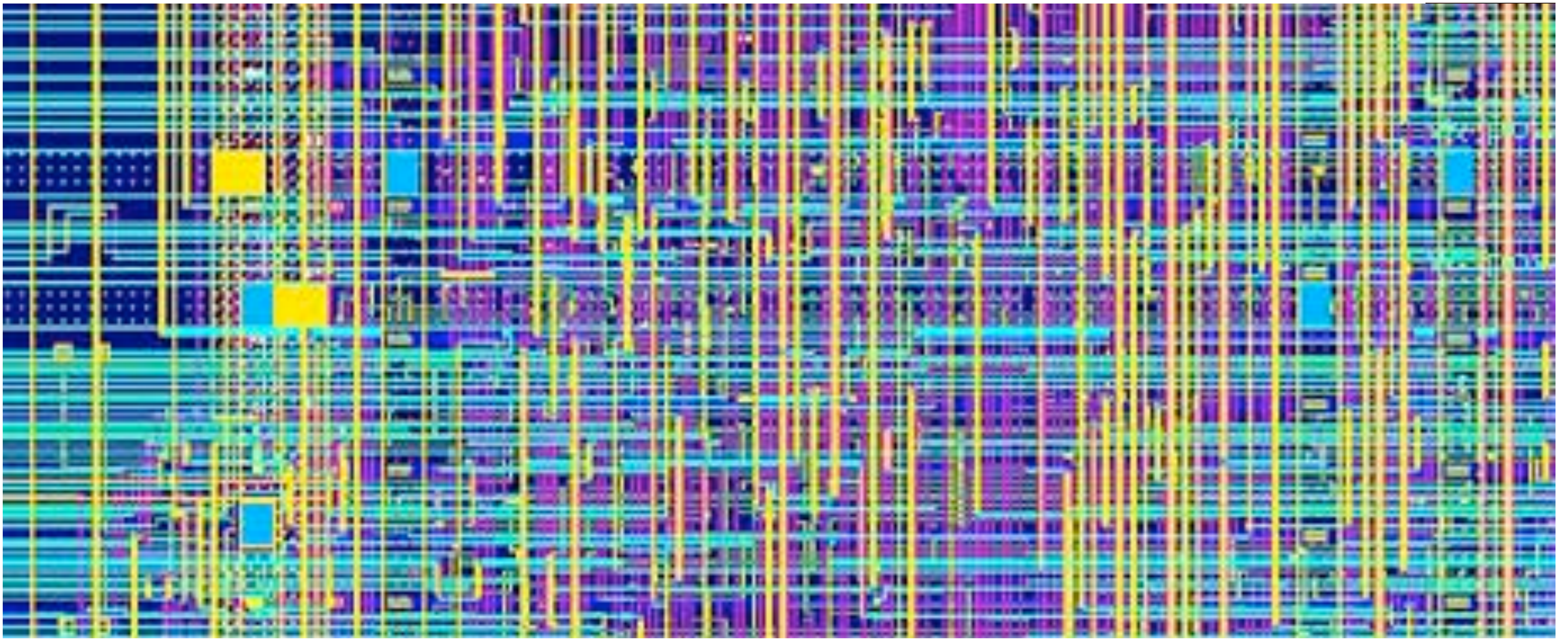


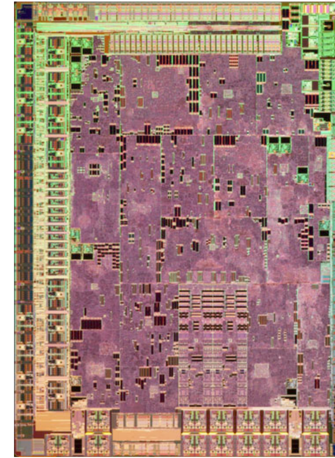
OpenCL

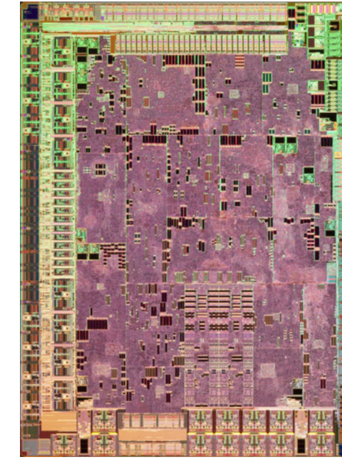
Making Use of What You've Got

Getting More Computing Power

- **Contains one Intel Core 2 Duo**
  - Two computing cores
  - One source of parallelism for the operating system to support
  - Better performance
    - core 1 @ 100% utilization ripping a DVD
    - core 2 running OS + Microsoft PowerPoint

Getting More Computing Power





- **Also contains two NVIDIA 9400M Graphics Processors**
  - Each has 16 computing cores
  - Untapped source of parallelism to support even better performance

- **But how?**
  - GPUs are not standardised like INTEL architectures
  - GPUs are optimised for very different copmputing tasks

The Khronos API Ecosystem

Desktop 3D Ecosystem

COLLADA
3D Asset Interchange Format

OpenGL
Cross platform desktop 3D

Parallel computing and visualization in scientific and consumer applications

OpenCL
Heterogeneous Parallel Computing

Streamlined APIs for mobile and embedded graphics, media and compute acceleration

OpenGL|ES
Embedded 3D

OpenMAX
Streaming Media and Image Processing

OpenVG
Vector 2D

OpenSL|ES
Enhanced Audio

EGL
Surface and synch abstraction

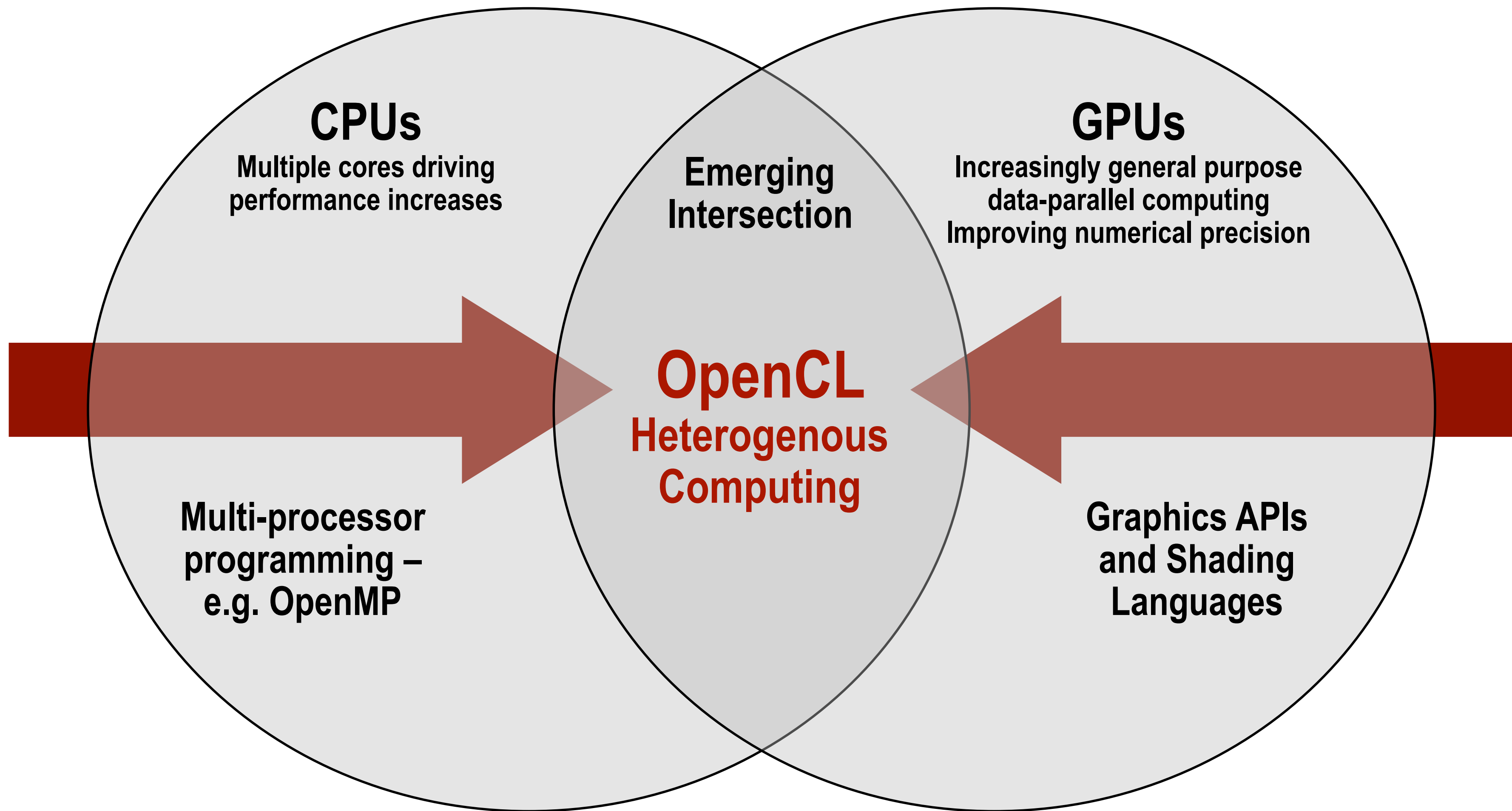
OpenKOGS
Integrated Mixed-media Stack

OpenKODE
Mobile OS Abstraction

Umbrella specifications define coherent acceleration stacks for mobile application portability

Hundreds of man years invested by industry experts to create coordinated ecosystem

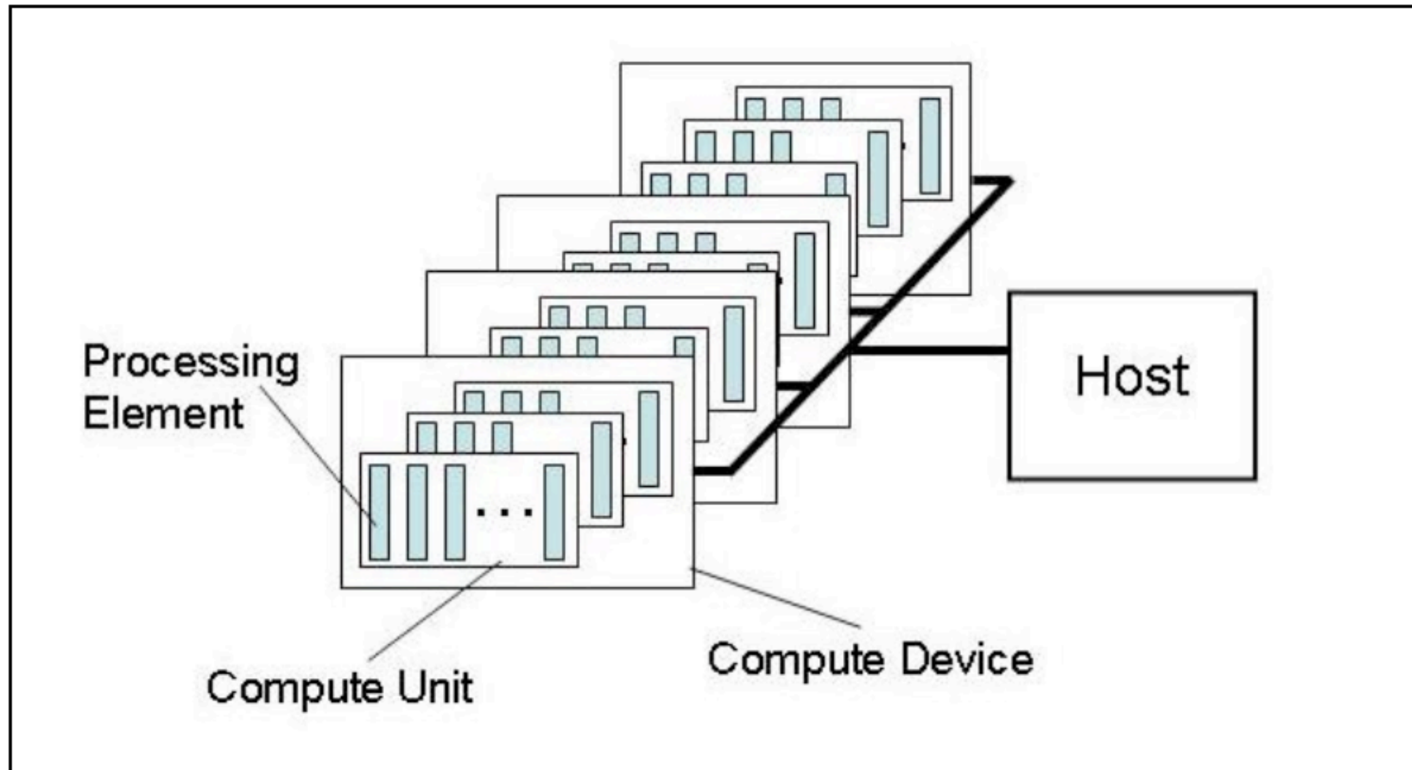
Processor Parallelism



OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

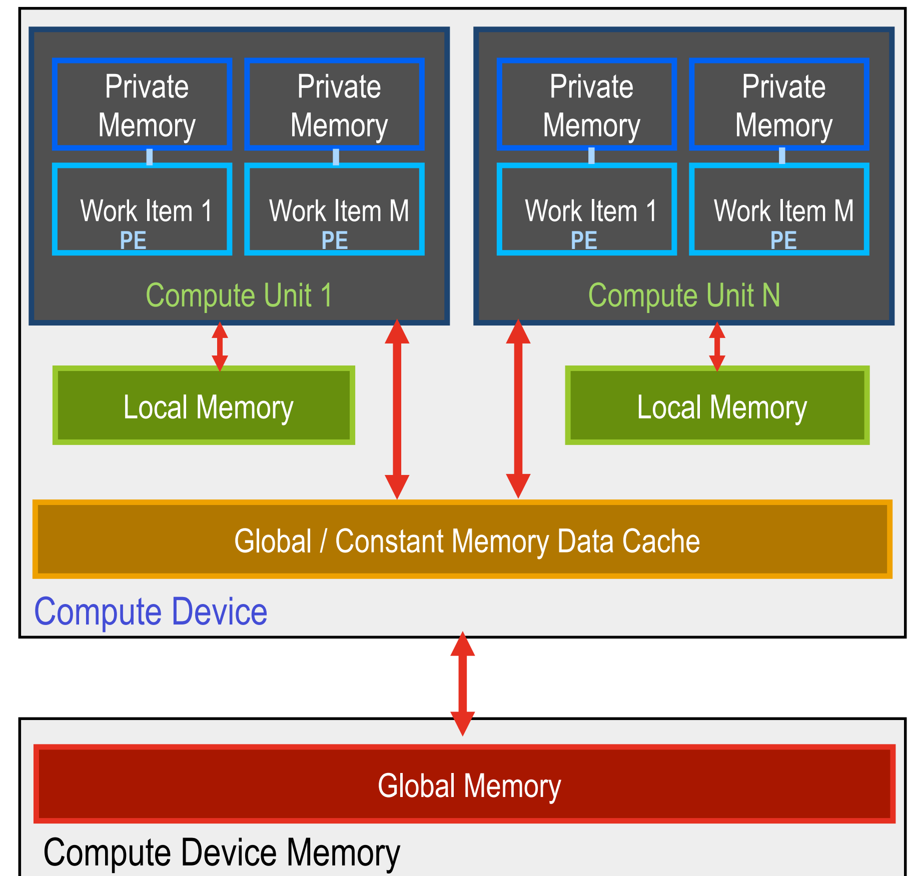
OpenCL Platform Model (Section 3.1)



- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

OpenCL Memory Model (Section 3.3)

- **Shared memory model**
  - Relaxed consistency
- **Multiple distinct address spaces**
  - Address spaces can be collapsed depending on the device's memory subsystem
- **Address spaces**
  - Private - private to a *work-item*
  - Local - local to a *work-group*
  - Global - accessible by all work-items in all work-groups
  - Constant - read only global space
- **Implementations map this hierarchy**
  - To available physical memories



OpenCL Execution Model (Section 3.2)

- **OpenCL *Program*:**
  - Kernels
    - Basic unit of executable code — similar to C functions, CUDA kernels, etc.
    - Data-parallel or task-parallel

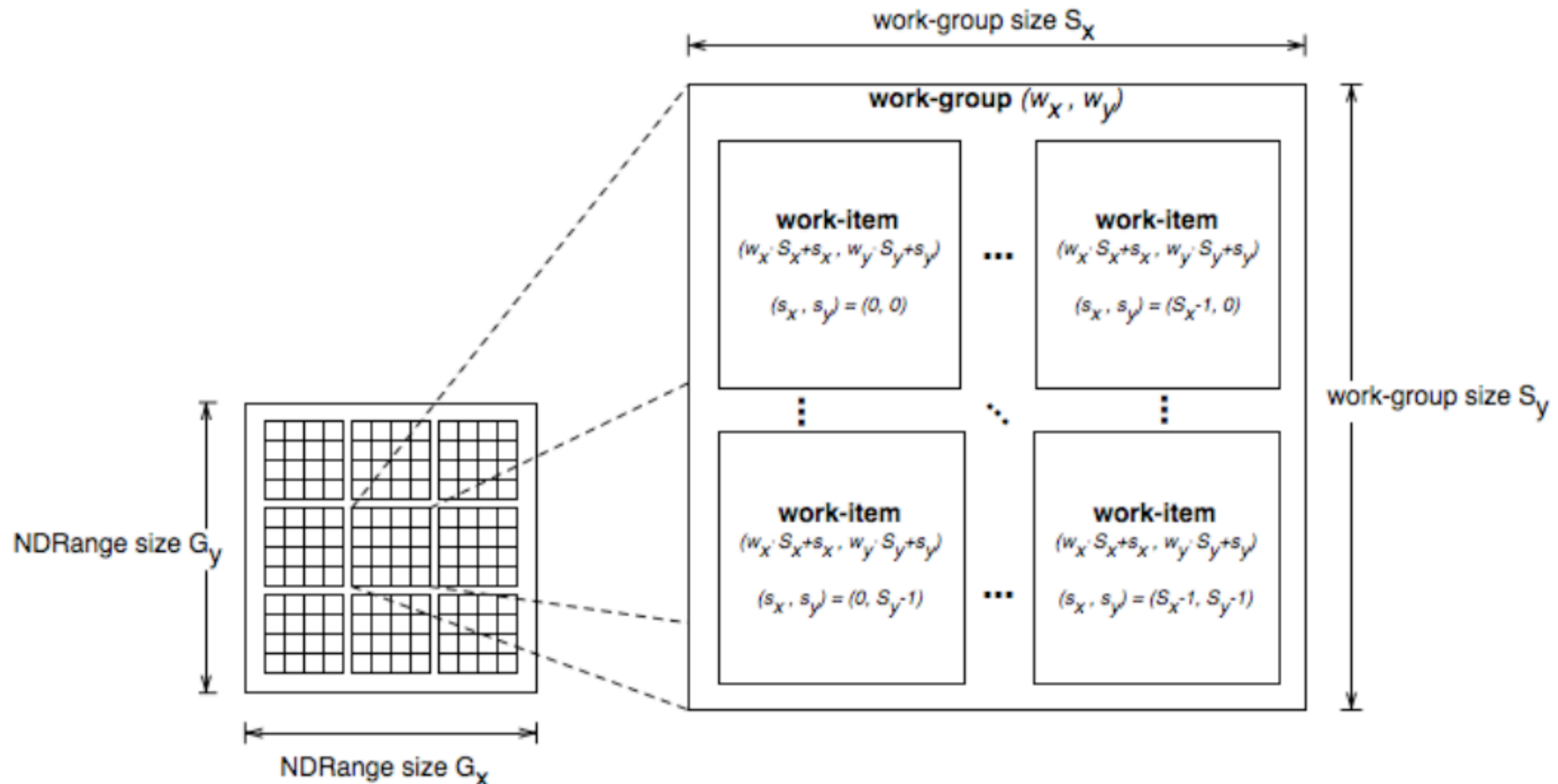
  - Host Program
    - Collection of compute kernels and internal functions
    - Analogous to a dynamic library

- ***Kernel* Execution**
  - The host program invokes a kernel over an index space called an ***NDRange***
    - NDRange, "N-Dimensional Range", can be a 1D, 2D, or 3D space

  - A single kernel instance at a point in the index space is called a ***work-item***
    - Work-items have unique global IDs from the index space

  - Work-items are further grouped into ***work-groups***
    - Work-groups have a unique work-group ID
    - Work-items have a unique local ID within a work-group

Kernel Execution



- Total number of work-items = G_x * G_y
- Size of each work-group = S_x * S_y
- Global ID can be computed from work-group ID and local ID

Programming Model

Data-Parallel Model (Section 3.4.1)

- **Must be implemented by *all* OpenCL compute devices**

- **Define N-Dimensional computation domain**
  - Each independent element of execution in an N-Dimensional domain is called a *work-item*
  - N-Dimensional domain defines total # of work-items that execute in parallel = *global work size*

- **Work-items can be grouped together — *work-group***
  - Work-items in group can communicate with each other
  - Can synchronize execution among work-items in group to coordinate memory access

- **Execute multiple work-groups in parallel**
  - Mapping of global work size to work-group can be implicit or explicit

Programming Model

Task-Parallel Model (Section 3.4.2)

- **Some compute devices can also execute task-parallel compute kernels**
- **Execute as a *single* work-item**
  - A compute kernel written in OpenCL
  - A native C / C++ function

Basic OpenCL Program Structure

- **Host program**
  - Query compute devices
  - Create contexts
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources
- **Kernels**
  - C code with some restrictions and extensions

Platform Layer (Query compute devices; Create contexts)

Runtime (Create memory objects … Clean up OpenCL resources)

Language (Kernels)

Memory Objects (Section 5.2)

- **Buffer objects**
  - 1D collection of objects (like C arrays)
  - Scalar types & Vector types, as well as user-defined Structures
  - Buffer objects accessed via pointers in the kernel

- **Image objects**
  - 2D or 3D texture, frame-buffer, or images
  - Must be addressed through built-in functions

- **Sampler objects**
  - Describe how to sample an image in the kernel
    - Addressing modes
    - Filtering modes

Sample walkthrough oclVectorAdd

- **Simple element by element vector addition**

  For all i,

  C(i) = A(i) + B(i)

- **Outline**
  - Query compute devices
  - Create Context and Queue
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

Kernel Code

```
// The JIT source code for the computation kernel
// **************************************************************
const char* cVectorAdd[ ] =
{
    "__kernel void VectorAdd(",
    "    __global const float* a,",
    "    __global const float* b,",
    "    __global float* c)",
    "{",
    "        int iGID = get_global_id(0);",
    "        c[iGID] = a[iGID] + b[iGID];",
    "}"
};
const int SOURCE_NUM_LINES = sizeof(cVectorAdd) / sizeof(cVectorAdd[0]);
```

Declarations

```
cl_context cxMainContext;                  // OpenCL context
cl_command_queue cqCommandQue;             // OpenCL command que
cl_device_id* cdDevices;                   // OpenCL device list
cl_program cpProgram;                      // OpenCL program
cl_kernel ckKernel;                        // OpenCL kernel
cl_mem cmMemObjs[3];                       // OpenCL memory buffer objects
cl_int ciErrNum = 0;                       //  Error code var
size_t szGlobalWorkSize[1];                //  Global # of work items
size_t szLocalWorkSize[1];                 //  # of Work Items in Work Group
size_t szParmDataBytes;                    //  byte length of parameter storage
size_t szKernelLength;                     //  byte Length of kernel code
int iTestN = 10000;                  //  Length of demo test vectors
```

Contexts and Queues

```
// create the OpenCL context on a GPU device
cxMainContext = clCreateContextFromType (0, CL_DEVICE_TYPE_GPU,  NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo (cxMainContext, CL_CONTEXT_DEVICES, 0, NULL,  &szParmDataBytes);
cdDevices = (cl_device_id*)malloc(szParmDataBytes);
clGetContextInfo (cxMainContext, CL_CONTEXT_DEVICES, szParmDataBytes, cdDevices, NULL);

// create a command-queue
cqCommandQue = clCreateCommandQueue (cxMainContext, cdDevices[0], 0, NULL);
```

Create Memory Objects

```
// allocate the first source buffer memory object… source data, so read only
cmMemObjs[0] = clCreateBuffer (cxMainContext,
                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(cl_float) * iTestN, srcA, NULL);

// allocate the second source buffer memory object … source data, so read only
cmMemObjs[1] = clCreateBuffer (cxMainContext,
                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(cl_float) * iTestN, srcB, NULL);

// allocate the destination buffer memory object … result data, so write only
cmMemObjs[2] = clCreateBuffer (cxMainContext, CL_MEM_WRITE_ONLY,
                               sizeof(cl_float) * iTestN, NULL, NULL);
```

Create Program and Kernel

```
// create the program, in this case from OpenCL C source string array
cpProgram = clCreateProgramWithSource (cxMainContext, SOURCE_NUM_LINES,
                                       cVectorAdd, NULL, &ciErrNum);

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

// create the kernel
ckKernel = clCreateKernel (cpProgram, "VectorAdd", &ciErrNum);

// set the kernel Argument values
ciErrNum = clSetKernelArg (ckKernel, 0, sizeof(cl_mem), (void*)&cmMemObjs[0] );
ciErrNum |= clSetKernelArg (ckKernel, 1, sizeof(cl_mem), (void*)&cmMemObjs[1] );
ciErrNum |= clSetKernelArg (ckKernel, 2, sizeof(cl_mem), (void*)&cmMemObjs[2] );
```

Launch Kernel and Read Results

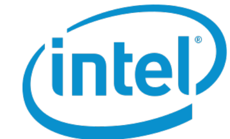
```
// set work-item dimensions
szGlobalWorkSize[0] = iTestN;
szLocalWorkSize[0]= 1;

// execute kernel
ciErrNum = clEnqueueNDRangeKernel (cqCommandQue, ckKernel, 1, NULL,
                                   szGlobalWorkSize, szLocalWorkSize,
                              0, NULL, NULL);

// read output
ciErrNum = clEnqueueReadBuffer(cqCommandQue, cmMemObjs[2], CL_TRUE,
                                   0, iTestN * sizeof(cl_float), dst, 0, NULL, NULL);
```

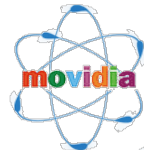
Cleanup

```
// release kernel, program, and memory objects
DeleteMemobjs (cmMemObjs, 3);
free (cdDevices);
clReleaseKernel (ckKernel);
clReleaseProgram (cpProgram);
clReleaseCommandQueue (cqCommandQue);
clReleaseContext (cxMainContext);
```

OpenCL Working Group

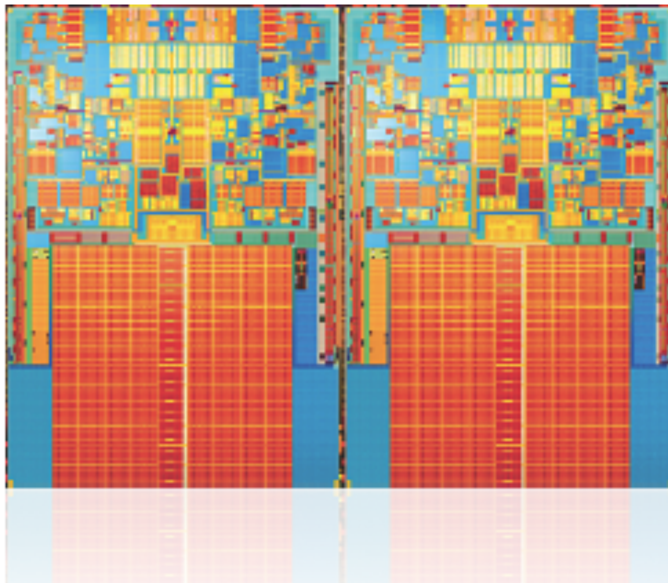
- **Diverse industry participation**
  - Processor vendors, system OEMs, middleware vendors, application developers
- **Many industry-leading experts involved in OpenCL's design**
  - A healthy diversity of industry perspectives
- **Apple initially proposed and is very active in the working group**
  - Serving as specification editor
- **Here are some of the other companies in the OpenCL working group**





Products?

- Apple “Snow Leopard” OS X.6



“Grand Central,” a new set of technologies built into Snow Leopard, brings unrivaled support for multicore systems to Mac OS X. More cores, not faster clock speeds, drive performance increases in today’s processors. Grand Central takes full advantage by making all of Mac OS X multicore aware and optimizing it for allocating tasks across multiple cores and processors. Grand Central also makes it much easier for developers to create programs that squeeze every last drop of power from multicore systems.