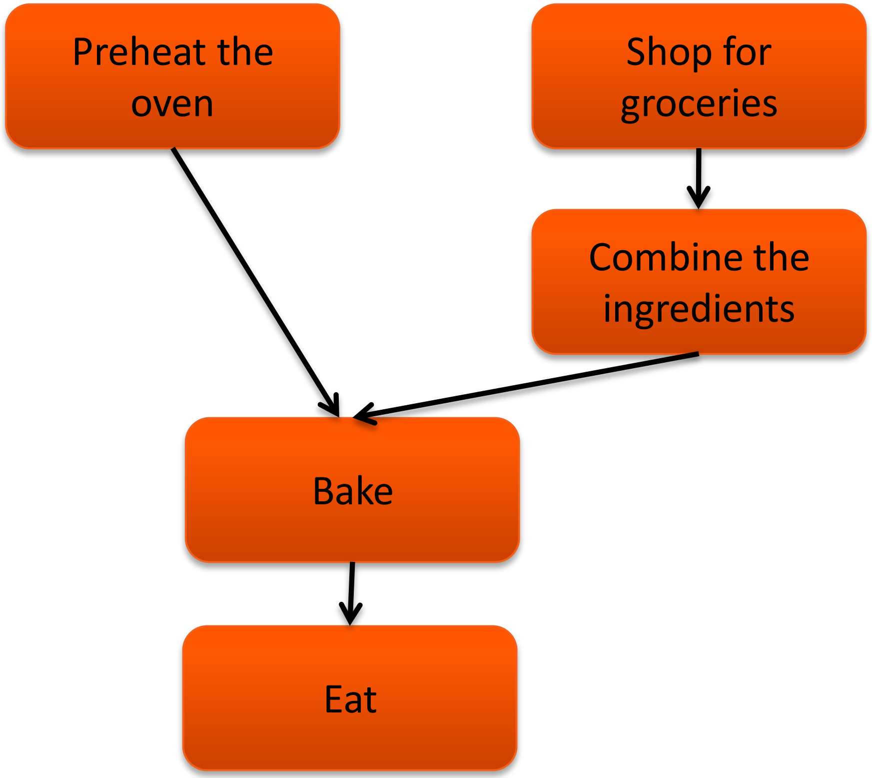
- Task decomposition reduces an algorithm to functionally independent parts
- Tasks may have dependencies on other tasks
  - If the input of task B is dependent on the output of task A, then task B is dependent on task A
  - Tasks that don't have dependencies (or whose dependencies are completed) can be executed at any time to achieve parallelism
  - *Task dependency graphs* are used to describe the relationship between tasks





TASK DEPENDENCY GRAPHS

- We can create a simple task dependency graph for baking cookies
  - Any tasks that are not connected via the graph can be executed in parallel (such as preheating the oven and shopping for groceries)



OUTPUT/INPUT DATA DECOMPOSITION

- For most scientific and engineering applications, data is decomposed based on the output data
  - Each output pixel of an image convolution is obtained by applying a filter to a region of input pixels
  - Each output element of a matrix multiplication is obtained by multiplying a row by a column of the input matrices
- This technique is valid any time the algorithm is based on one-to-one or many-to-one functions

- Input data decomposition is similar, except that it makes sense when the algorithm is a one-to-many function
  - A histogram is created by placing each input datum into one of a fixed number of bins
  - A search function may take a string as input and look for the occurrence of various substrings
- For these types of applications, each thread creates a “partial count” of the output, and synchronization, atomic operations, or another task are required to compute the final result

PARALLEL COMPUTING

- The choice of how to decompose a problem is based solely on the algorithm
- However, when actually implementing a parallel algorithm, both hardware and software considerations must be taken into account

- There are both hardware and software approaches to parallelism
- Much of the 1990s was spent on getting CPUs to *automatically* take advantage of Instruction Level Parallelism (ILP)
  - Multiple instructions (without dependencies) are issued and executed in parallel
  - Automatic hardware parallelization will not be considered for the remainder of the lecture
- Higher-level parallelism (e.g. threading) cannot be done automatically, so software constructs are required for programmers to tell the hardware where parallelism exists
  - When parallel programming, the programmer must choose a programming model and parallel hardware that are suited for the problem

PARALLEL HARDWARE

AMD

- Hardware is generally better suited for some types of parallelism more than others

Hardware type	Examples	Parallelism
Multi-core superscalar processors	Phenom II CPU	Task
Vector or SIMD processors	SSE units (x86 CPUs)	Data
Multi-core SIMD processors	Radeon 7970 GPU	Data

- Currently, GPUs are comprised of many independent “processors” that have SIMD processing elements
  - One task is run at a time on the GPU
  - *Loop strip mining* (next slide) is used to split a data parallel task between independent processors
  - Every instruction must be data parallel to take full advantage of the GPU’s SIMD hardware
    - SIMD hardware is discussed later in the lecture

LOOP STRIP MINING

- *Loop strip mining* is a loop-transformation technique that partitions the iterations of a loop so that multiple iterations can be:
  - executed at the same time (vector/SIMD units),
  - split between different processing units (multi-core CPUs),
  - or both (GPUs)
- An example with loop strip mining is shown in the following slides

PARALLEL SOFTWARE – SPMD

- GPU programs are called *kernels*, and are written using the Single Program Multiple Data (SPMD) programming model
  - SPMD executes multiple instances of the same program independently, where each program works on a different portion of the data
- For data-parallel scientific and engineering applications, combining SPMD with loop strip mining is a very common parallel programming technique
  - Message Passing Interface (MPI) is used to run SPMD on a distributed cluster
  - POSIX threads (pthreads) are used to run SPMD on a shared-memory system
  - Kernels run SPMD within a GPU

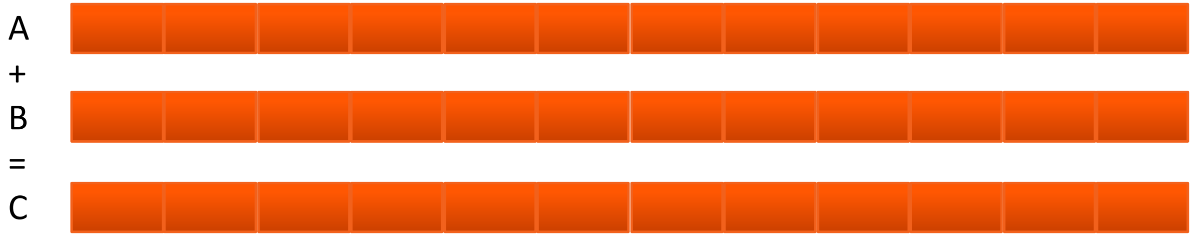
PARALLEL SOFTWARE – SPMD

AMD

- Consider the following vector addition example

```
for( i = 0:11 ) {
  C[ i ] = A[ i ] + B[ i ]
}
```

Serial program:
one program completes
the entire task



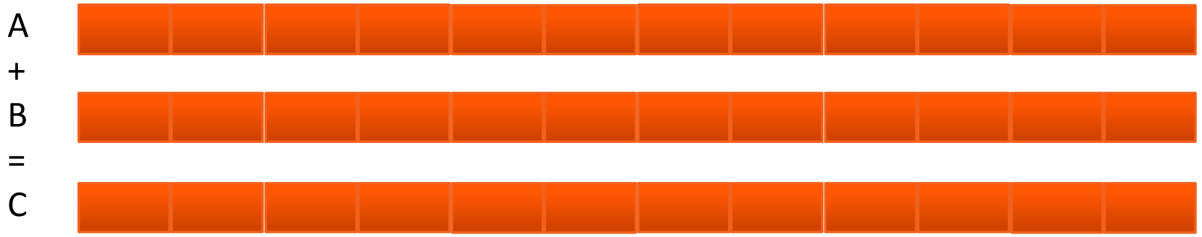
- Combining SPMD with loop strip mining allows multiple copies of the same program execute on different data in parallel

```
for( i = 0:3 ) {
  C[ i ] = A[ i ] + B[ i ]
}
```

```
for( i = 4:7 ) {
  C[ i ] = A[ i ] + B[ i ]
}
```

```
for( i = 8:11 ) {
  C[ i ] = A[ i ] + B[ i ]
}
```

SPMD program:
multiple copies of the
same program run on
different chunks of the
data



30

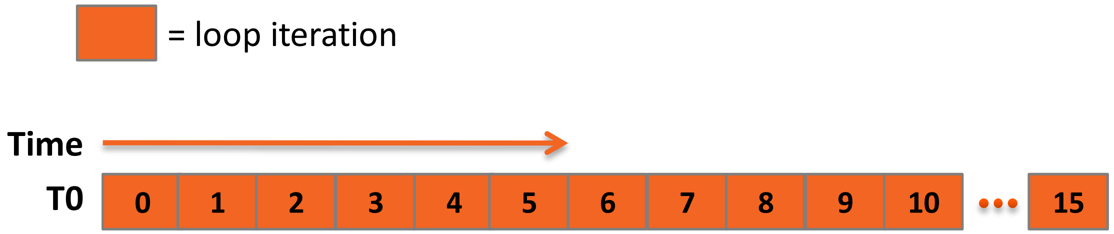
PARALLEL SOFTWARE – SPMD

- In the vector addition example, each chunk of data could be executed as an independent thread
- On modern CPUs, the overhead of creating threads is so high that the chunks need to be large
  - In practice, usually a few threads (about as many as the number of CPU cores) and each is given a large amount of work to do
- For GPU programming, there is low overhead for thread creation, so we can create one thread per loop iteration

PARALLEL SOFTWARE – SPMD

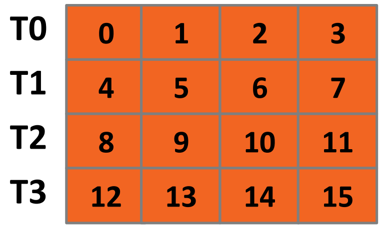
Single-threaded (CPU)

```
// there are N elements
for(i = 0; i < N; i++)
  C[i] = A[i] + B[i]
```



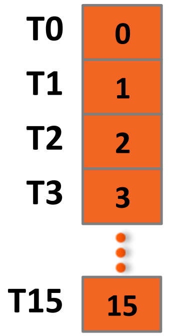
Multi-threaded (CPU)

```
// tid is the thread id
// P is the number of cores
for(i = 0; i < tid*N/P; i++)
  C[i] = A[i] + B[i]
```



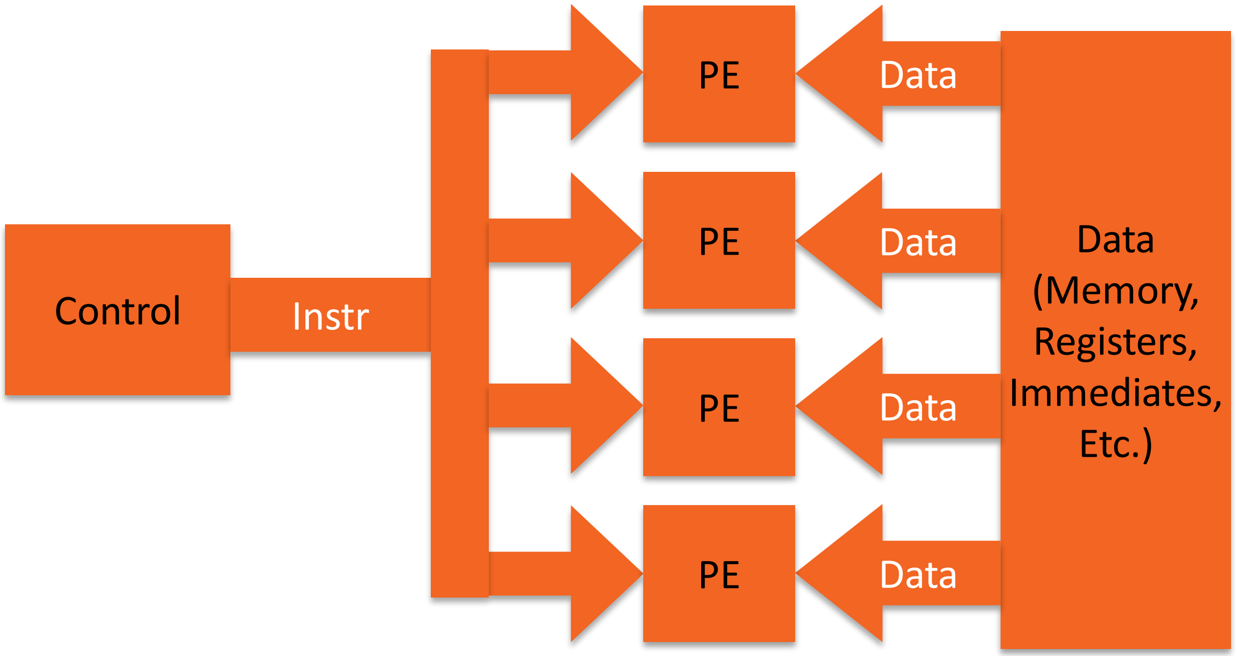
Massively Multi-threaded (GPU)

```
// tid is the thread id
C[tid] = A[tid] + B[tid]
```



PARALLEL HARDWARE – SIMD

- Each processing element of a Single Instruction Multiple Data (SIMD) processor executes the same instruction with different data at the same time
  - A single instruction is issued to be executed simultaneously on many ALU units
  - We say that the number of ALU units is the *width* of the SIMD unit
- SIMD processors are efficient for data parallel algorithms
  - They reduce the amount of control flow and instruction hardware in favor of ALU hardware



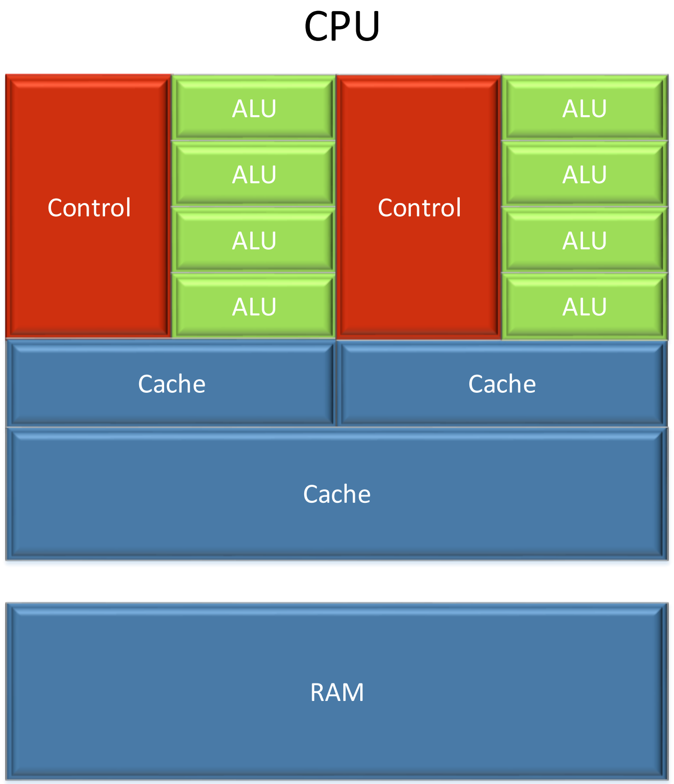
PARALLEL HARDWARE – SIMD

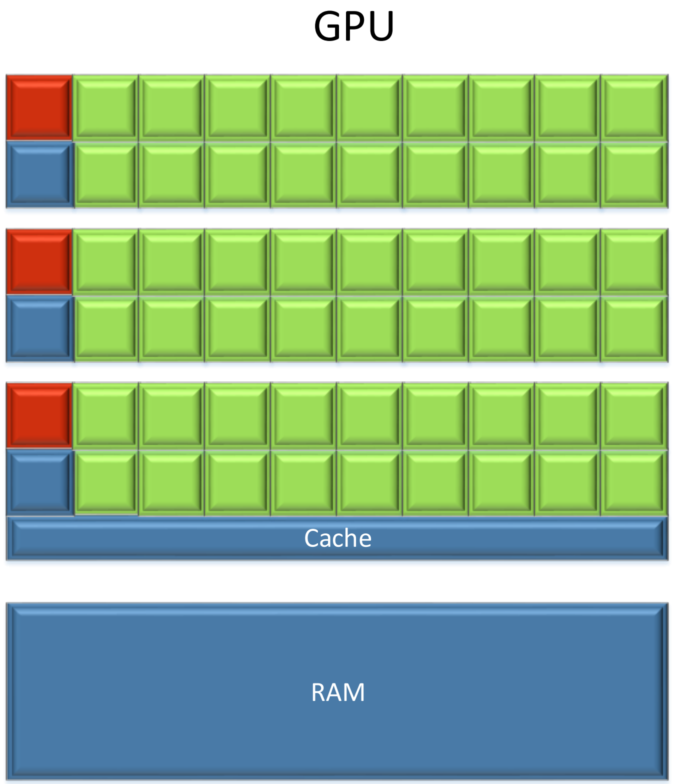
- In the vector addition example, a SIMD unit with a width of four could execute four iterations of the loop at once
- Relating to the apple-picking example, a worker picking apples with both hands would be analogous to a SIMD unit of width 2
- All current GPUs are based on SIMD hardware
  - The GPU hardware implicitly maps each SPMD thread to a SIMD "core"
    - The programmer does not need to consider the SIMD hardware for correctness, just for performance
  - This model of running threads on SIMD hardware is referred to as Single Instruction Multiple Threads (SIMT)

CHALLENGES OF PARALLELIZATION

- On CPUs, hardware-supported atomic operations are used to enable concurrency
  - Atomic operations allow data to be read and written without intervention from another thread
- Some GPUs support system-wide atomic operations, but with a large performance trade-off
  - Usually code that requires global synchronization is not well suited for GPUs (or should be restructured)
  - Any problem that is decomposed using input data partitioning (i.e., requires results to be combined at the end) will likely need to be restructured to execute well on a GPU

PHILOSOPHY OF GPU ARCHITECTURE
FROM GENERAL PURPOSE COMPUTING PERSPECTIVE





CPU vs GPU: Latency vs Throughput
- CPU/multicore: optimized for latency
- GPU/manycore: optimized for throughput

Heterogeneous computing with GPGPU
- Latency-optimized cores for logic part
- Throughput-optimized cores for compute part

CPU VS. GPU

- Memory access
  - CPU is optimized to memory access latency
    - Take advantage of large amount of cache
  - GPU is optimized to memory access bandwidth
    - Take advantage of “0” overhead thread switching, large amount of computing thread and quick switching hide the memory access latency and keep GPU core busy

- Core
  - CPU has heavy core which is good at complex data structure, branch, pre-fetch, fit for serial code; with SIMD and IPL, CPU cores are also fit for lightweight parallel computing
  - GPU has large number of lightweight core, good at simple data layout, non-branch, fit for massive parallel computing

PARALLEL COMPUTING WITH MANY-CORE GPU DEVICES

- Massive parallel thinking
  - Not 4 threads or 16 threads with SIMD instruction extensions on CPU
  - Image tens of thousands threads are in flight on GPU device with “zero” thread creation and scheduling overhead
  - Enough parallelism is the key to explore the GPU horsepower
    - It’s more important for I/O sensitive algorithm
    - Carefully analysis the data dependency

- Scalability is the consequence to be carefully considered
  - Scalability is an important topic on SMP architecture, it’s more important on many-core GPU devices
  - Consider the overhead of inter-thread communication and atomic operation on GPU devices

- Design architecture-oriented algorithm instead of “text-book” algorithm
  - Like, consider the cycles of computing instruction and cycles of memory access, replace memory access with computing for performance speedup

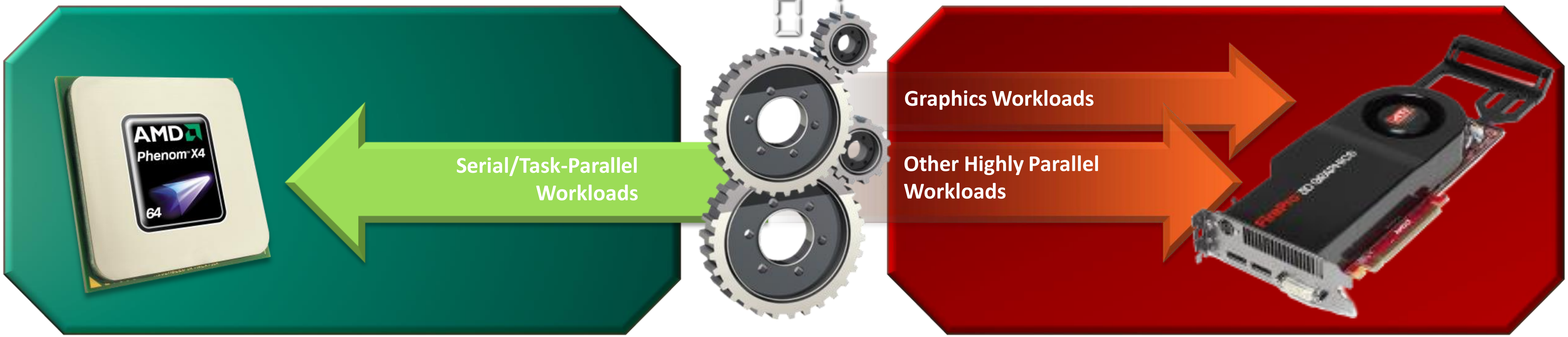
THE NATURE OF CPU+GPU COMPUTING

CPU: scalar processing

+ Latency
+ Optimized for sequential and branching algorithms
+ Single core performance
- Throughput

GPU: parallel processing

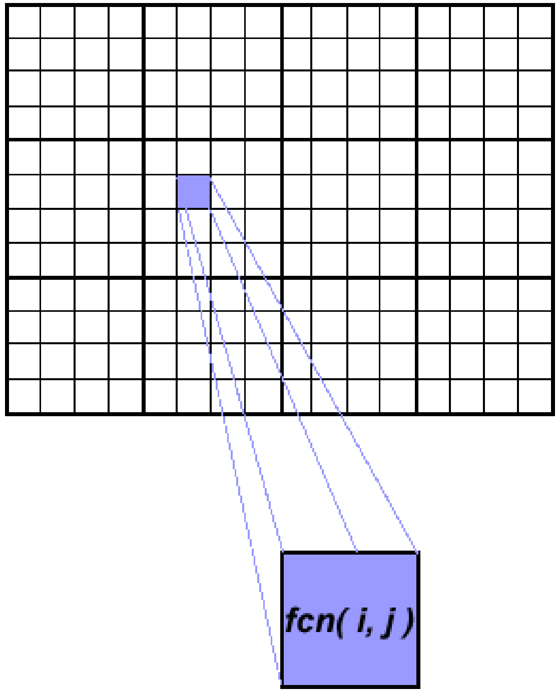
+ Throughput computing
+ High aggregate memory bandwidth
+ Very high overall metal performance/watt
- Latency



CPU+GPU provides optimal performance combinations for a wide range of platform configurations

PARALLEL ALGORITHM DESIGN FOR HETEROGENEOUS PLATFORM

A CASE STUDY – WITH 4 CORE CPU



Step 1: analysis on algorithm and application

Step 2: automatic parallelization or explicit parallelization

Step 3: task/data parallel?

Step 4: parallelism granularity

Step 5: dependency

Step 6: communication

Step 7: load balance

```
do j = 1,n
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

Loop unrolling, partition data set to 4 cores with total 4 threads

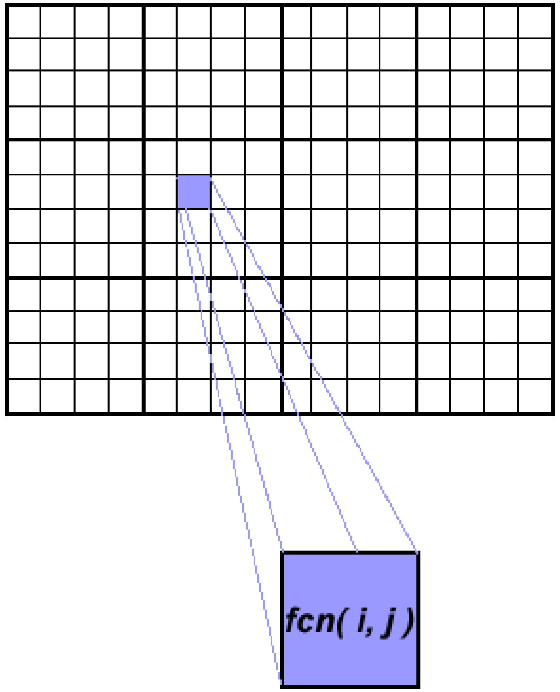
```
do j = 1,n/4
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

……

```
do j = 3n/4+1, n
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

PARALLEL ALGORITHM DESIGN FOR HETEROGENEOUS PLATFORM

A CASE STUDY – OPENMP ON 4 CORE CPU



Step 1: analysis on algorithm and application

Step 2: automatic parallelization or explicit parallelization

Step 3: task/data parallel?

Step 4: parallelism granularity

Step 5: dependency

Step 6: communication

Step 7: load balance

```
do j = 1,n
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

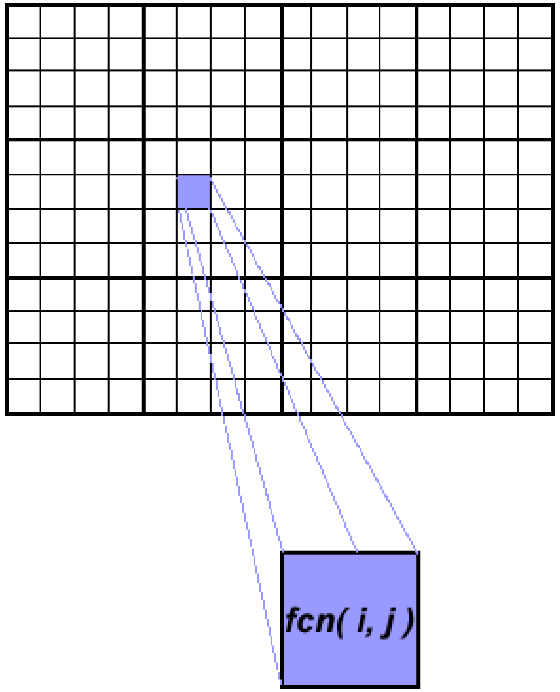
Loop unrolling,
openMP

```
#pragma omp parallel for
do j = 1,n
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

PARALLEL ALGORITHM DESIGN FOR HETEROGENEOUS PLATFORM

A CASE STUDY – MOVE TO MASSIVE PARALLELISM

AMD



Step 1: analysis on algorithm and application

Step 2: automatic parallelization or explicit parallelization

Step 3: task/data parallel?

Step 4: parallelism granularity

Step 5: dependency

Step 6: communication

Step 7: load balance

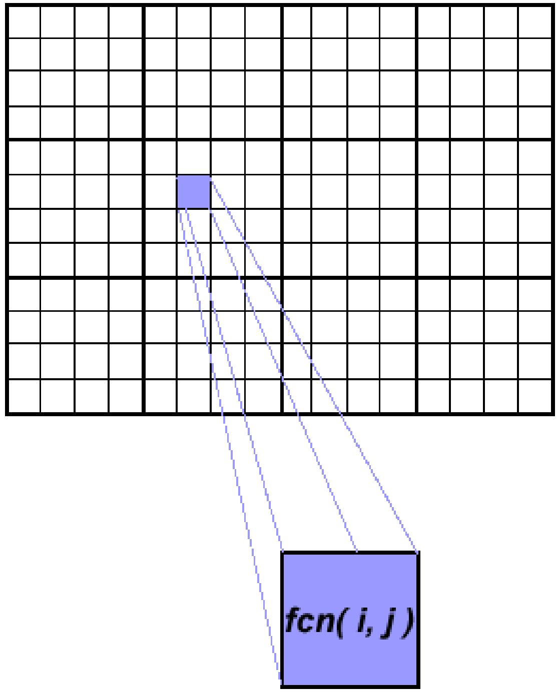
```
do j = 1,n
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

Loop unrolling with very fine-granularity

~~do j = 1, n~~
~~do i = 1, n~~
  uint gidx = get_global_id(0);
  a[gidx] = fcn[gidx];
~~end do~~
~~end do~~

PARALLEL ALGORITHM DESIGN FOR HETEROGENEOUS PLATFORM

A CASE STUDY – WITH GPU



Step 1: analysis on algorithm and application

Step 2: automatic parallelization or explicit parallelization

Step 3: task/data parallel?

Step 4: parallelism granularity

Step 5: dependency

Step 6: communication

Step 7: load balance

```
do j = 1,n
do i = 1,n
   a(i,j) = fcn(i,j)
end do
end do
```

Loop unrolling with very fine-granularity

```
__kernel void fcn() {
   uint gidx = get_global_id( 0 );
   a[gidx] = fcn[gidx];
}
```

OpenCL Kernel

A TYPICAL OPENCL CODE

HOST PART

Start

↓

Initialize devices

↓

Copy data to GPU

↓

Execute GPU Kernel

↓

Copy data back to CPU

↓

Other instructions

```
int main(int argc, char ** argv)
{
  ......

  clGetPlatformIDs(numPlatforms, platforms, NULL);
  clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, numDevices,
              devices, NULL);
  clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);

  clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
             datasize, A, &status);
  clEnqueueWriteBuffer (myqueue , d_ip, CL_TRUE,0, mem_size, (void *)src_image,
             0, NULL, NULL)

  clCreateProgramWithSource(context, 1, (const char**)&source, NULL, &status);
  clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
  clCreateKernel(program, "vecadd", &status);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
  clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize,
                 NULL, 0, NULL, NULL);

  clEnqueueReadBuffer(cmdQueue, d_C, CL_TRUE, 0, datasize, C,
            0, NULL, NULL);

  ......

}
```

A TYPICAL OEPNCL CODES

```
int main(int argc, char ** argv)
{
  ......

  clGetPlatformIDs(numPlatforms, platforms, NULL);
  clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, numDevices,
              devices, NULL);
  clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);

  clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
             datasize, A, &status);
  clEnqueueWriteBuffer (myqueue , d_ip, CL_TRUE,0, mem_size, (void *)src_image,
               0, NULL,  NULL)


  clCreateProgramWithSource(context, 1, (const char**)&source,  NULL, &status);
  clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
  clCreateKernel(program, "vecadd", &status);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
  clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize,
                  NULL, 0, NULL, NULL);

  clEnqueueReadBuffer(cmdQueue, d_C, CL_TRUE, 0, datasize, C,
             0, NULL, NULL);

  ……

}
```

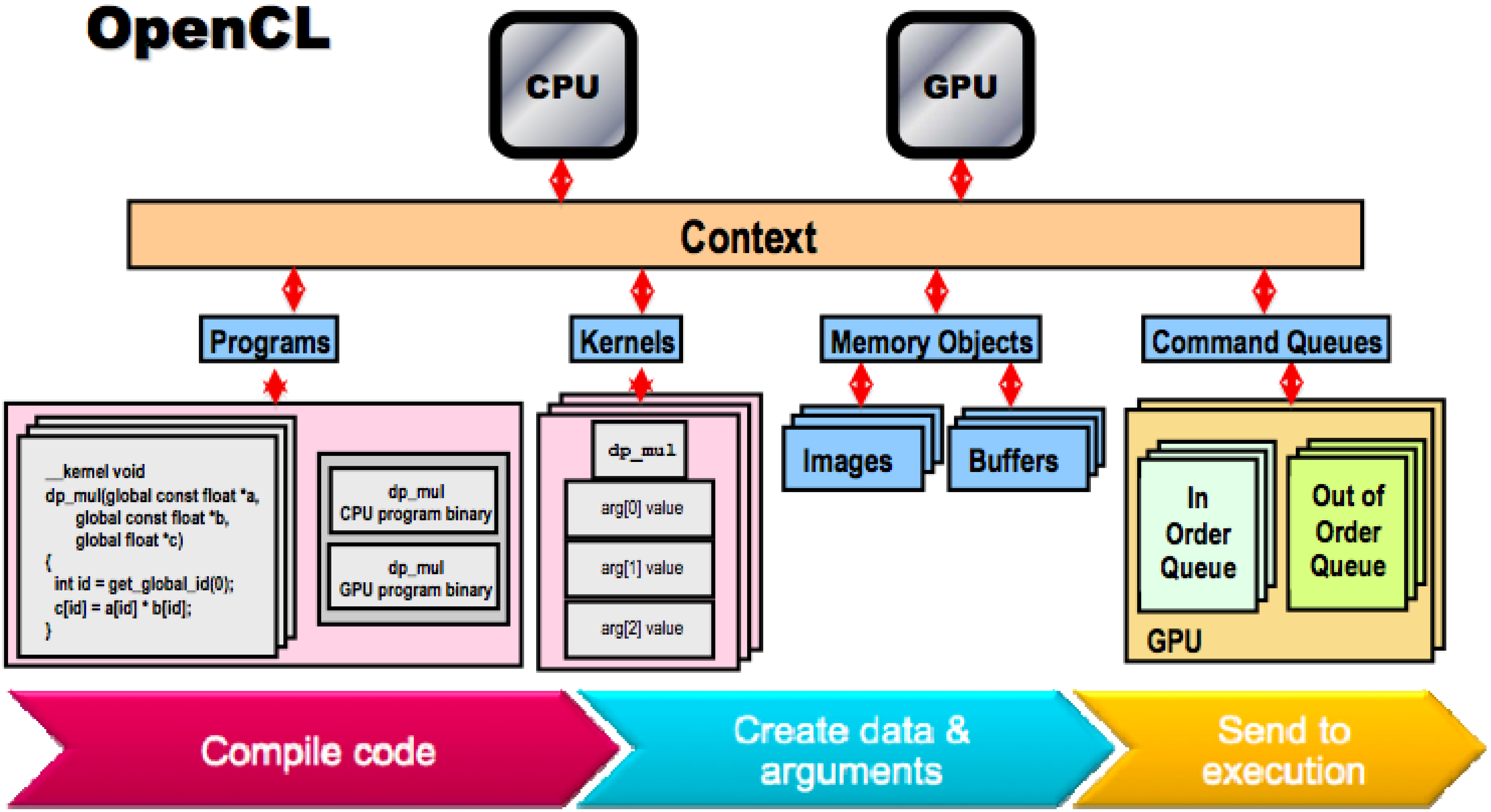
```
__kernel void vecadd(__global int *A,
          __global int *B,
          __global int *C) {

  int idx = get_global_id(0);

  C[idx] = A[idx] + B[idx];

}
```

AGENDA

- What's OpenCL
- Fundamentals for OpenCL programming
- OpenCL programming basics
  - OpenCL architecture and platform
  - OpenCL key components and APIs
- OpenCL programming tools
- Demos

OPENCL ARCHITECTURE BIG PICTURE

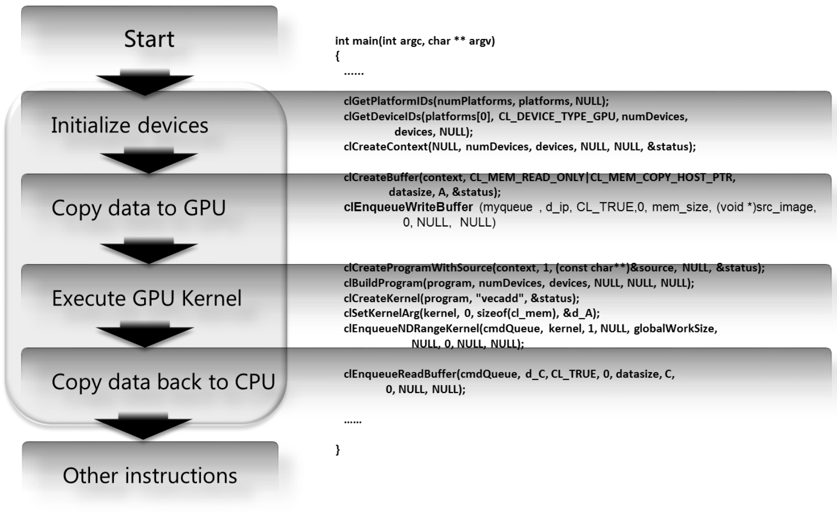
AMD

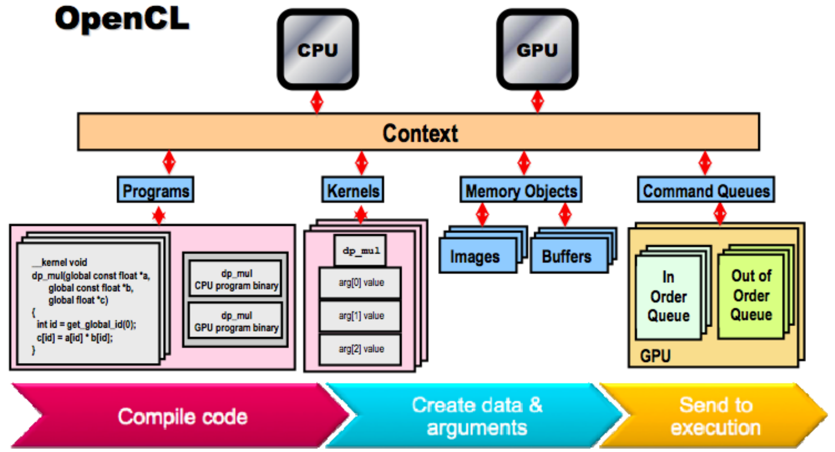


OPENCL ARCHITECTURE OVERVIEW

- OpenCL architecture abstracts the operation into four parts

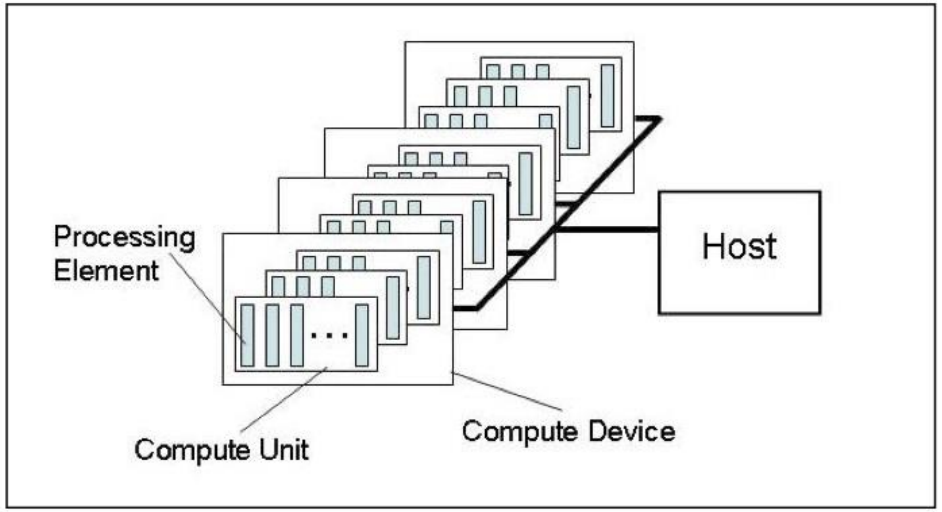
- Platform model
  - Defines the OpenCL devices
- Execution model
  - Defines OpenCL devices actions and inter-actions
- Memory model
  - Defines data location and communications among OpenCL devices
- Programming model
  - Defines how different OpenCL devices working together for a single problem





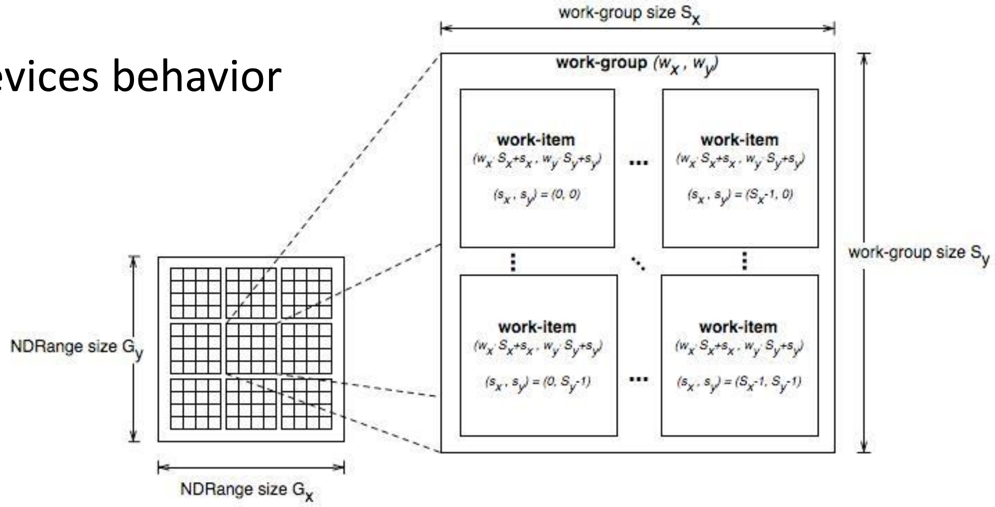
PLATFORM MODEL

- The model consists of a host connected to one or more OpenCL devices
  - A device is divided into one or more compute units
  - Compute units are divided into one or more processing elements
  - Each processing element maintains its own program counter
- The host is whatever the OpenCL library runs on
  - x86 CPUs for GPUs
- Devices are processors that the library can talk to
  - CPUs, GPUs, and generic accelerators
- For AMD
  - All CPUs are combined into a single device (each core is a compute unit and processing element)
  - Each GPU is a separate device
- Every vendor has their implementation of platform model, OpenCL API provides the details of platform information



EXECUTION MODEL

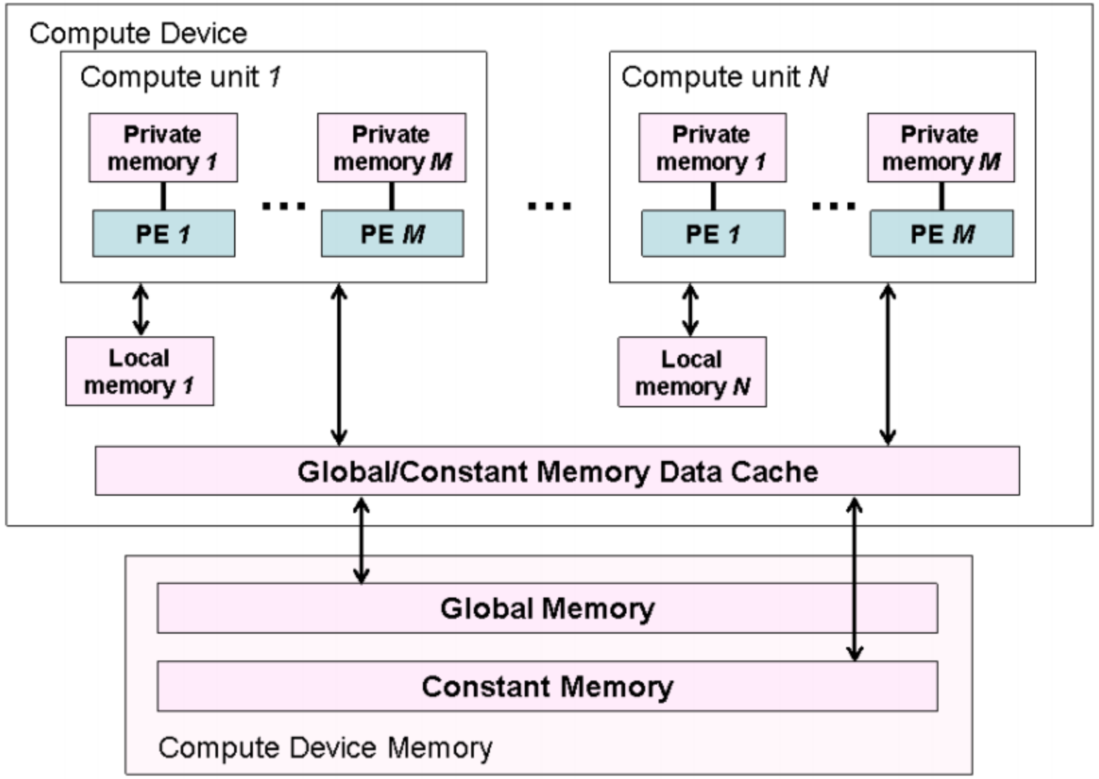
- “Host” program and “Kernel”
  - Host program is just a traditional CPU program consists of all OpenCL components
  - Kernel runs on the OpenCL devices to perform off-loaded computing workloads
- “Context” is defined in host to control Kernel execution
  - “Program” is the object to be JIT compiled into “Kernel” and executed on devices
  - “Memory” is the object as the data communication unit
  - “Command queue” is existing among host and devices to control devices behavior
- Execution model defines how threads are organized in Kernel
  - Each thread is a work-item
  - Work-items are organized as work-group
  - Work-groups are organized as a NDRange



MEMORY MODEL

- Memory model defines
  - The type of memory objects
  - The location of different memory objects
- Two kinds of memory objects
  - Buffer and image

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access



PROGRAMMING MODEL

- The OpenCL execution model supports data parallel and task parallel programming models
- Data parallel programming model
  - One-to-one mapping between work-items and elements in a memory object
  - Work-groups can be defined explicitly or implicitly (specify the number of work-items and OpenCL creates the work-groups)
- Task Parallel Programming Model
  - The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space
  - Under this model, users express parallelism by:
    - Enqueuing multiple tasks, and/or
    - Enqueing native kernels developed using a programming model orthogonal to OpenCL
- Synchronization
  - Possible between items in a work-group
  - Possible between commands in a context command queue

OPENCL FRAMEWORK

AMD

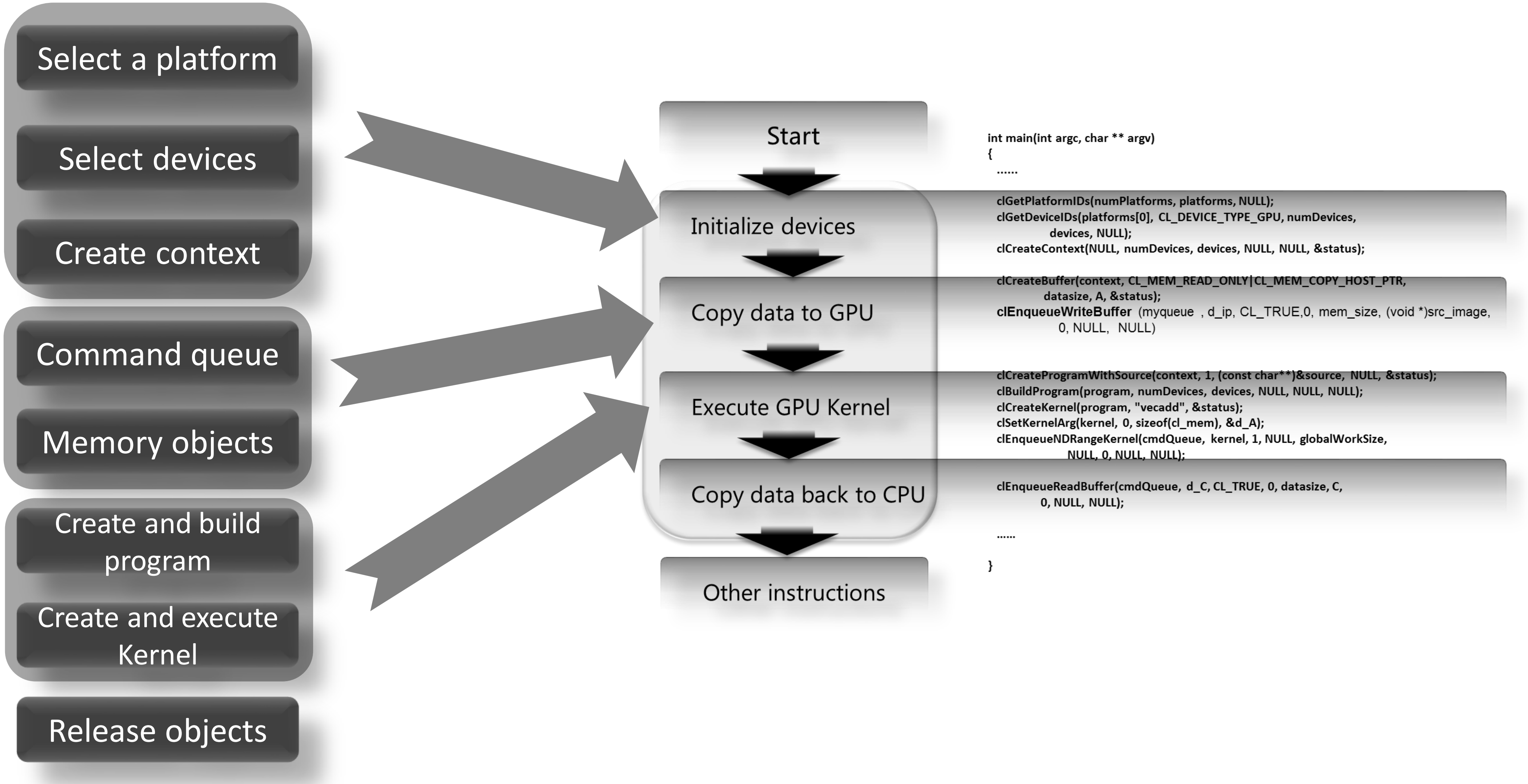
- From API definition point of view, OpenCL framework consists of three parts

- OpenCL Platform layer
  - The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts
  - Platform, device, context

- OpenCL Runtime
  - The runtime allows the host program to manipulate
  - Command queue, memory objects, program, Kernel, kernel execution, event……

- OpenCL Compiler
  - The OpenCL compiler creates program executable that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism. contexts once they have been created

OPENCL PROGRAMMING DIAGRAM



OPENCL PLATFORM LAYER

SELECTING A PLATFORM AND SELECTING DEVICES

```
cl_int clGetPlatformIDs (cl_uint num_entries,
                         cl_platform_id *platforms,
                         cl_uint *num_platforms)
```

```
clGetDeviceIDs4 (cl_platform_id platform,
                 cl_device_type device_type,
                 cl_uint num_entries,
                 cl_device_id *devices,
                 cl_uint *num_devices)
```

- This function is usually called twice
  - The first call is used to get the number of platforms available to the implementation
  - Space is then allocated for the platform objects
  - The second call is used to retrieve the platform objects
- Once a platform is selected, we can then query for the devices that it knows how to interact with
- We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)
- This call is performed twice as with clGetPlatformIDs
  - The first call is to determine the number of devices, the second retrieves the device objects

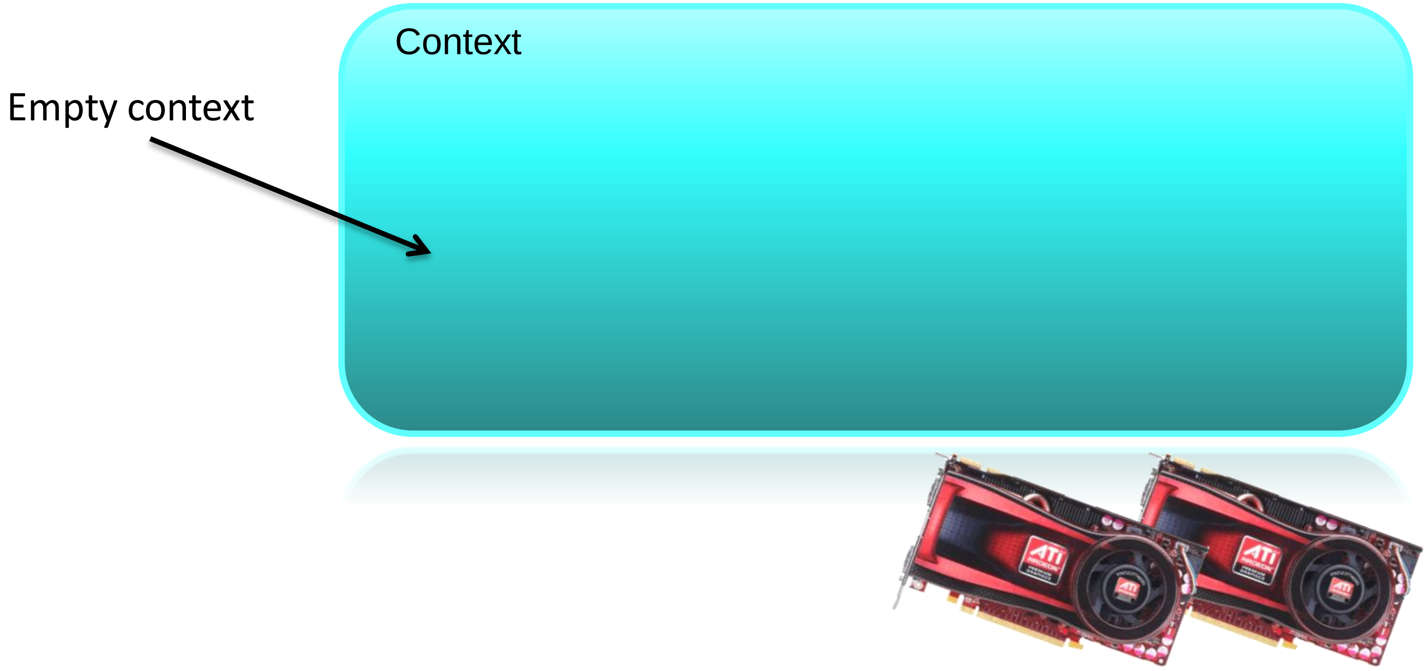
CONTEXT

```
cl_context    clCreateContext (const cl_context_properties *properties,
                               cl_uint num_devices,
                               const cl_device_id *devices,
                               void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                         const void *private_info, size_t cb,
                                         void *user_data),
                               void *user_data,
                               cl_int *errcode_ret)
```

- A context refers to the environment for managing OpenCL objects and resources
- To manage OpenCL programs, the following are associated with a context
  - Devices: the things doing the execution
  - Program objects: the program source that implements the kernels
  - Kernels: functions that run on OpenCL devices
  - Memory objects: data that are operated on by the device
  - Command queues: mechanisms for interaction with the devices
    - Memory commands (data transfers)
    - Kernel execution
    - Synchronization

CREATE CONTEXT

- When you create a context, you will provide a list of devices to associate with it
  - For the rest of the OpenCL resources, you will associate them with the context as they are created



OPENCL RUNTIME - COMMAND QUEUES

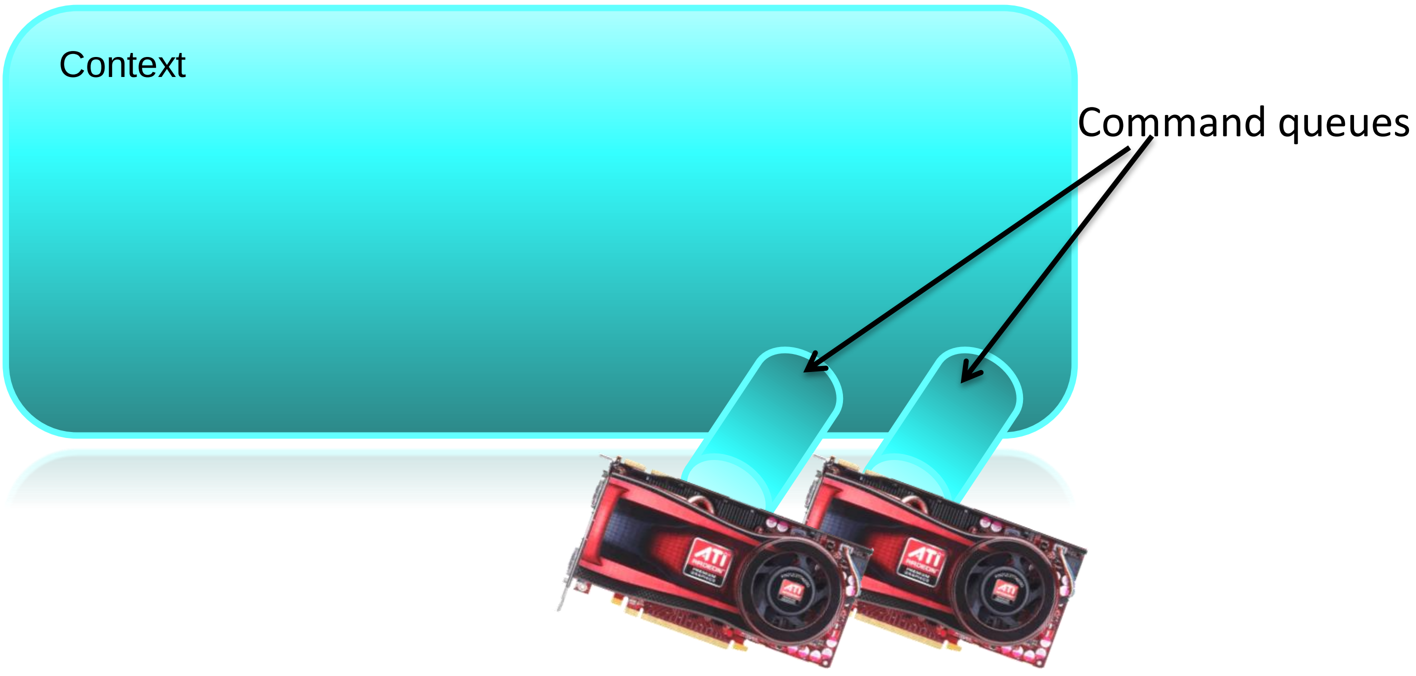
```
cl_command_queue    clCreateCommandQueue (cl_context context,
                                          cl_device_id device,
                                          cl_command_queue_properties properties,
                                          cl_int *errcode_ret)
```

- A *command queue* is the mechanism for the host to request that an action be performed by the device
  - Perform a memory transfer, begin executing, etc.

- A command queue establishes a relationship between a context and a device
- A separate command queue is required for each device
- Commands within the queue can be synchronous or asynchronous

COMMAND QUEUES

- Command queues associate a context with a device
  – Despite the figure below, they are not a physical connection



MEMORY OBJECTS

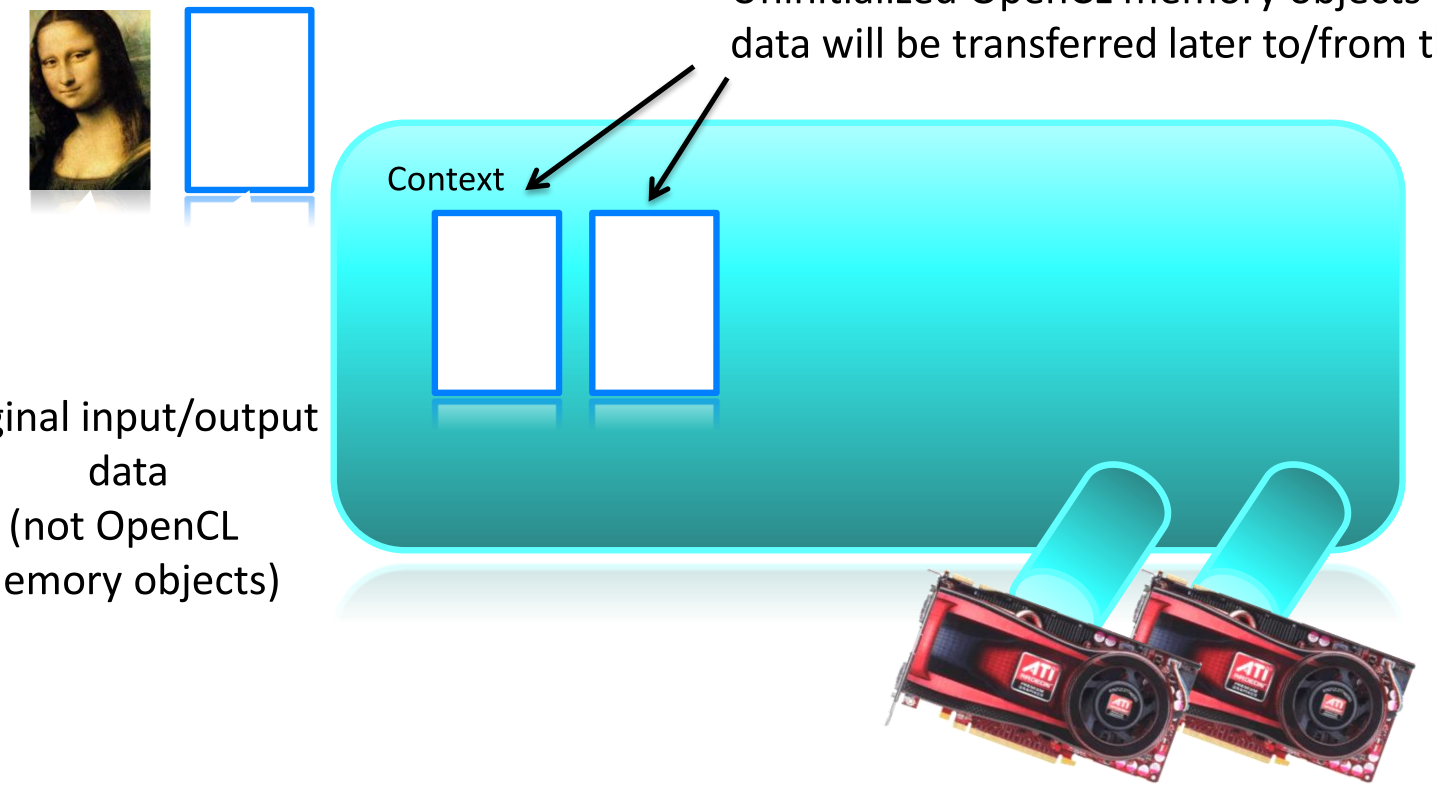
```
cl_mem    clCreateBuffer (cl_context context,
                          cl_mem_flags flags,
                          size_t size,
                          void *host_ptr,
                          cl_int *errcode_ret)
```

- Memory objects are OpenCL data that can be moved on and off devices for the given context
  - Objects are classified as either buffers or images
- Buffers
  - Contiguous chunks of memory – stored sequentially and can be accessed directly (arrays, pointers, structs)
  - Read/write capable
- Images
  - Opaque objects (2D or 3D)
  - Can only be accessed via read_image() and write_image()
  - Can either be read or written in a kernel, but not both

MEMORY OBJECTS

- Memory objects are associated with a context
  - They must be explicitly transferred to devices prior to execution

Uninitialized OpenCL memory objects—the original data will be transferred later to/from these objects



Original input/output data
(not OpenCL memory objects)

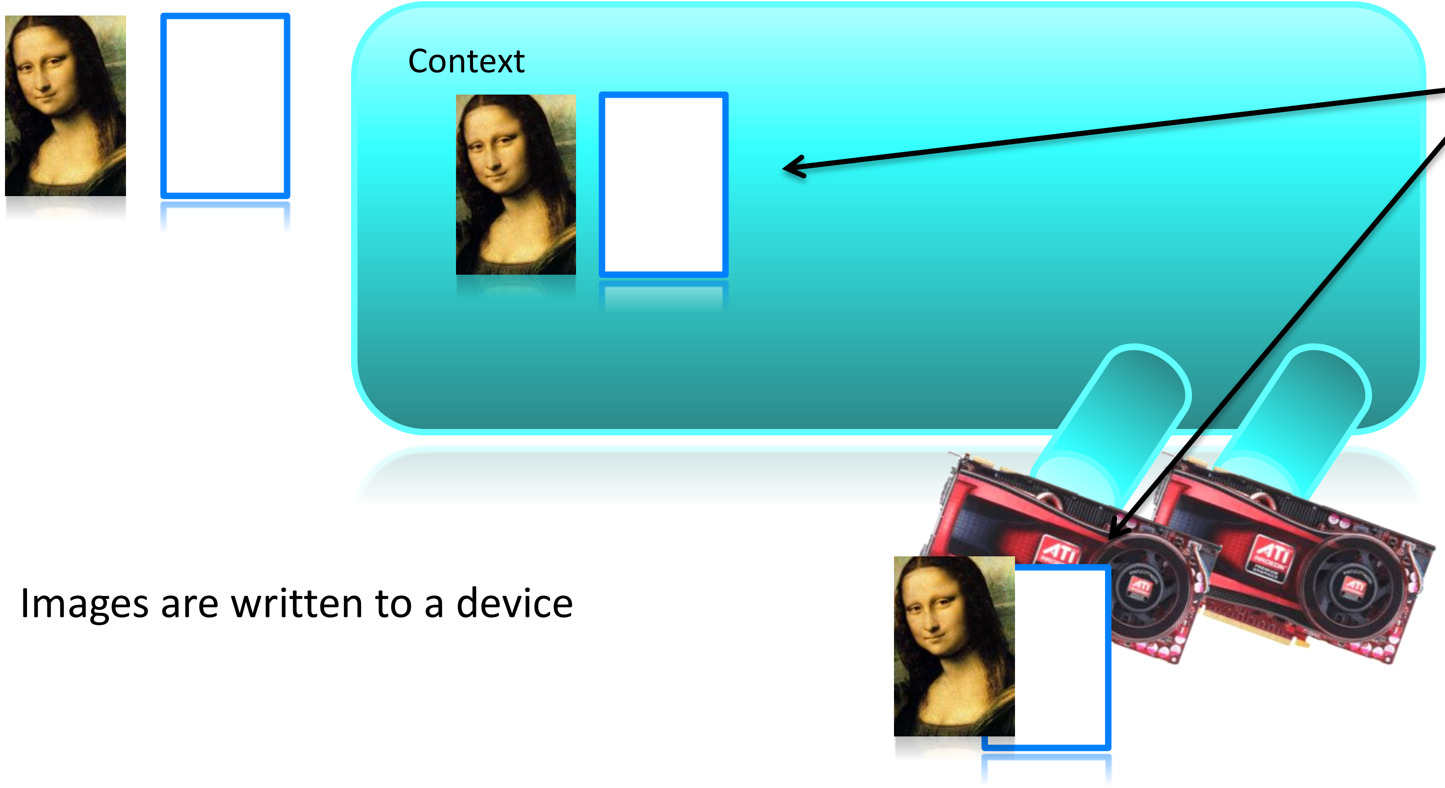
TRANSFERRING DATA

```
cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              const void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

- OpenCL provides commands to transfer data to and from devices
  - clEnqueue{Read|Write}{Buffer|Image}
  - Copying from the host to a device is considered *writing*
  - Copying from a device to the host is *reading*
- The write command both initializes the memory object with data and places it on a device
  - The validity of memory objects that are present on multiple devices is undefined by the OpenCL spec (i.e. are vendor specific)
- OpenCL calls also exist to directly map part of a memory object to a host pointer

TRANSFERRING DATA

- Memory objects are transferred to devices by specifying an action (read or write) and a command queue
  - The validity of memory objects that are present on multiple devices is undefined by the OpenCL spec (i.e. is vendor specific)



The images are redundant here to show that they are both part of the context (on the host) and physically on the device

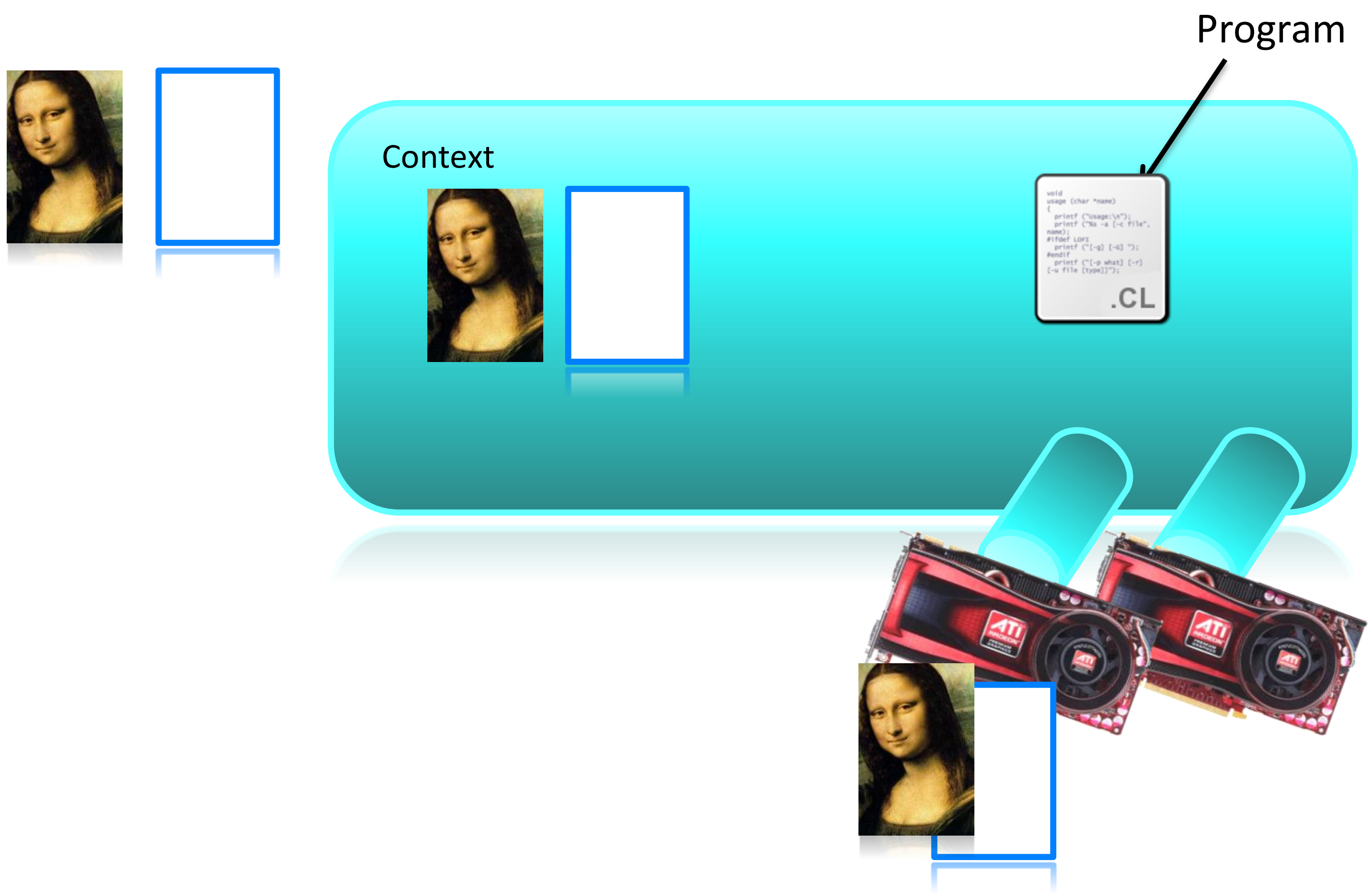
Images are written to a device

PROGRAMS

- A program object is basically a collection of OpenCL kernels
  - Can be source code (text) or precompiled binary
  - Can also contain constant data and auxiliary functions
- Creating a program object requires either reading in a string (source code) or a precompiled binary
- To compile the program
  - Specify which devices are targeted
    - Program is compiled for each device
  - Pass in compiler flags (optional)
  - Check for compilation errors (optional, output to screen)

PROGRAMS

- A program object is created and compiled by providing source code or a binary file and selecting which devices to target



CREATING PROGRAMS AND COMPILING PROGRAMS

```
cl_program   clCreateProgramWithSource (cl_context context,
                                        cl_uint count,
                                        const char **strings,
                                        const size_t *lengths,
                                        cl_int *errcode_ret)
```

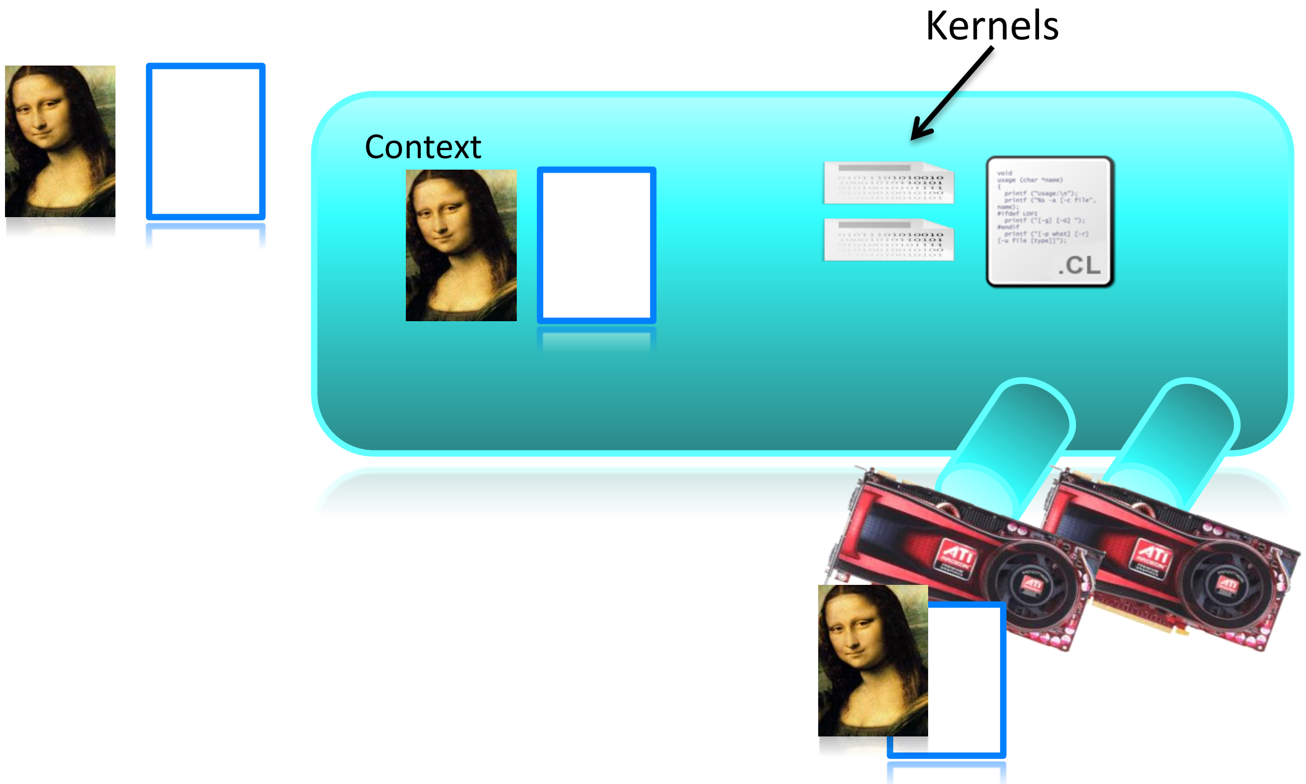
```
cl_int   clBuildProgram (cl_program program,
                         cl_uint num_devices,
                         const cl_device_id *device_list,
                         const char *options,
                         void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                        void *user_data),
                         void *user_data)
```

- The program object is created from strings of source code, JIT capability
- The program object also can be created from a compiled executable binary

- If a program fails to compile, OpenCL requires the programmer to explicitly ask for compiler output
  - A compilation failure is determined by an 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

KERNELS

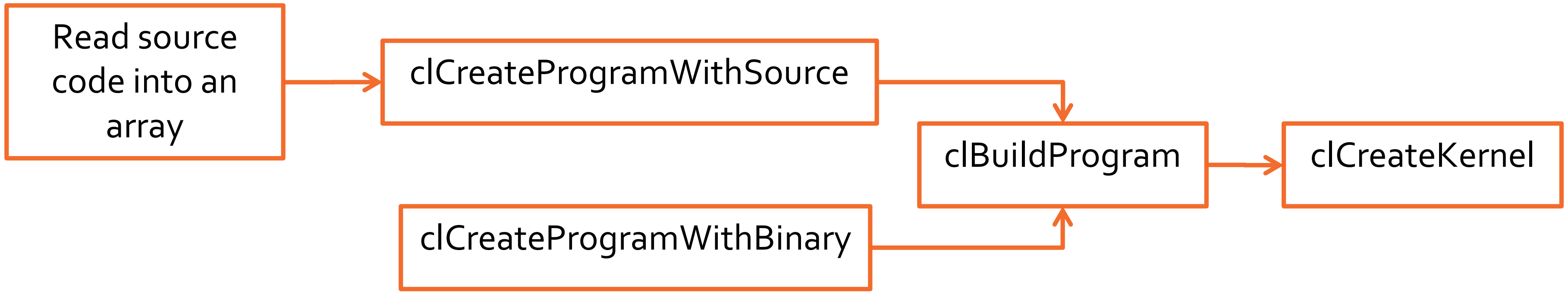
- A kernel is a function declared in a program that is executed on an OpenCL device
  - A kernel object is a kernel function along with its associated arguments
  - Kernel objects are created from a program object by specifying the name of the kernel function
- Must explicitly associate arguments (memory objects, primitives, etc) with the kernel object



KERNELS

```
cl_kernel      clCreateKernel (cl_program program,
                               const char *kernel_name,
                               cl_int *errcode_ret)
```

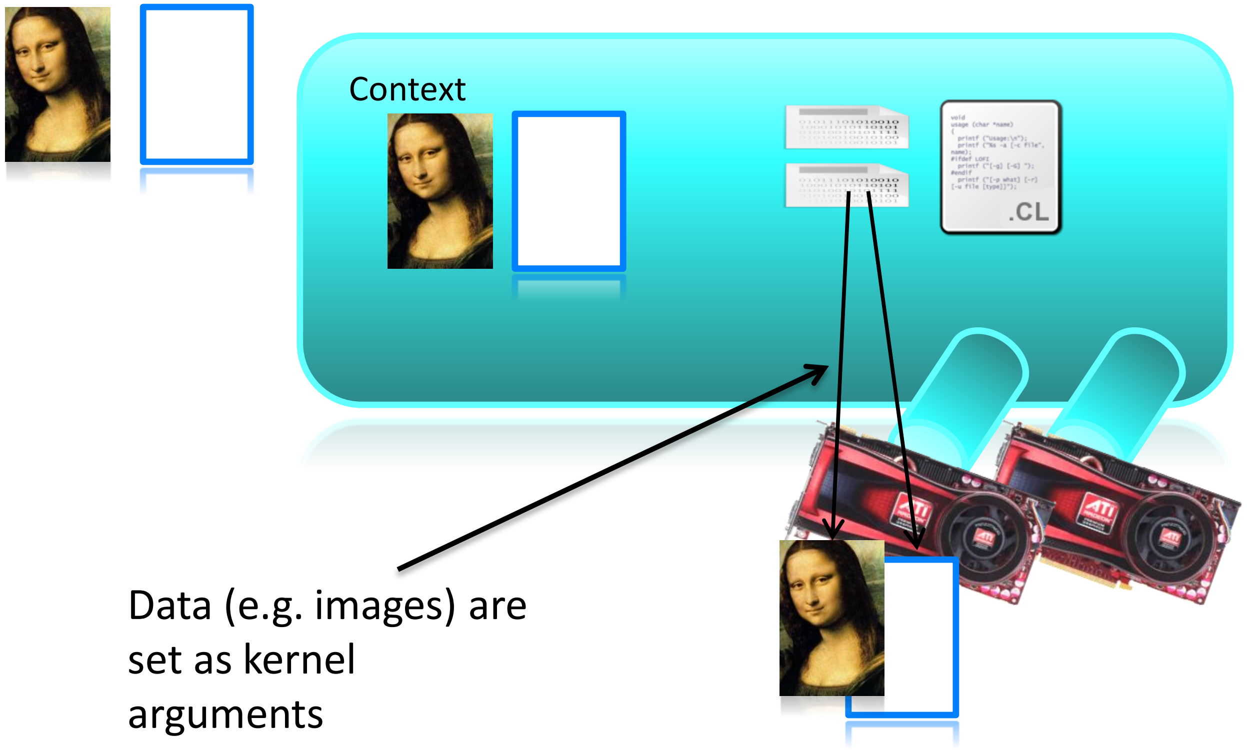
- There is a high overhead for compiling programs and creating kernels
  - Each operation only has to be performed once (at the beginning of the program)
    - The kernel objects can be reused any number of times by setting different arguments



SETTING KERNEL ARGUMENTS

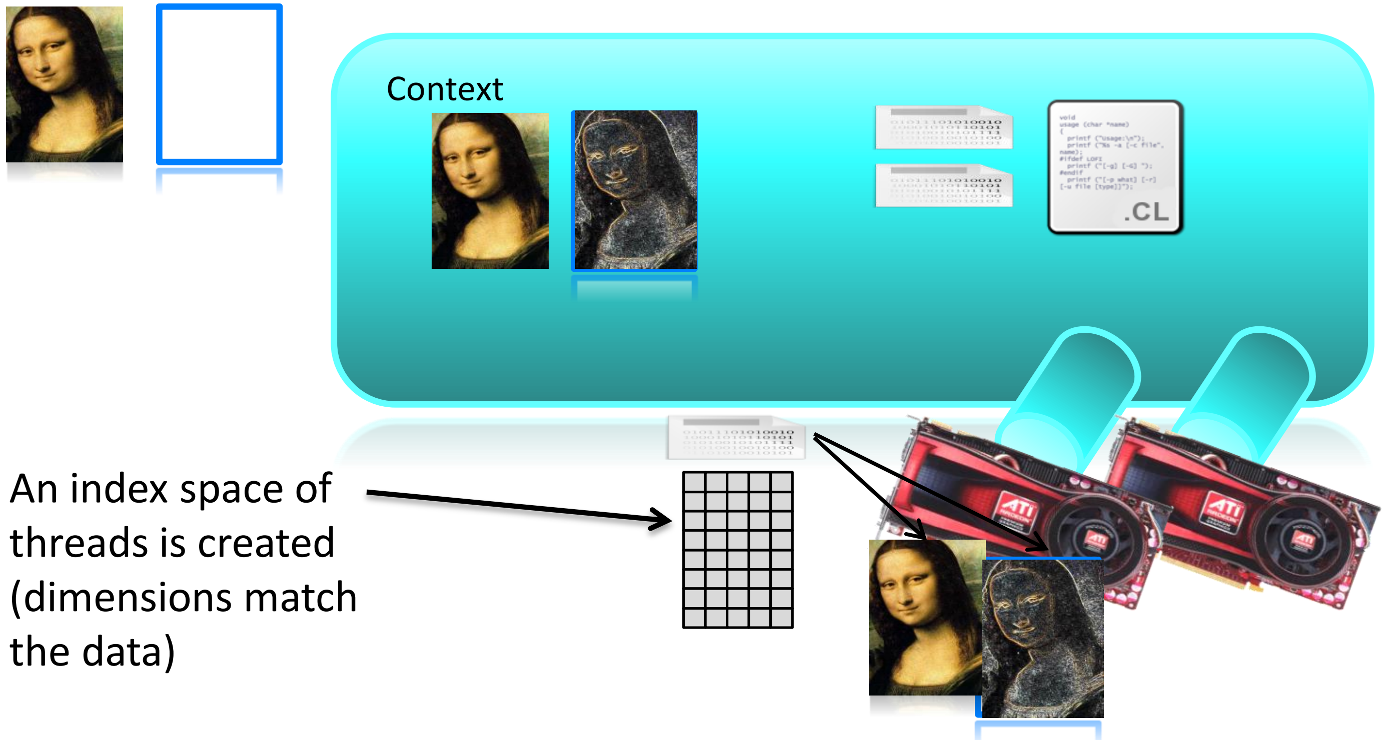
```
cl_int  clSetKernelArg (cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

- Kernel arguments are set by repeated calls to clSetKernelArgs
- Memory objects and individual data values can be set as kernel arguments



EXECUTING THE KERNEL

- Need to set the dimensions of the index space, and (optionally) of the work-group sizes
- Kernels execute asynchronously from the host
  - clEnqueueNDRangeKernel just adds is to the queue, but doesn't guarantee that it will start executing
- A thread structure defined by the index-space that is created
  - Each thread executes the same kernel on different data



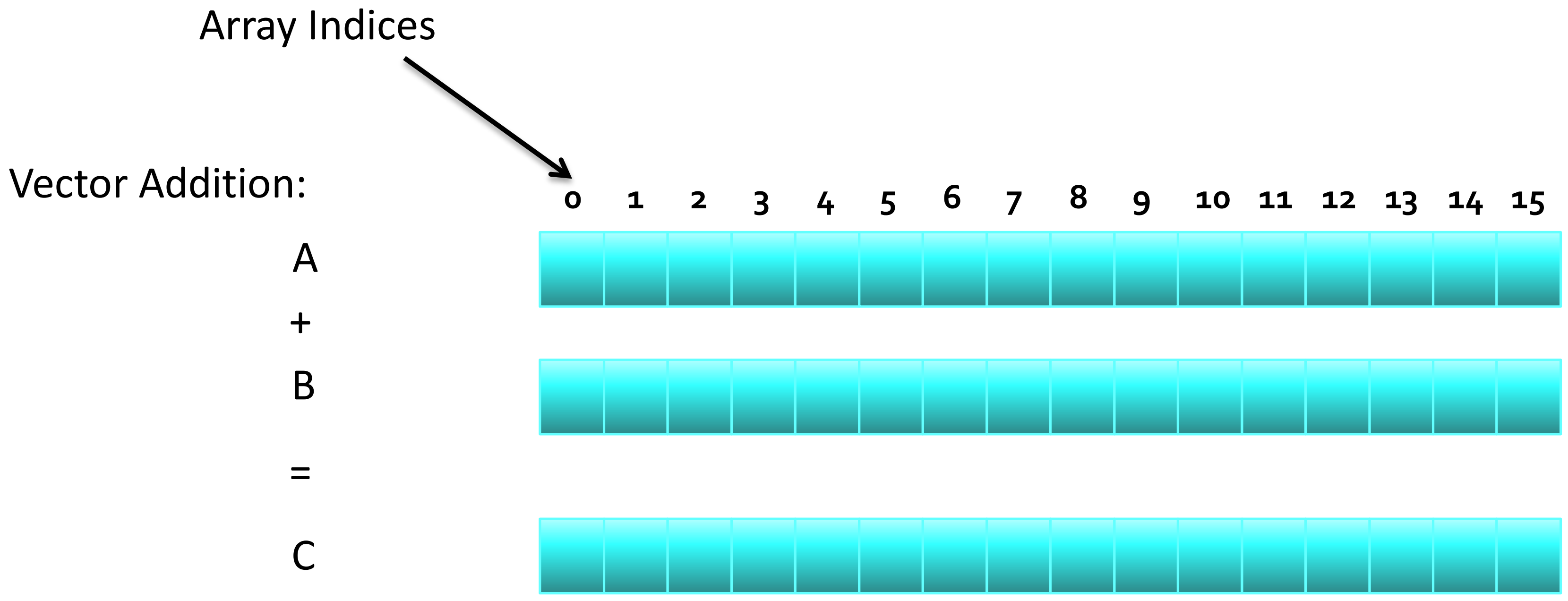
EXECUTING THE KERNEL

```
cl_int    clEnqueueNDRangeKernel (cl_command_queue command_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)
```

- Tells the device associated with a command queue to begin executing the specified kernel
- The global (index space) must be specified and the local (work-group) sizes are optionally specified
- A list of events can be used to specify prerequisite operations that must be complete before executing

THREAD STRUCTURE

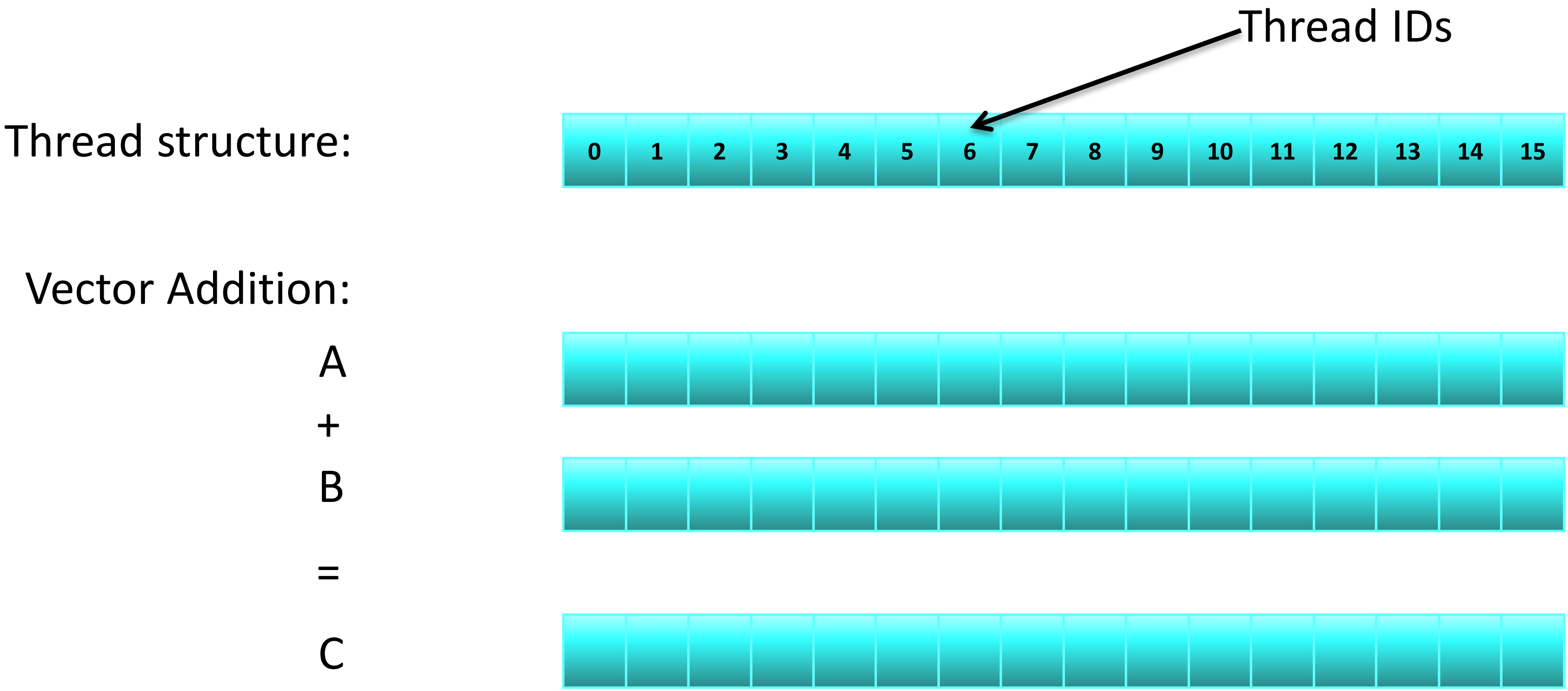
- Massively parallel programs are usually written so that each thread computes one part of a problem
  - For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition
  - If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data
- Consider a simple vector addition of 16 elements
  - 2 input buffers (A, B) and 1 output buffer (C) are required



THREAD STRUCTURE

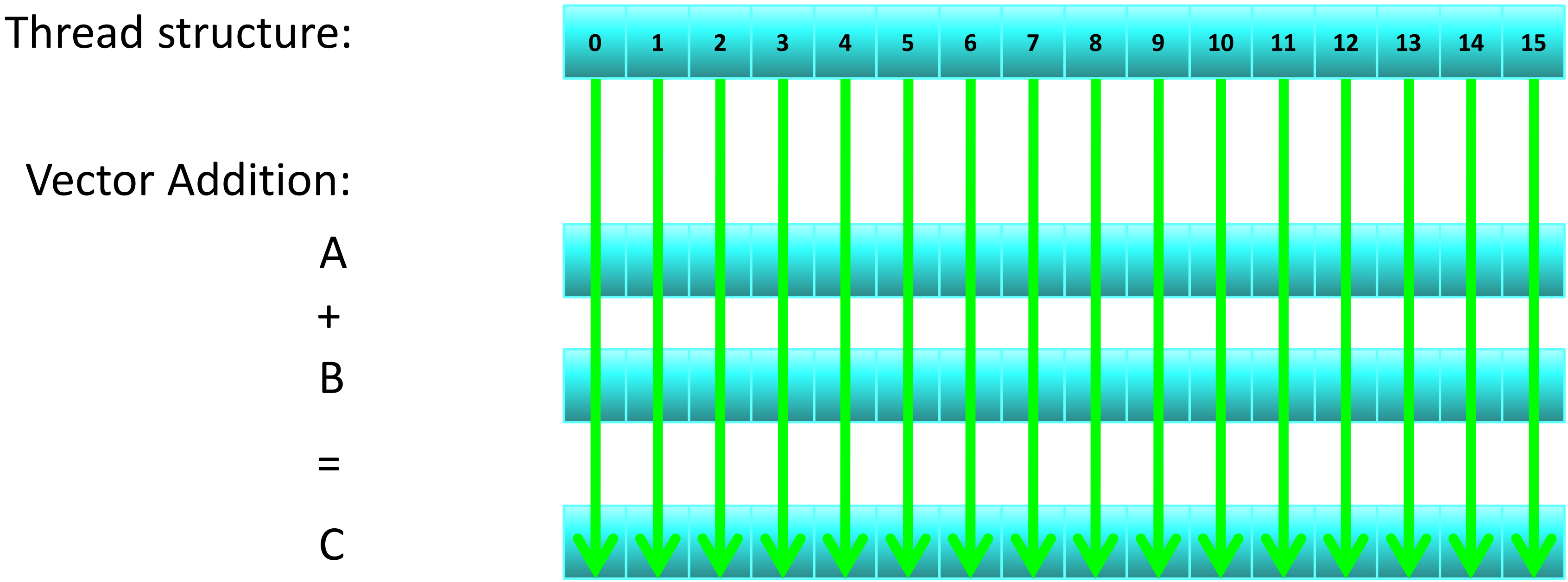
AMD

- Create thread structure to match the problem
  - 1-dimensional problem in this case



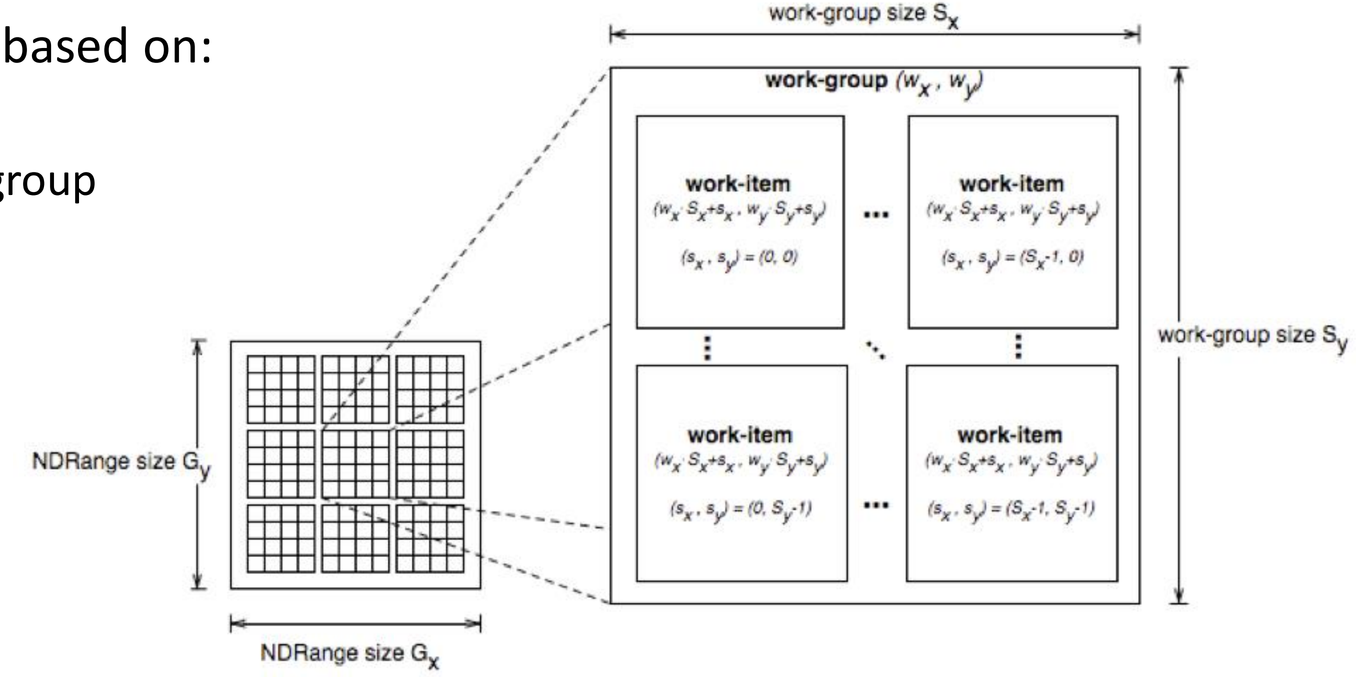
THREAD STRUCTURE

- Each thread is responsible for adding the indices corresponding to its ID



THREAD STRUCTURE

- OpenCL's thread structure is designed to be scalable
- Each instance of a kernel is called a work-item (though "thread" is commonly used as well)
- Work-items are organized as work-groups
  - Work-groups are independent from one-another (this is where scalability comes from)
- An index space defines a hierarchy of work-groups and work-items
- Work-items can uniquely identify themselves based on:
  - A global id (unique within the index space)
  - A work-group ID and a local ID within the work-group



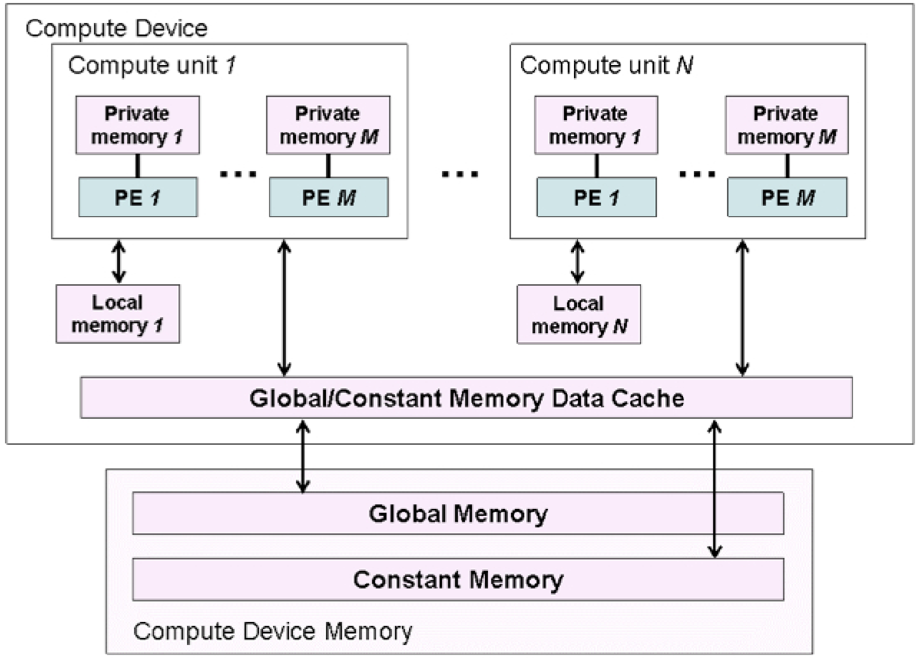
THREAD STRUCTURE

- API calls allow threads to identify themselves and their data
- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
- Or they can determine their work-group ID and ID within the workgroup
  - get_group_id(dim)
  - get_num_groups(dim)
  - get_local_id(dim)
  - get_local_size(dim)

MEMORY MODEL

- The OpenCL memory model defines the various types of memories (closely related to GPU memory hierarchy)
- Memory management is explicit
  - Must move data from host memory to device global memory, from global memory to local memory, and back
- Work-groups are assigned to execute on compute-units
  - No guaranteed communication/coherency between different work-groups (no software mechanism in the OpenCL specification)

Memory	Description
Global	Accessible by all work-items
Constant	Read-only, global
Local	Local to a work-group
Private	Private to a work-item



WRITING A KERNEL

- One instance of the kernel is created for each thread
- Kernels:
  - Must begin with keyword __kernel
  - Must have return type void
  - Must declare the address space of each argument that is a memory object (next slide)
  - Use API calls (such as get_global_id()) to determine which data a thread will work on
- Address Space Identifiers:
  - __global, memory allocated from global address space
  - __constant, a special type of read-only memory
  - __local, memory shared by a work-group
  - __private, private per work-item memory
  - __read_only/__write_only, used for images
- Kernel arguments that are memory objects must be global, local, or constant

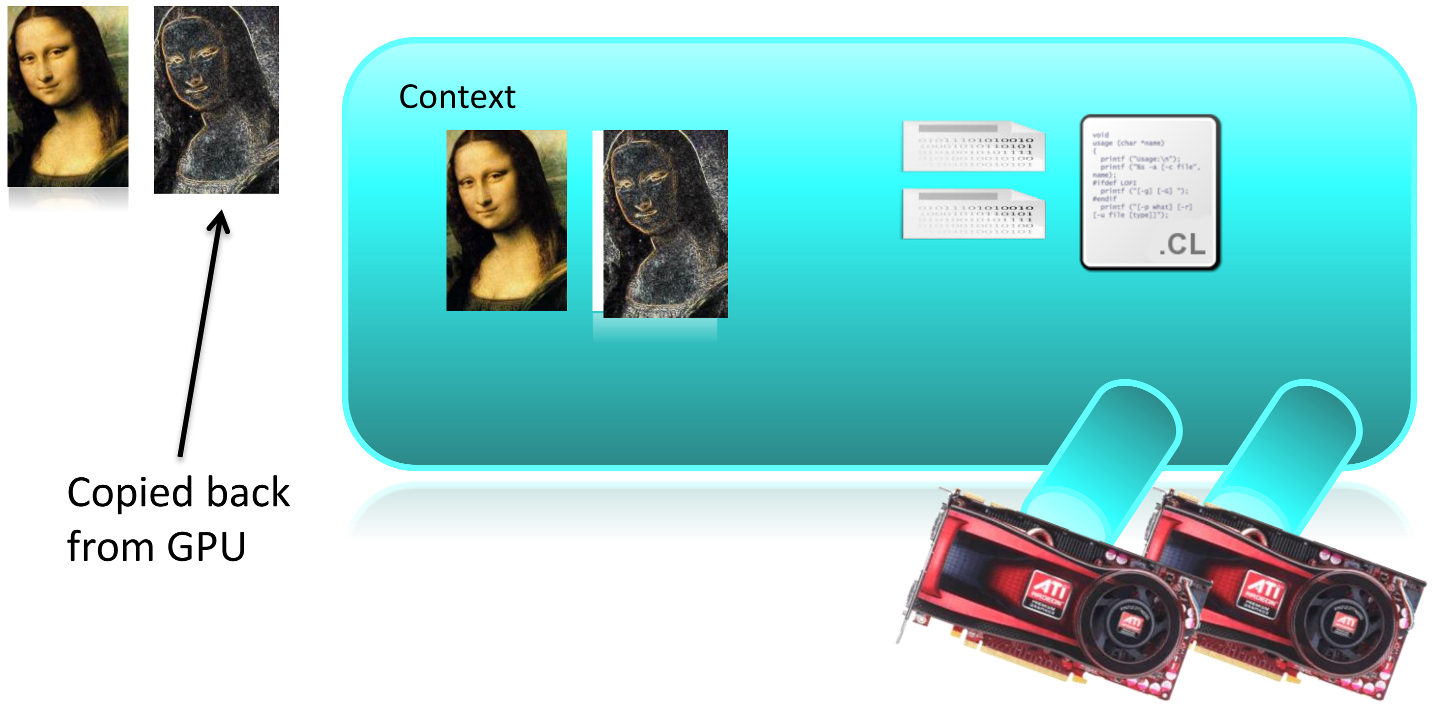
A TYPICAL OEPNCL CODES

```
int main(int argc, char ** argv)
{
  ......

  clGetPlatformIDs(numPlatforms, platforms, NULL);
  clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, numDevices,
              devices, NULL);
  clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);

  clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
             datasize, A, &status);
  clEnqueueWriteBuffer (myqueue , d_ip, CL_TRUE,0, mem_size, (void *)src_image,
               0, NULL,  NULL)


  clCreateProgramWithSource(context, 1, (const char**)&source,  NULL, &status);
  clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
  clCreateKernel(program, "vecadd", &status);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
  clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize,
                  NULL, 0, NULL, NULL);

  clEnqueueReadBuffer(cmdQueue, d_C, CL_TRUE, 0, datasize, C,
            0, NULL, NULL);

  ……

}
```

```
__kernel void vecadd(__global int *A,
           __global int *B,
           __global int *C) {

  int idx = get_global_id(0);

  C[idx] = A[idx] + B[idx];

}
```

COPYING DATA BACK

- The last step is to copy the data back from the device to the host
- Similar call as writing a buffer to a device, but data will be transferred back to the host

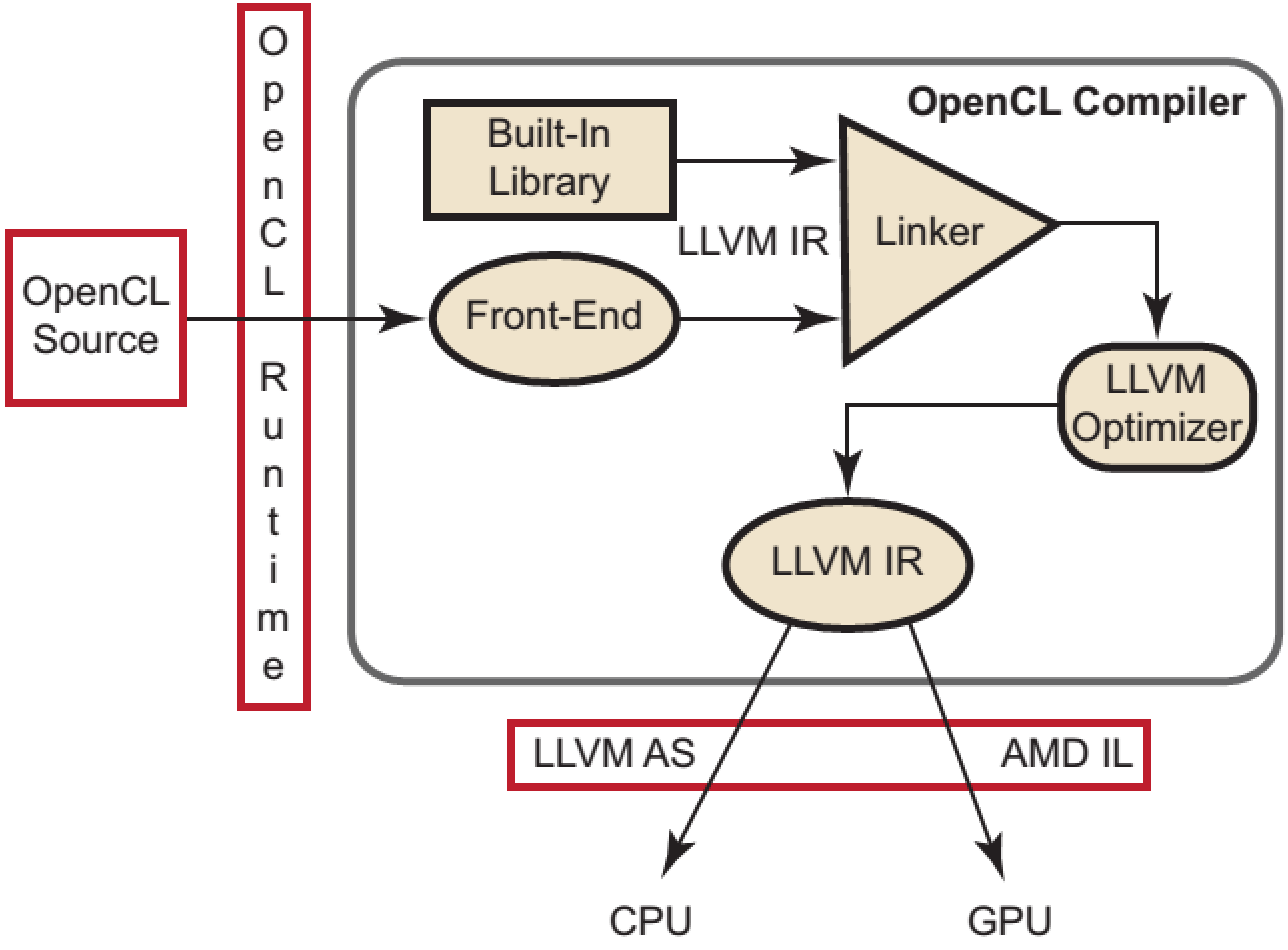


RELEASING RESOURCES

- Most OpenCL resources/objects are pointers that should be freed after they are done being used
- There is a clRelase{Resource} command for most OpenCL types
  - Ex: clReleaseProgram(), clReleaseMemObject()

COMPILING AND RUNNING OPENCL APPLICATION

- Host program is compiled by traditional compiler
  - gcc, MSVC++
- Kernel is compiled by OpenCL compiler
  - Both CPU and GPU computing device shares the same front-end (LLVM extension for OpenCL)
  - LLVM AS generates x86 binary
  - LLVM IR-to-AMD IL generates AMD GPU binary
  - Can be JIT for cross-platform
- Running OpenCL application
  - For CPU as computing device, OpenCL runtime automatically determines the number of processing elements
  - For GPU as computing device, Kernel runs as the exact instructions



AGENDA

- What's OpenCL
- Fundamentals for OpenCL programming
- OpenCL programming basics
- OpenCL programming tools
- Demos

COMPLETE TOOL-CHAIN FOR OEPCNL PROGRAMMING

- AMD APP SDK
  - SDK for OpenCL programming
  - Includes header files, libraries, compiler and sample codes
- AMD CodeXL
  - All-in-one debugger and profiler for OpenCL programming
  - With AMD Kernel Analyzer
    - Static OpenCL Kernel performance analyzer
    - Expose IL and ISA of various GPU platform
- Library
  - Bolt, a C++ template library
  - AMD clAmdBlas, AMD clAmdFFT, Aparapi
  - clMAGMA, OpenCV, etc…….

KERNEL DEBUGGING AND PROFILING

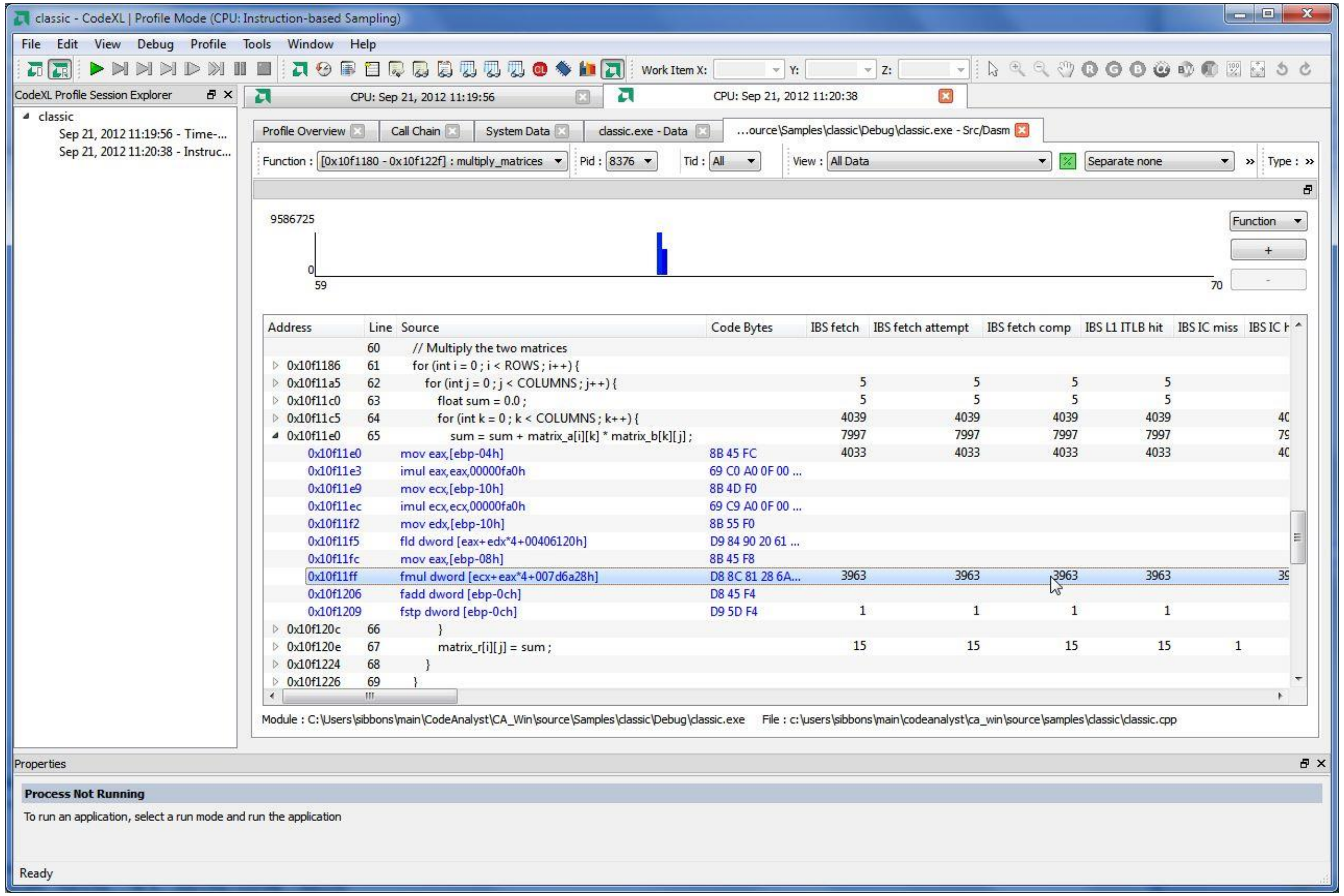
USING CODEXL

- AMD CodeXL is the all-in-one tool for
  - Powerful GPU debugging
  - Comprehensive GPU and CPU profiling
  - Static OpenCL™ kernel analysis capabilities
- AMD CodeXL is available both as a Visual Studio® extension and a standalone user interface application for Windows® and Linux®.

CPU PROFILING KEY FEATURES AND BENEFITS

AMD

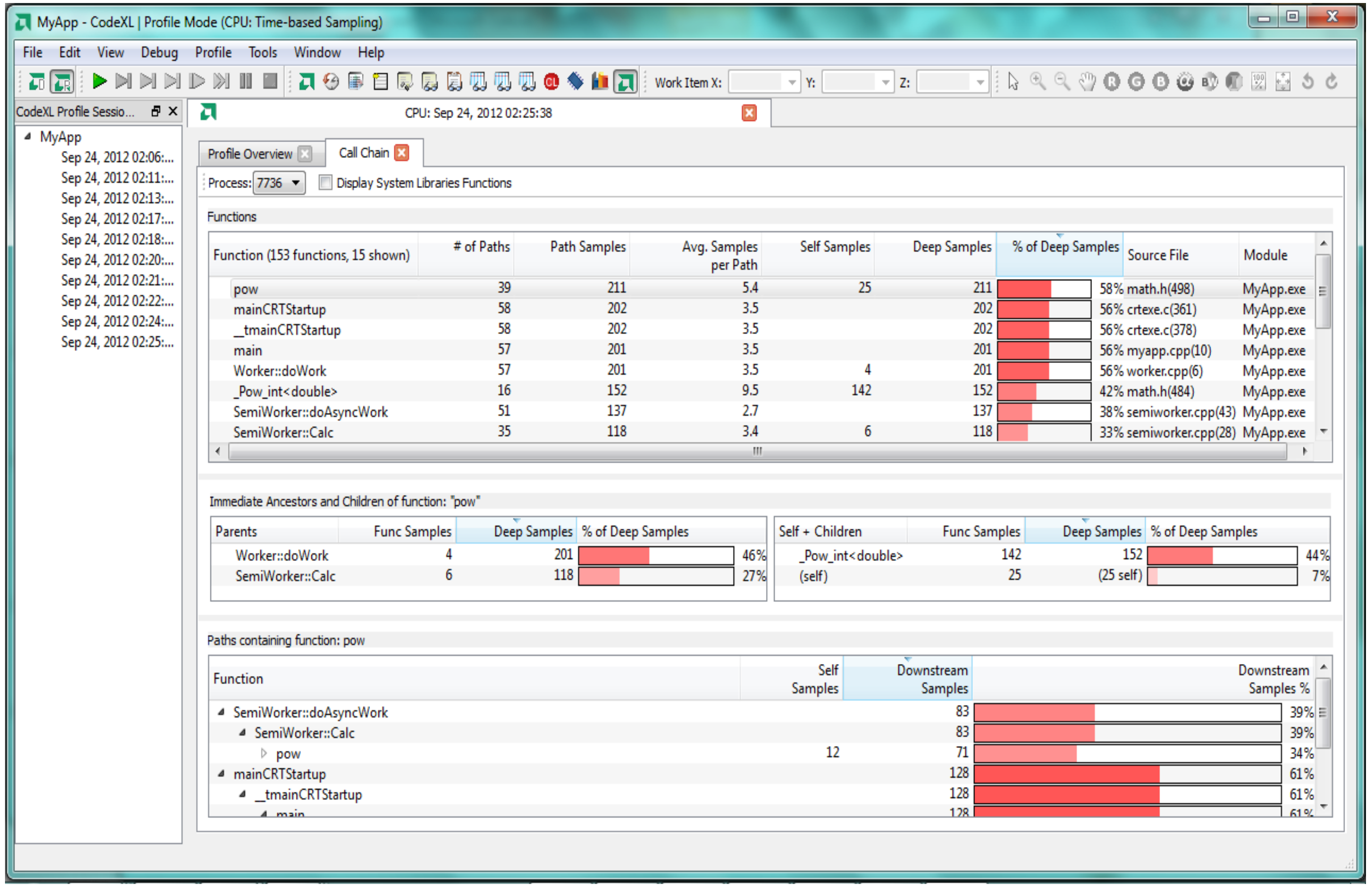
- **Diagnose performance issues in hot-spots**
  - AMD CodeXL uses hardware-level performance counters and instruction-based sampling to provide valuable clues about inefficient program behavior.
  - Use rates and ratios to quickly measure the efficiency of functions, loops and program statements.



CPU PROFILING KEY FEATURES AND BENEFITS

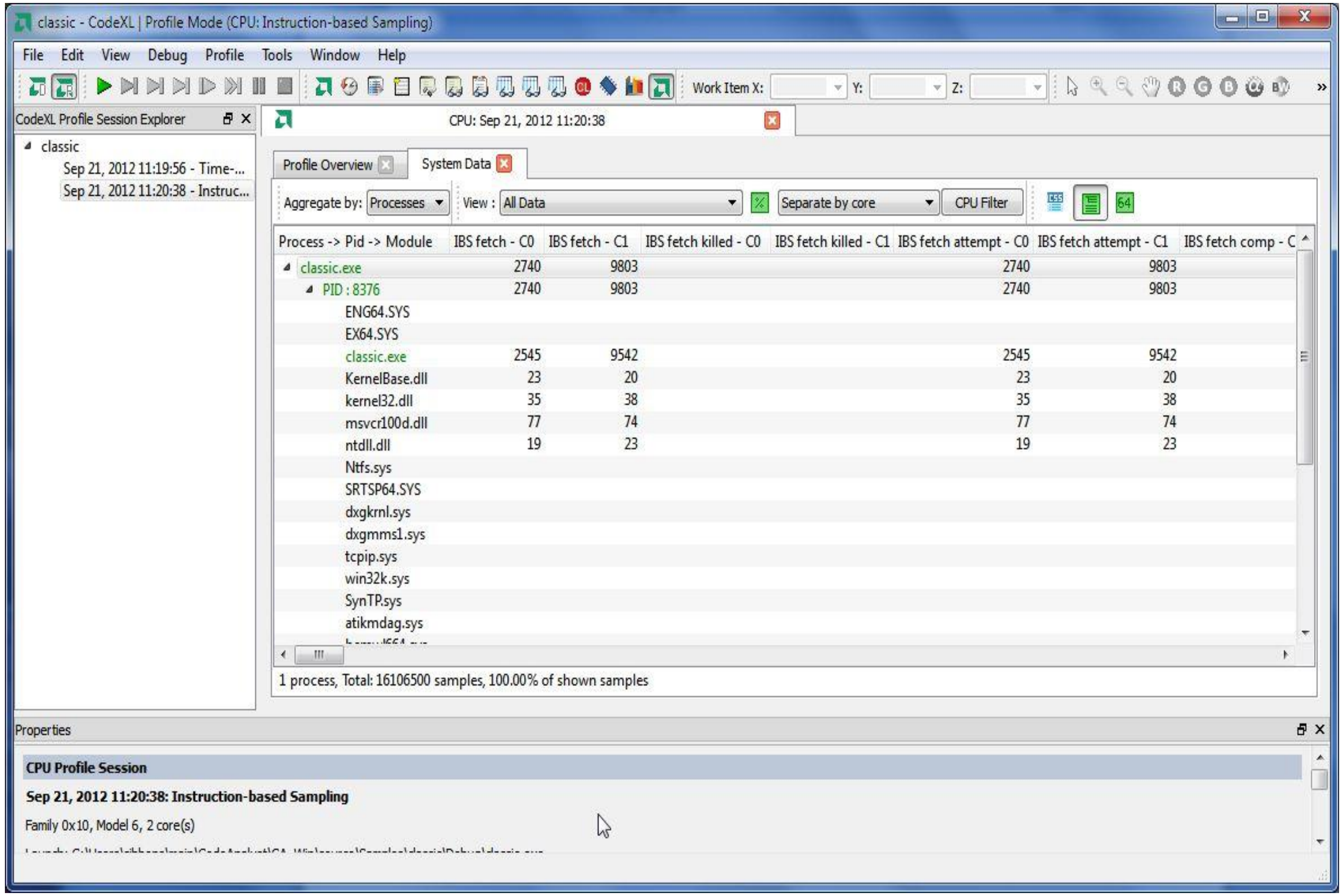
AMD

- **Analyze Call Chain relationships**
  - Diagnose issues from a caller / callee relationship perspective.
  - Quickly determine which call trees are using the most resources (time or events) to isolate potential optimization opportunities.



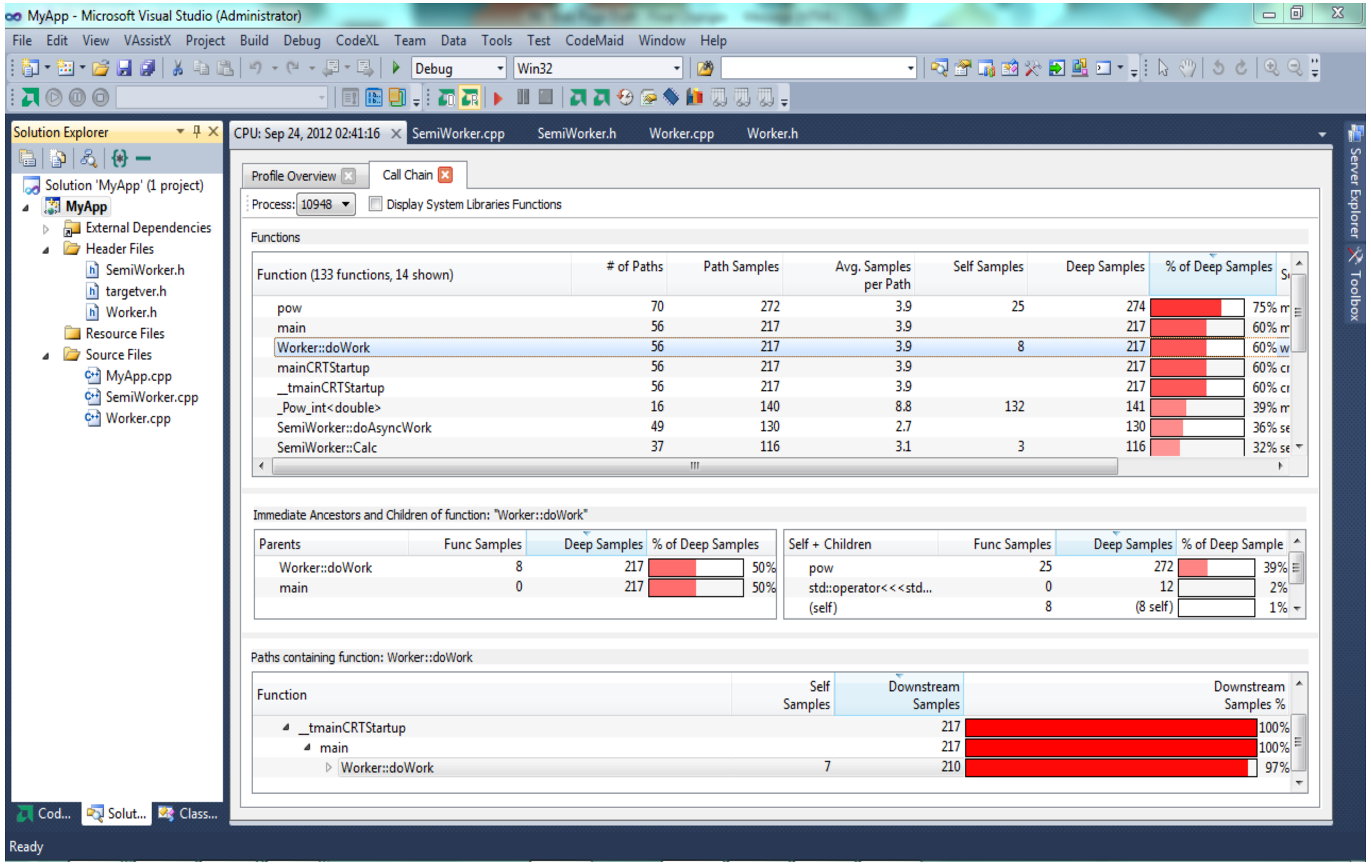
CPU PROFILING KEY FEATURES AND BENEFITS

- **Supports multi-core Windows and Linux platforms**
  – AMD CodeXL supports all of the latest AMD processors on both Windows and Linux platforms.



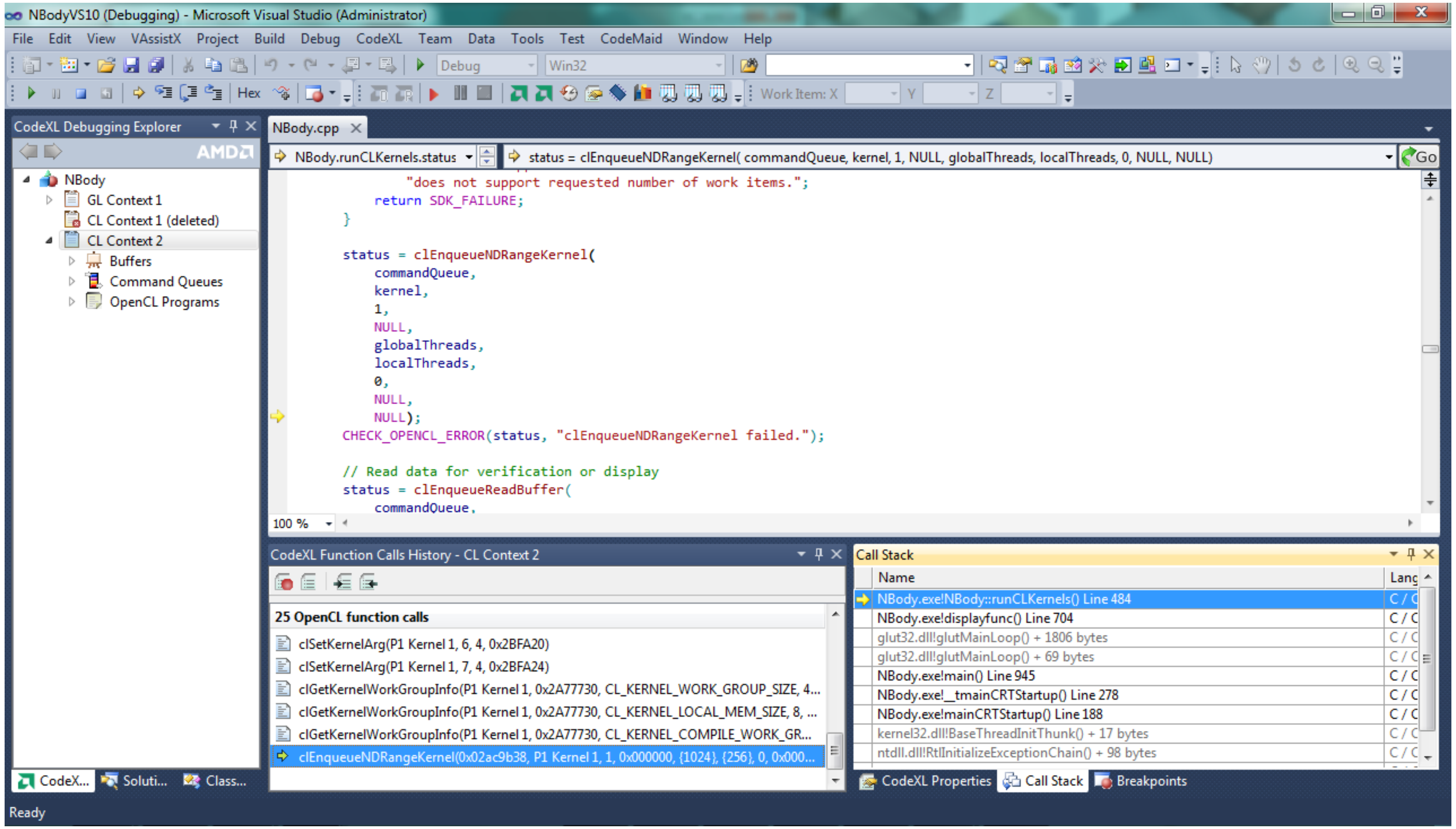
CPU PROFILING KEY FEATURES AND BENEFITS

- **Extends Microsoft Visual Studio**
  - Microsoft Visual Studio user can analyze their programs without leaving the Visual Studio environment.
  - The AMD CodeXL Visual Studio plug-in provides all of the profiling features supported by the stand-alone AMD CodeXL for Windows GUI-based tool.



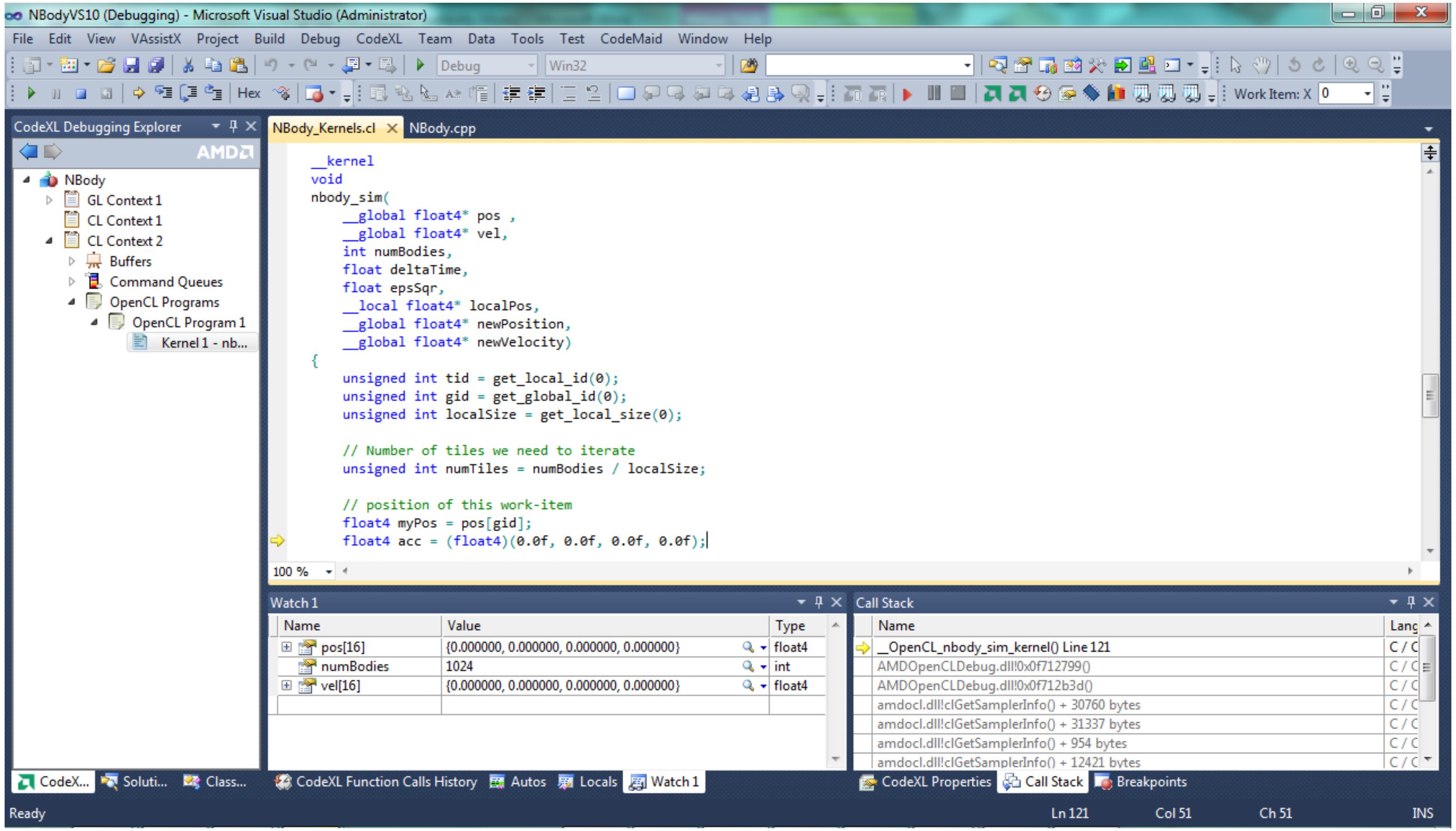
GPU DEBUGGING KEY FEATURES AND BENEFITS

- **Real-time OpenCL and OpenGL API-level debugging**
  - Allows locating API function calls and the code paths that led to them



GPU DEBUGGING KEY FEATURES AND BENEFITS

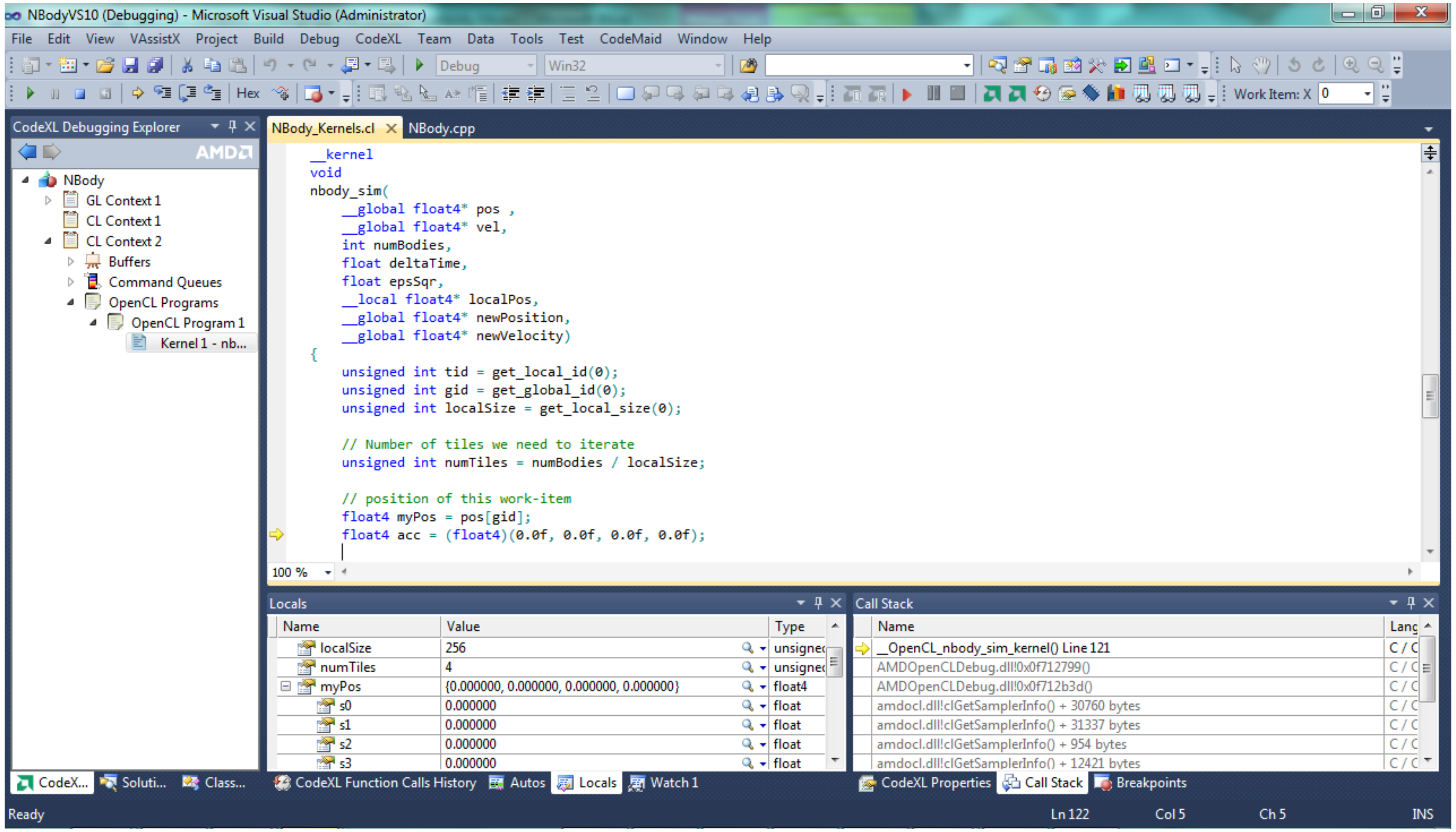
- **Online OpenCL kernel debugging**
  - Works with present hardware. Requires no special configuration or changes to the code. Develop and debug on a single computer with just one GPU. Step through the workflow of a single work item or compare values across all work items.



GPU DEBUGGING KEY FEATURES AND BENEFITS

AMD

- **Full integration with Visual Studio**
  – Now API-level debugging is performed inside the Visual Studio source editor. If OpenCL kernel source code .cl files are included in the project, they will be identified and used for kernel debugging. In addition, Visual Studio views such as the call stack view and locals view will be filled with kernel debugging information.

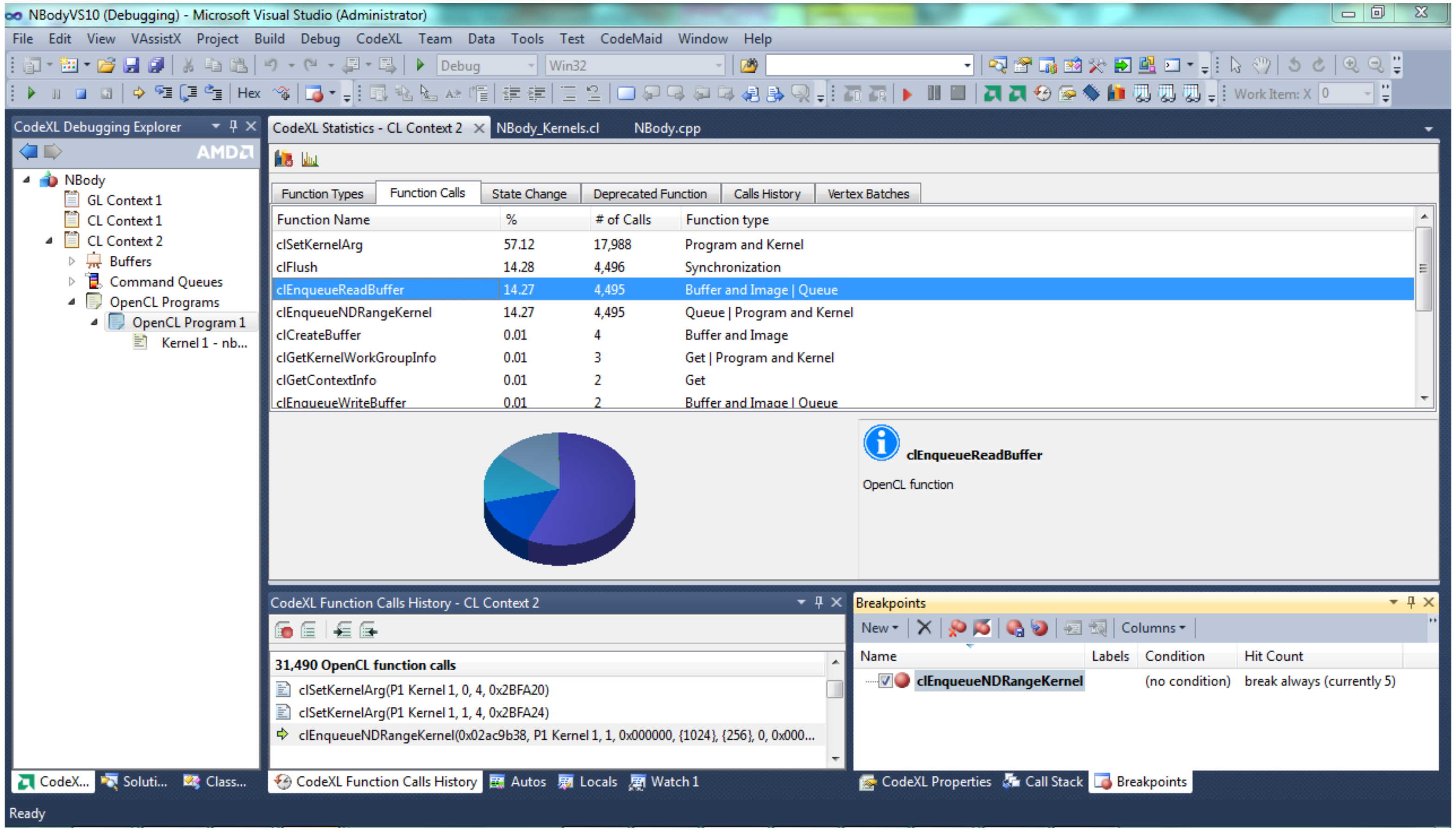


GPU DEBUGGING KEY FEATURES AND BENEFITS

AMD

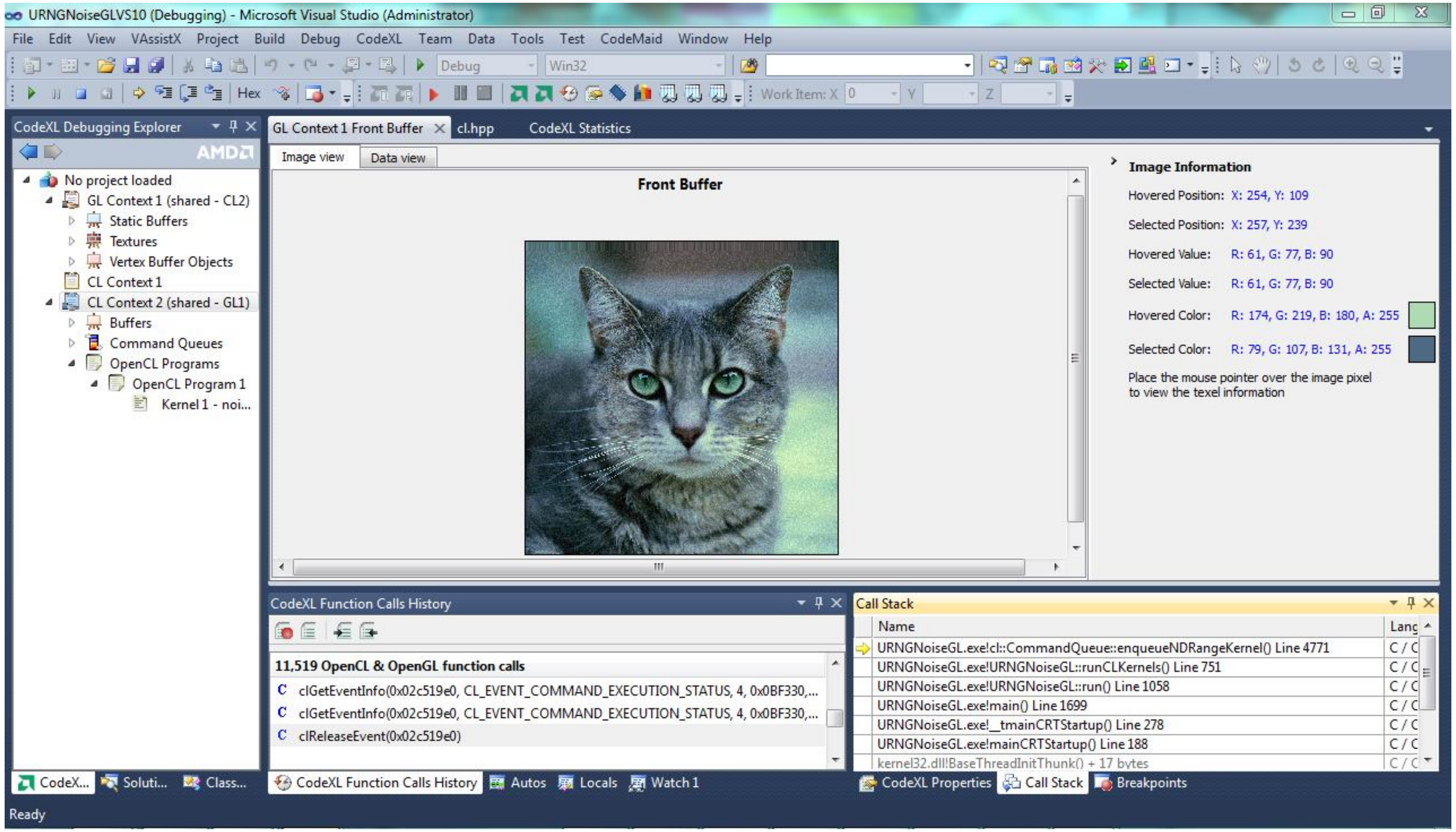
API statistics view

- Gives an overview of OpenCL and OpenGL API usage, and more detailed views, including unrecommended function calls (with alternative suggestions) and deprecated behavior.



GPU DEBUGGING KEY FEATURES AND BENEFITS

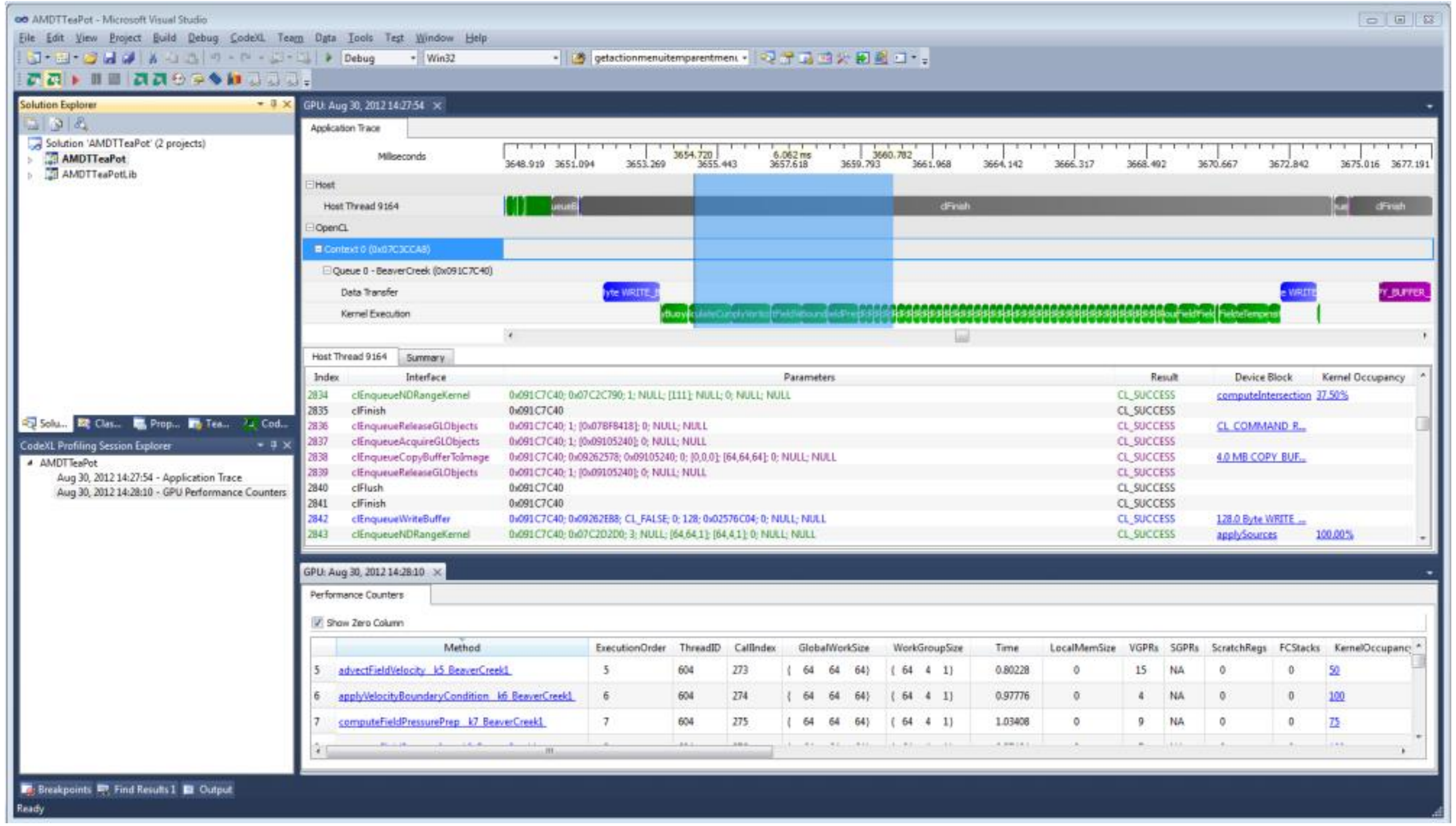
- **Object visualization**
  - View and export OpenCL buffers and Images and OpenGL Textures and buffers as pictures or as spreadsheet data.



GPU PROFILING KEY FEATURES AND BENEFITS

AMD

- **Collect OpenCL™ Application Trace**
  - View and debug the input parameters and output results for all OpenCL™ API calls
  - Search the API calls
  - Navigate to the source code that called an OpenCL™ API
  - Specify which OpenCL™ APIs will be traced



GPU PROFILING KEY FEATURES AND BENEFITS

AMD

- **Collect GPU Performance Counters of AMD Radeon™ graphics cards**
  - Show kernel resource usage
  - Show the number of instructions executed by the GPU
  - Show the GPU utilization
  - Show the GPU memory access characteristics
  - Measure kernel execution time

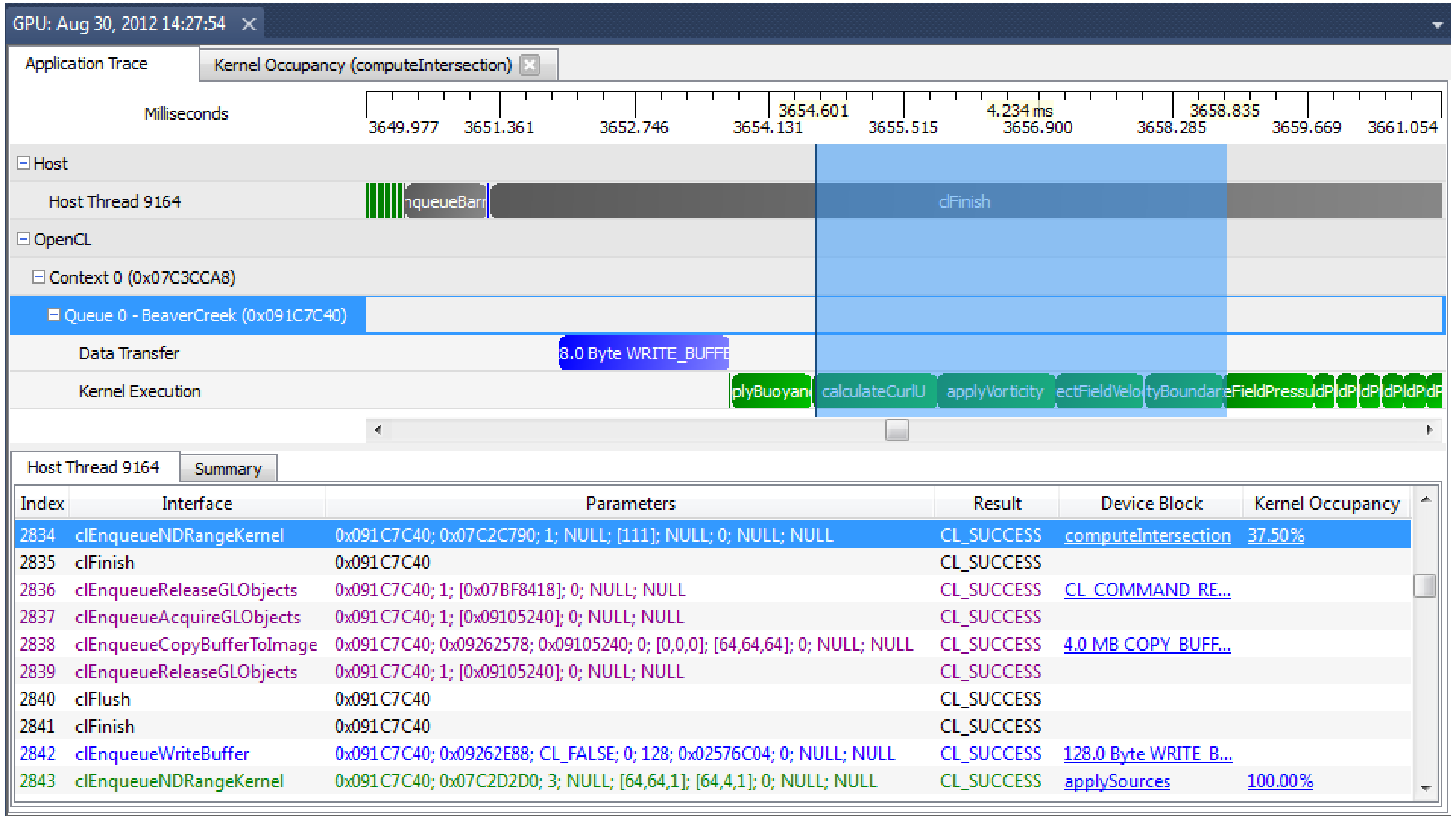
Performance Counters

☑ Show Zero Column

	Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	ALUBusy (%)
1	applySources__k1_BeaverCreek1	1	604	269	{ 64 64 1}	{ 64 4 1}	0.03500	0	7	NA	0	0	100	23.35
2	applyBuoyancy__k2_BeaverCreek1	2	604	270	{ 64 64 64}	{ 64 4 1}	0.81560	0	4	NA	0	0	100	22.28
3	calculateCurlU__k3_BeaverCreek1	3	604	271	{ 64 64 64}	{ 64 4 1}	1.20496	0	9	NA	0	0	75	15.54
4	applyVorticity__k4_BeaverCreek1	4	604	272	{ 64 64 64}	{ 64 4 1}	1.72332	0	5	NA	0	2	100	85.82
5	advectFieldVelocity__k5_BeaverCreek1	5	604	273	{ 64 64 64}	{ 64 4 1}	0.80228	0	15	NA	0	0	50	54.60
6	applyVelocityBoundaryCondition__k6_BeaverCreek1	6	604	274	{ 64 64 64}	{ 64 4 1}	0.97776	0	4	NA	0	0	100	12.85
7	computeFieldPressurePrep__k7_BeaverCreek1	7	604	275	{ 64 64 64}	{ 64 4 1}	1.03408	0	9	NA	0	0	75	24.89
8	computeFieldPressureIter__k8_BeaverCreek1	8	604	276	{ 64 64 64}	{ 64 4 1}	0.37424	0	7	NA	0	0	100	76.58

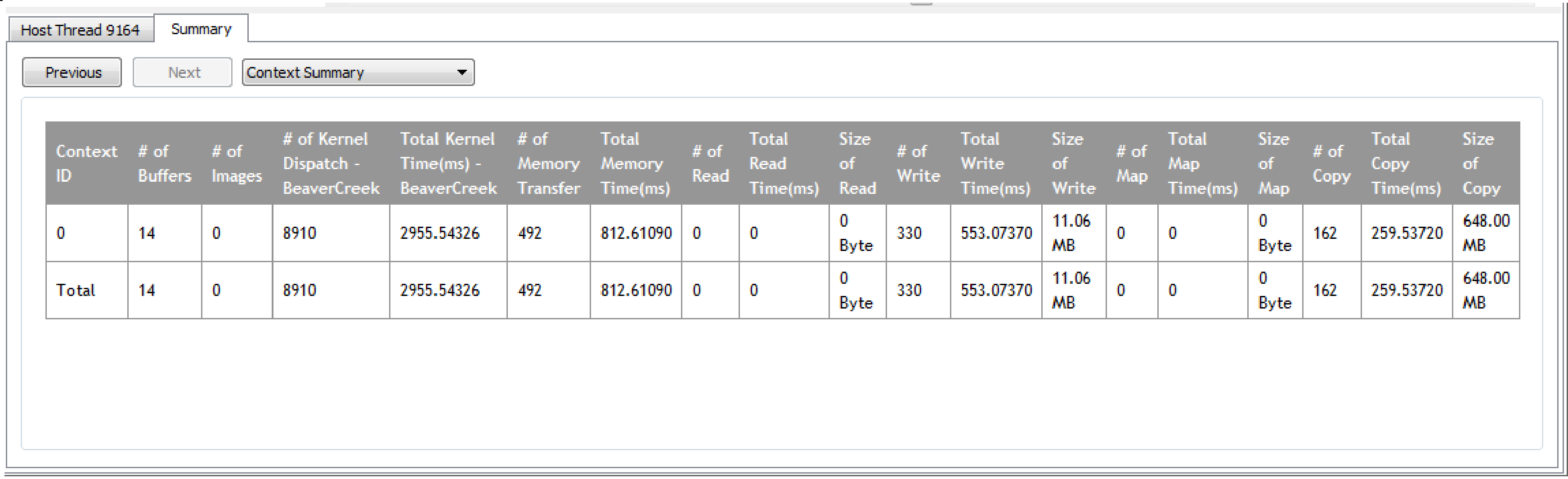
GPU PROFILING KEY FEATURES AND BENEFITS

- **OpenCL™ Timeline visualization**
  - Visualize the application high level structure
  - Visualize kernel execution and data transfer operations
  - Visualize host code execution



GPU PROFILING KEY FEATURES AND BENEFITS

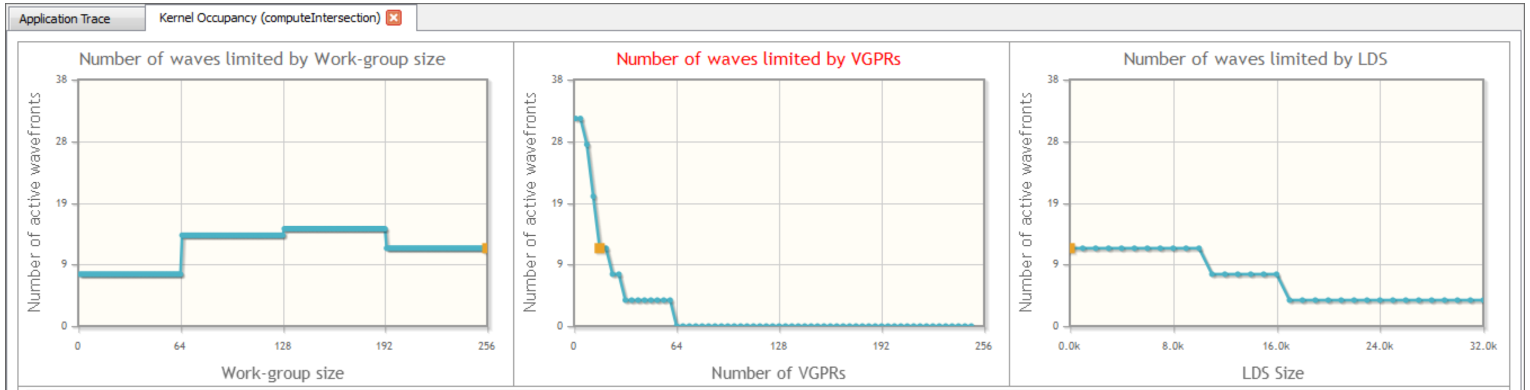
- **OpenCL™ Application Summary pages**
  - Find incorrect or inefficient usage of the OpenCL™ API using the OpenCL™ analysis module
  - Find the API hotspots
  - Find the bottlenecks between kernel execution and data transfer operations
  - Find the top 10 data transfer and kernel execution operations



Context ID	# of Buffers	# of Images	# of Kernel Dispatch - BeaverCreek	Total Kernel Time(ms) - BeaverCreek	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time(ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time(ms)	Size of Map	# of Copy	Total Copy Time(ms)	Size of Copy
0	14	0	8910	2955.54326	492	812.61090	0	0	0 Byte	330	553.07370	11.06 MB	0	0	0 Byte	162	259.53720	648.00 MB
Total	14	0	8910	2955.54326	492	812.61090	0	0	0 Byte	330	553.07370	11.06 MB	0	0	0 Byte	162	259.53720	648.00 MB

GPU PROFILING KEY FEATURES AND BENEFITS

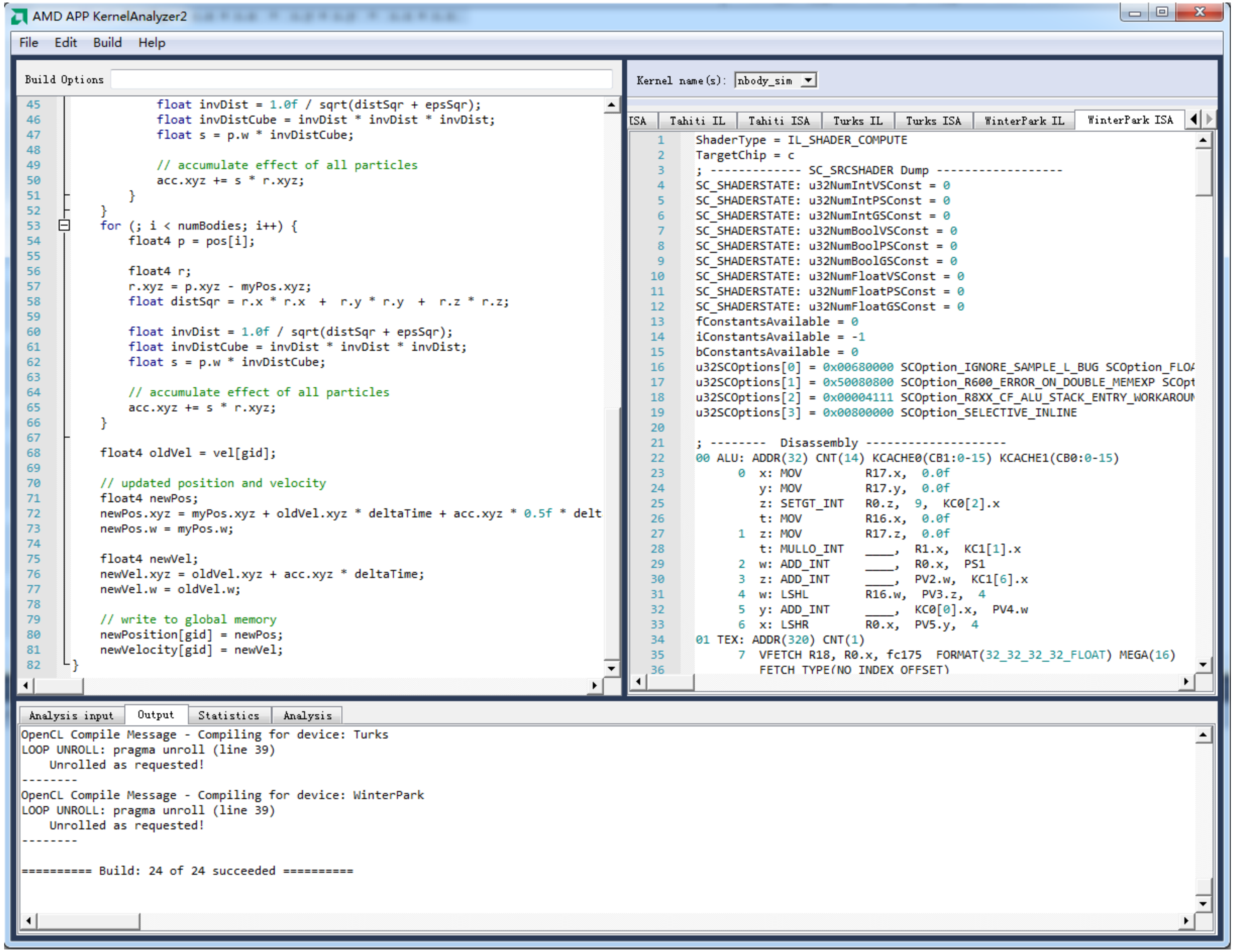
- **OpenCL™ Kernel Occupancy Viewer**
  - Calculates and displays a kernel occupancy number, which estimates the number of in-flight wavefronts on a compute unit as a percentage of the theoretical maximum number of wavefronts that the compute unit can support
  - Find out which kernel resource (GPR usage, LDS size, or Work-group size) is currently limiting the number of in-flight wavefronts
  - Displays graphs showing how kernel occupancy would be affected by changes in each kernel resource



Variable	Value	Device Limit
Device Info		
Device name	BeaverCreek	
Number of compute units	5	
Max number of waves per compute unit	32	
Max number of work-groups per compute unit	8	
Wavefront size	64	
Kernel Info		
Kernel name	computeIntersection	
Vector GPR usage per work-item	16	248
LDS usage per work-group	0	32768
Flattened work-group size	256	256
Flattened global work size	111	16777216
Number of waves per work-group	4	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	12	32
Number of waves limited by LDS and Work-group size	32	32
Number of waves limited by Work-group size	32	32
Limiting factor(s)	VGPR	
Estimated occupancy	37.5%	

STATIC KERNEL ANALYSIS – KEY FEATURES AND BENEFITS

- Compile, analyze and disassemble the OpenCL kernel and supports multiple GPU device targets.
- View any kernel compilation errors and warnings generated by the OpenCL runtime.
- View the AMD Intermediate Language (IL) code generated by the OpenCL run-time.
- View the ISA code generated by the AMD Shader Compiler.
- View various statistics generated by analyzing the ISA code.
- View General Purpose Registers and spill registers allocated for the kernel.



LANGUAGE BINDING

Language Binding Tools: allows you to write the OpenCL host code in your own programming language. The OpenCL kernels you use are still written in the OpenCL language.

C	•Calseum (for ATI CAL) •HMPP Workbench from CAPS entreprise •Libra SDK from GPU Systems
Fortran	•HMPP Workbench from CAPS entreprise
Java	•JavaCL
Matlab	•IPT_ATI_PROJECT •Libra SDK from GPU Systems
.NET	•OpenCL .Net •OpenTK
Python	•CLyther •PyGWA (for ATI CAL) •PyOpenCL •Pythoncl

Kernel Translation Tools: additionally allow you to write the kernel itself in your own programming language. The tools then translate your kernel to the OpenCL language.

Java	•Aparapi
Scala	•ScalaCL

AGENDA

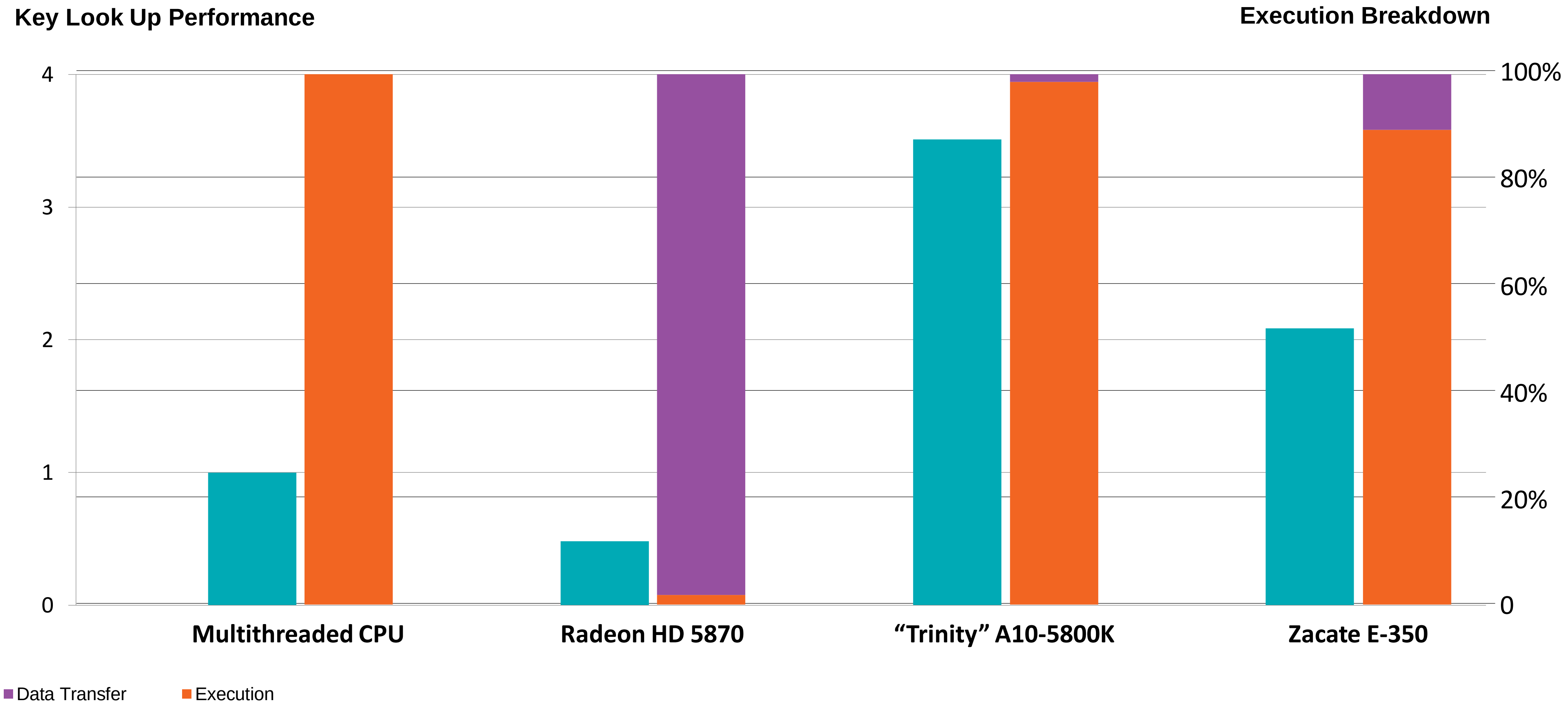
- What's OpenCL
- Fundamentals for OpenCL programming
- OpenCL programming basics
- OpenCL programming tools
- Demos

MEMCACHED

A Distributed Memory Object Caching System Used in Cloud Servers

- Generally used for short-term storage and caching, handling requests that would otherwise require database or file system accesses
- Used by Facebook, YouTube, Twitter, Wikipedia, Flickr, and others
- Effectively a large distributed hash table
  - Responds to store and get requests received over the network
  - Conceptually:
    - store(key, object)
    - object = get(key)

OFFLOADING MEMCACHED KEY LOOKUP TO THE GPU



T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2012)*, April 2012.
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6189209

SUMMARY

- OpenCL is an open standard for programming on heterogeneous computing platforms
- OpenCL programming requires
  - Parallel computing thinking
  - GPU architecture knowledge for performance consideration
  - Deep understanding of OpenCL architecture to control devices
- OpenCL key concepts
  - Platform, device, context
  - Command queue, buffer/image, data copying, program, Kernel, Kernel execution
- OpenCL programming tools
  - Code XL
- Next day
  - GPU architecture
  - Kernel optimization
  - OpenCL application optimization

THANKS!

DISCLAIMER & ATTRIBUTION

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

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

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

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

ATTRIBUTION