Sistemi Operativi e Reti

GPGPU Computing: the multi/many core computing era

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in Informatica

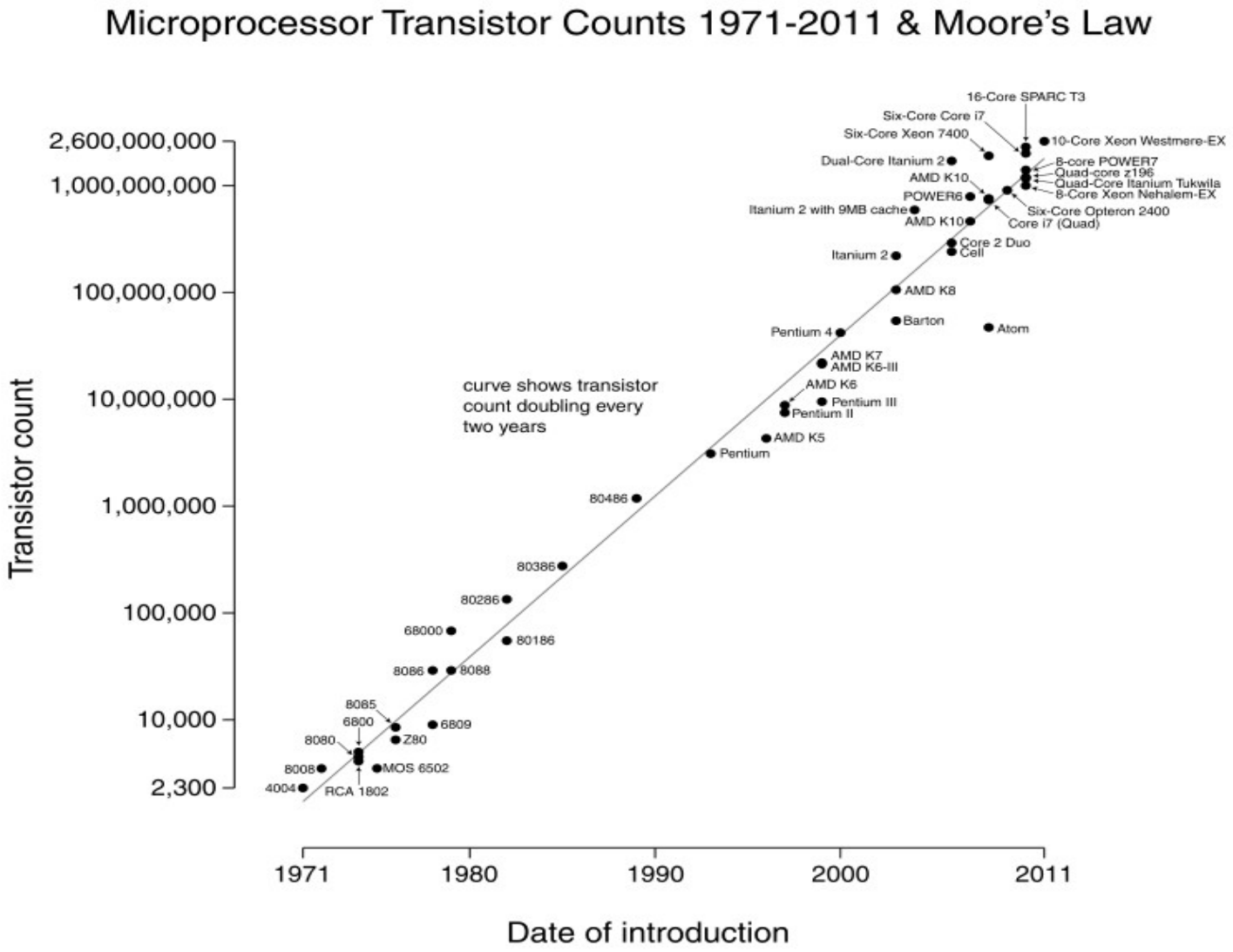
Osvaldo Gervasi

ogervasi@computer.org

Outline

- Introduction
- General Purpose GPU Computing
- GPU and CPU evolution
- **OpenCL**: a standard for programming heterogeneous devices
- A case study: Advanced Encryption Standard (AES)
- Scheduling issues

Moore's law

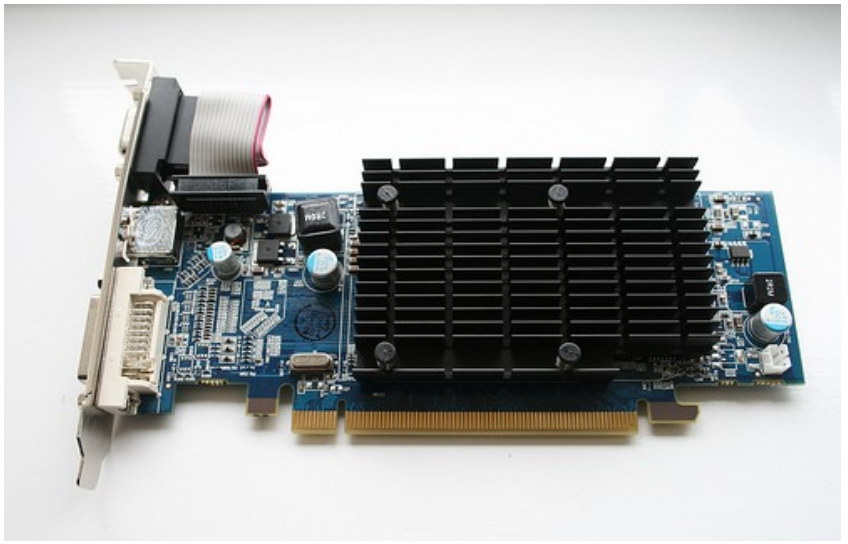


Source: Wikipedia

GPGPU Computing

- GPU computing or GPGPU “is the use of a GPU (graphics processing unit) to carry on general purpose scientific and engineering computing” [Nvidia].

Nvidia TESLA GPU



Sapphire ATI Radeon HD 4550 GPU

Heterogeneous Computing

- Heterogeneous Computing is the transparent use of all computational devices to carry out general purpose scientific and engineering computing.

Android and Ubuntu support

The **Arndale** Board based on **ARM Cortex-A15** with **Mali-T604 Samsung Exynos 5250** development platform

Very promising architecture for hererogeneous computing: it is built on **32nm low-power HKMG (High-K Metal Gate)**, and features a dual-core **1.7GHz mobile CPU built on ARM® Cortex™-A15** architecture plus an integrated **ARM Mali™-T604 GPU** for increased performance density and energy efficiency

The future of Super Computing Centers: the MontBlanc EU project

- Heterogeneous Computing and minimization of power consumption: the new HPC Center of the future!

http://www.montblanc-project.eu

MontBlanc selected the Samsung Exynos 5 Processors

NVIDIA Tegra

Linux support: Linux for Tegra (L4T)

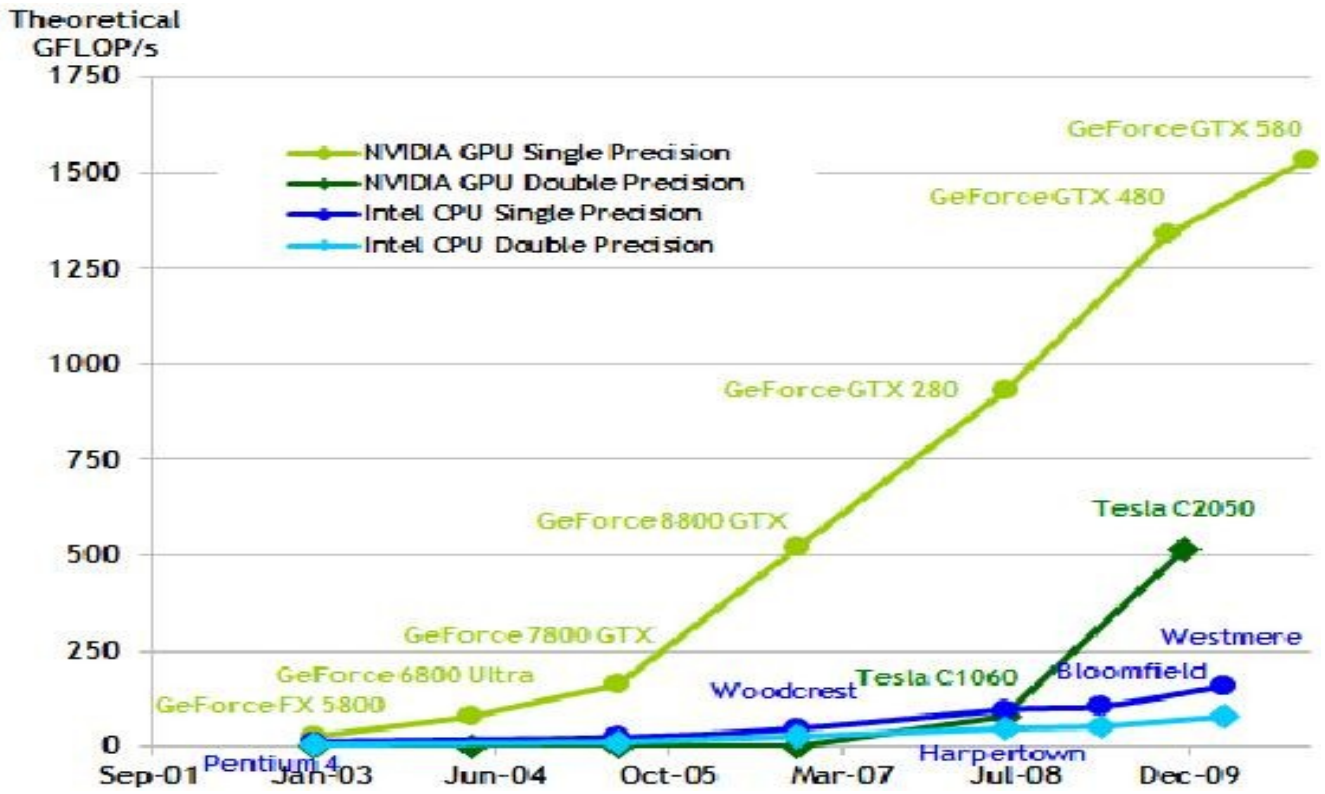
Quad-core **NVIDIA Tegra T3** based Embedded Toradex Colibri T30 Computer On Module, announced on January 31, 2012. The cores are ARM Cortex-A9. The GPU is a 520 ULP GeForce.

Audi had selected the Tegra T3 processor for its **in-vehicle infotainment systems and digital instruments display**. The processor has been integrated into Audi's entire line of vehicles worldwide, since 2013. The latest versions of the Tegra (K1 and X1) are revealing extremely interesting capabilities for developing **self-driving cars**.

General Purpose GPU Computing

- GPUs are:
  - cheap and powerful
  - ready to use
  - highly parallel (thousands of cores)
  - suitable for SIMD applications
- SIMD architectures may help solving a large set of computational problems:
  - Data Mining
  - Cryptography
  - Earth sciences
  - Montecarlo simulations
  - Astrophysics ….

GPU evolution vs. CPU evolution



Tipologia	Modello	GFLOPS	Anno di produzione
CPU	Opteron 6 series	120	2011-Q1
GPU	Nvidia GTX 580	1500	2010-Q4
CPU	Intel i7 9x	50	2009-Q4
GPU	FireStream 9270	1000	2009-Q1

GPUs

- GPUs are low cost devices available on the market
- Incredible performances
- Very fast developments
- SIMD architecture (the same as vector computers)
- Several problems are suitable to be solved using a SIMD approach

Applications on

- Criptography
- Linear Algebra
- Data mining
- Life sciences
- Scientific computing
- Signal theory
- Video processing

Computational Graphics

Formal Definition

The production of bitmap images based on data acquired from an external source or computed by means of a computational model

Phases

- Definition of the objects in the scene
- Image rendering

Graphic Pipeline

- Set of operations for the graphic rendering

Rendering operations

- **Transfer of the scene description**: the set of vertex defining the objects, the data associated to the scene illumination, the textures, the observer's point of view.
- **Vertex transformations**: rotations, scaling and objects' translation
- **Clipping**: elimination of the objects or parts of them not visible from the observer's point of view.
- **Lighting and shading**: evaluation of the interactions of the light sources with the shapes, evaluating their shadowing.
- **Rasterization**: generation of the bitmap image. 3D coordinates are transformed in 2D coordinates. Textures and other graphic effects are also applied.

GPU's evolution

- The Graphic Processing Unit is the device devoted to the carry out the rendering in the modern video boards
- **1980**: the first video chips with limited functions without 2D graphic capabilities
- 1985: the graphic chips were similar to CPUs, with some modifications (design and ISA). Expensive solution for promoting CAD applications.
- 1990: graphic chips integrated, dedicated and programmable (lower costs)
- Starting from **1995**, 3D graphics performance issues emerged thanks to the success of **video games**.
  - Integrated graphic chips for **3D acceleration**
  - The **OpenGL** and **DirectX** specifications were released, hiding the complexity of programming 3D graphics accelerators
  - The **graphic pipeline** started to be executed in the GPU

GPU's evolution

- In **2000** the **shading operations** (ability to perform operations which implement the graphic pipeline) are included in the GPU capabilities.

  Types of shader:

  - **Vertex shading**: manages and transforms the vertex positions in an object
  - **Pixel or fragment shading**: manages the image pixels, enabling the texture mapping
  - **Geometrical shading**: starting from the vertex of a given object builds more complex objects.
- Shading capabilities became **programmable**
  - each shader were executed on dedicated units
  - GPUs became flexible almost like CPUs

GPU's evolution

- The first General Purpose GPU Computing projects appear, which utilize the shading units on the vertices.
- The first examples were based on OpenGL APIs to define shaders on vertices which mapped the parallel general purpose program.
- The problem of load balancing the specialized shader units appear

GPU's evolution

- In **2005** the Unified Shader Model is introduced: the various types of shaders are defined using a common set of APIs.
- The compute units are all identical.
- In **2007** the concept of General Purpose GPU became a reality and was fully implemented:
  - NVIDIA released Compute Unified Device Architecture (**CUDA**)
  - AMD released **Brook+**
  - These frameworks allow to use the compute devices of the GPU without using graphic APIs.

The multicore era

- In the same years (2005-2007) CPUs became **multicore**

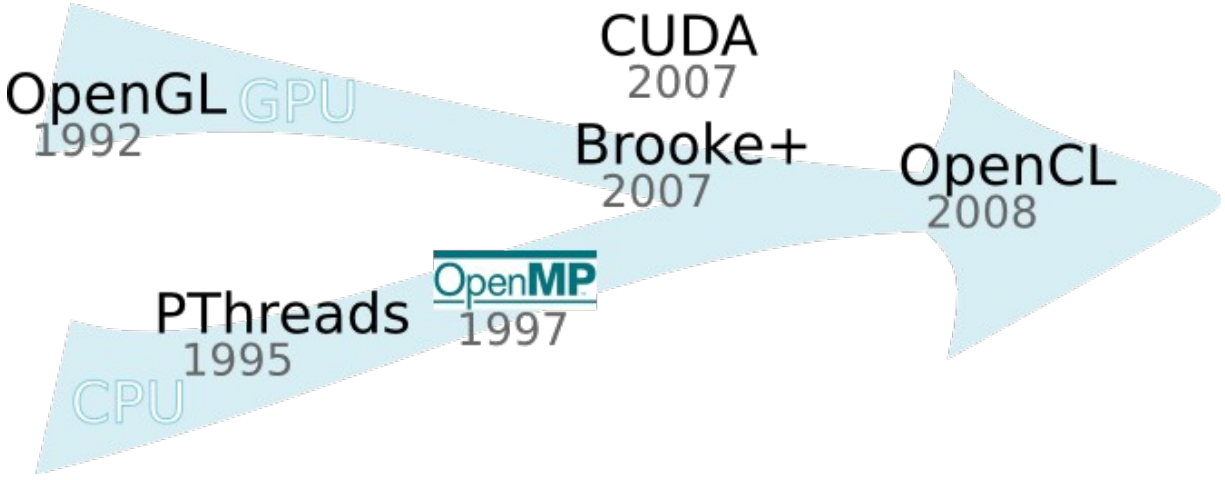
Intel Yonah (Core Duo) low-power dual core processor, introduced on January 2006
Intel Core i7-3920XM Processor Extreme Edition has 6 cores/12 threads (Q4'12).

AMD Opteron 2212 introduced in August 2006. The first AMD dual core (Opteron 875) was released on April 2005.
AMD Opteron 6366 HE (Q4'12) has 16 cores and high energy efficiency.

Transparent programming of heterogeneous devices

- In 2008 Khronos Compute Working Group released the Open Computing Language (**OpenCL**)

OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and hand-held/embedded devices. [Khronos].



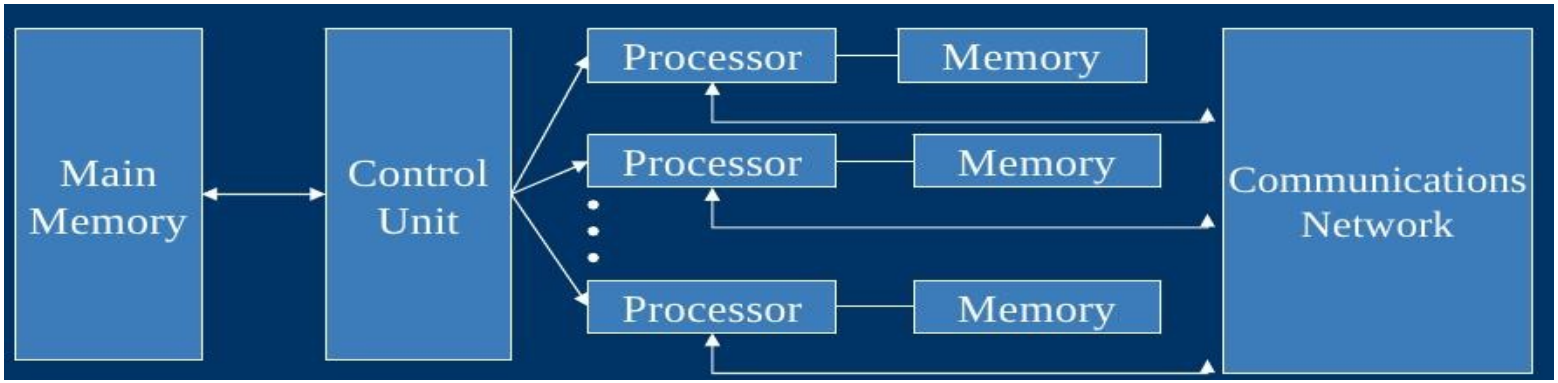
Computer architectures

Flynn taxonomy

- **SISD** Single instruction on Single Data

  (es. Architetture Von Neumann tradizionale)
- **SIMD** Single instruction on Multiple Data
  (es.Processori vettoriali)
- **MISD** Multiple instruction on Single Data
  (es. Controller di volo dello Space Shuttle)
- **MIMD** Multiple instruction on Multiple Data
  (es. architetture moderne multicore: Xeon Clovertown)

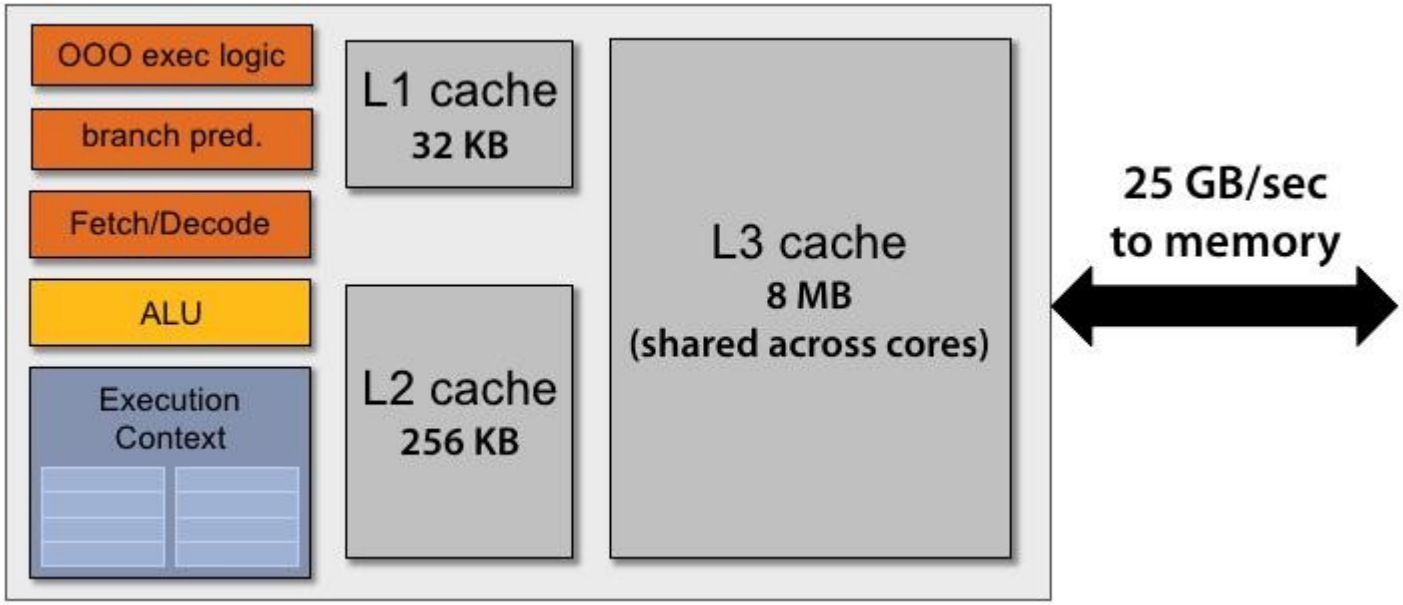
SIMD



- Executes a single instruction set on different sets of data utilizing several computational units at the same time
- Instruction fetch and decode occur only once
- There is a single control unit (CU) which manages the instruction flow of a given program

Vector Processors: computational units which, after the instruction fetch and decode, execute it on the data stored in the vector registers. The load-store unit moves the data from the central memory to the vector registers and vice-versa

MIMD



- Executes different instructions simultaneously
- Each processor has its own CU
- Each processor may execute a task or part of it

GPU architecture

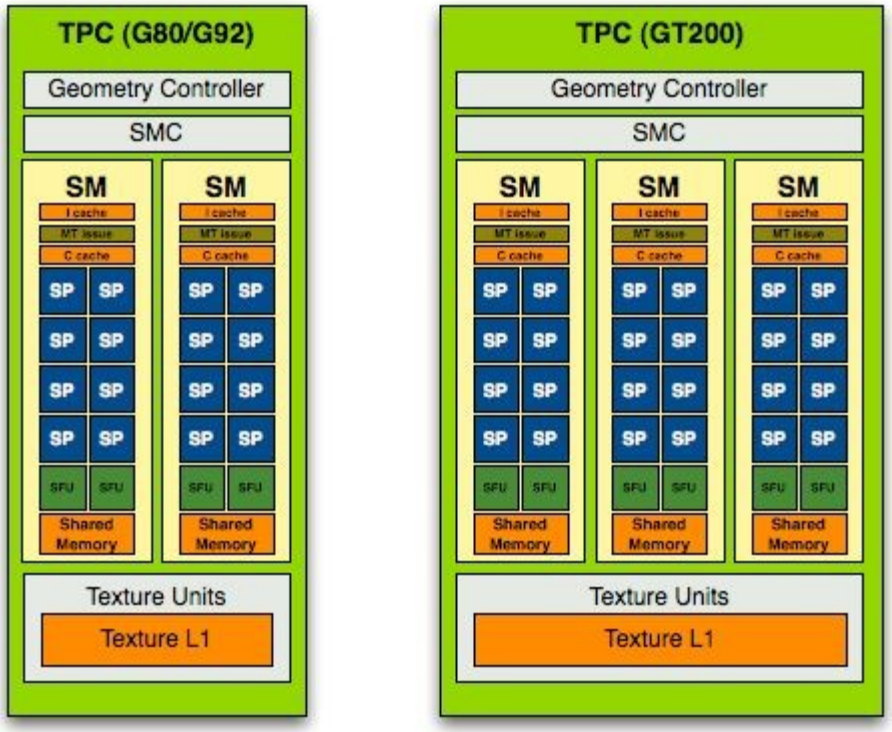
NVIDIA G80 (2007)

- GPU components:
  - Host interface: connects the device via PCIe bus to the Host and manages communications (instructions and data)
  - Scalable Processor Array (SPA) execute the programmable operations using the (programmable) Texture Processor Cluster (TPC).
  - Video RAM

Nvidia G80

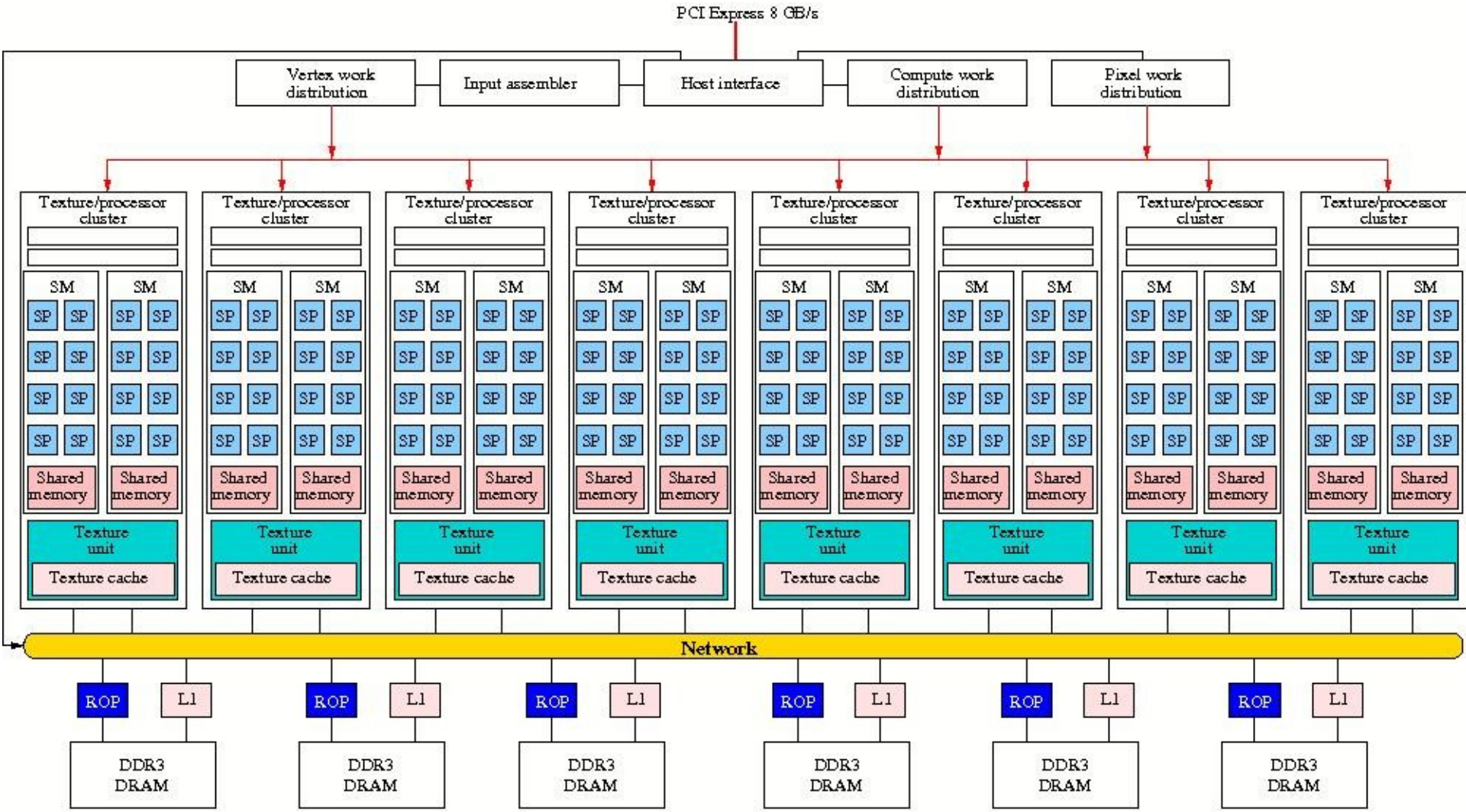
- TPC contains: Geometry controller, two Streaming Multiprocessors (SM), Texture Unit and Streaming Multiprocessor Controller (SMC)
- In particular a SM contains:
  - MT unit: multithreded instruction fetch and issue unit
  - 8 Streaming Processor (SP): scalar computational units
  - 2 Special Function Unit: used for calculating transcendental functions
  - Cache of instructions
  - Cache for constant memory
  - Shared memory (16KB)
  - 8192 registers (32 bit)

Nvidia G80



Texture Processor Cluster in chipset G80 and GT200

Vnidia G80

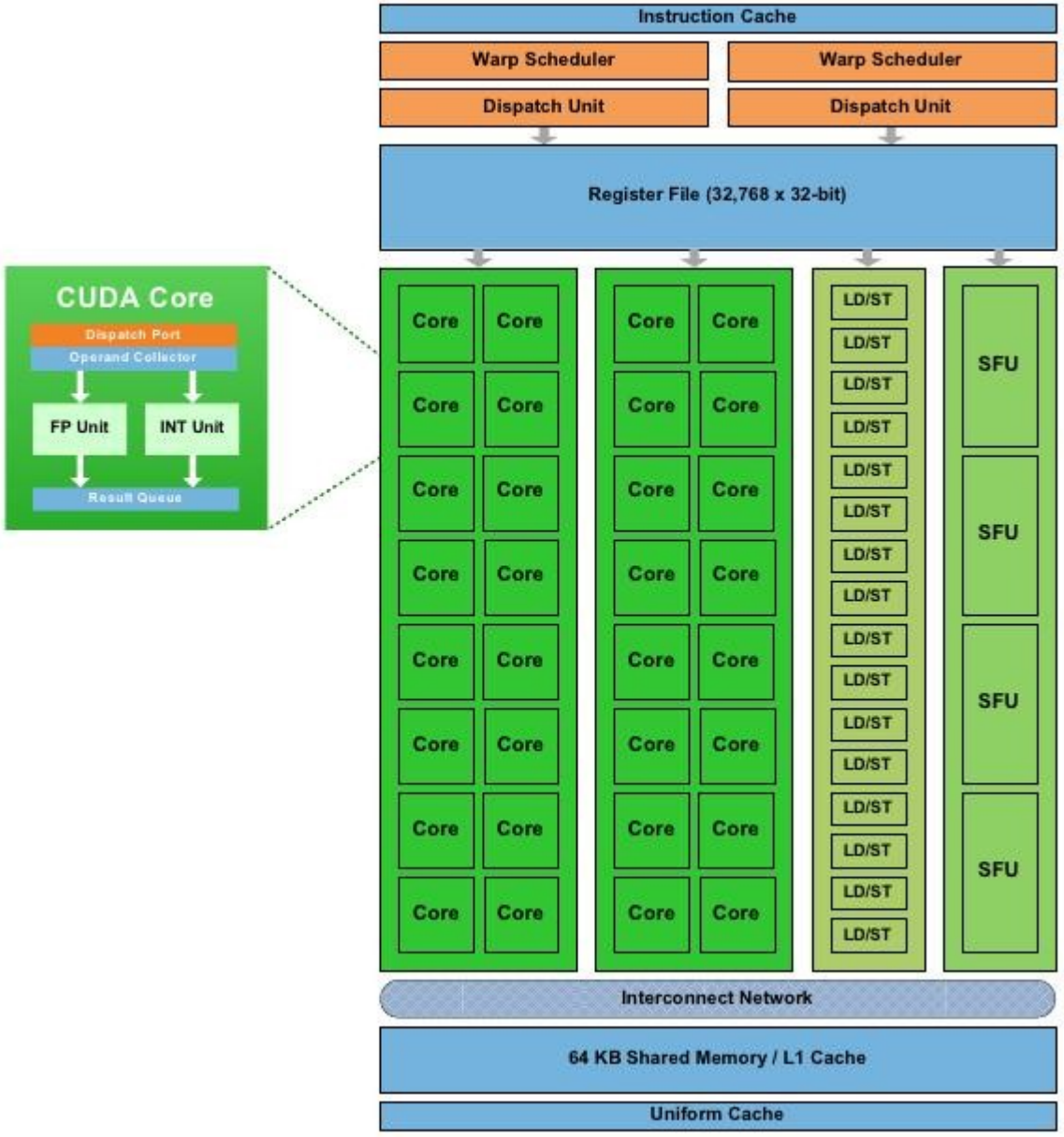


SM: Streaming Multiprocessor; SP: Streaming Processor; ROP: Raster Operation Processor

Nvidia G80

- VRAM or global memory: 80 GB/s.
  - Max dimension: 512 MB
- Shared memory: 16 KB (1KB blocks) 400 times faster then VRAM.
- Registers or private memory: 600 times faster than VRAM

Nvidia Fermi 2011



Nvidia architecture

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

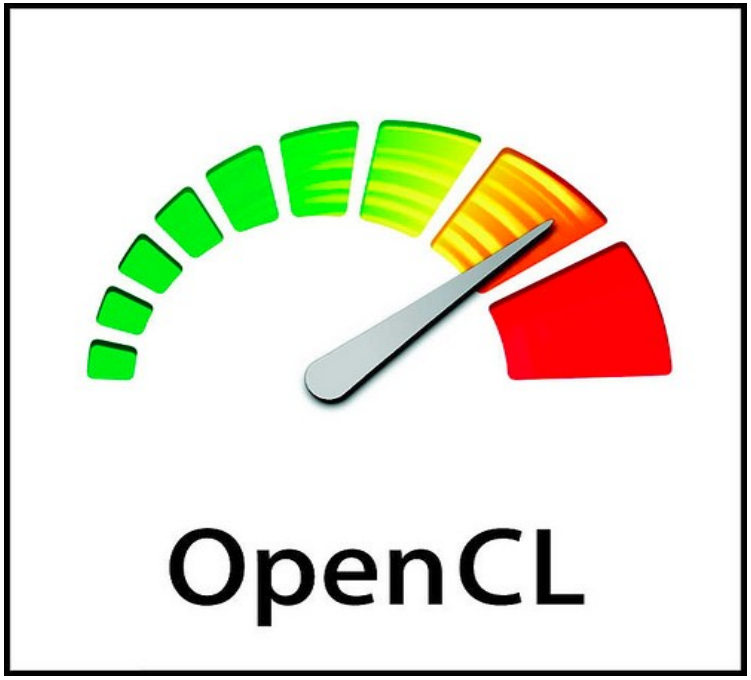
For details: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`

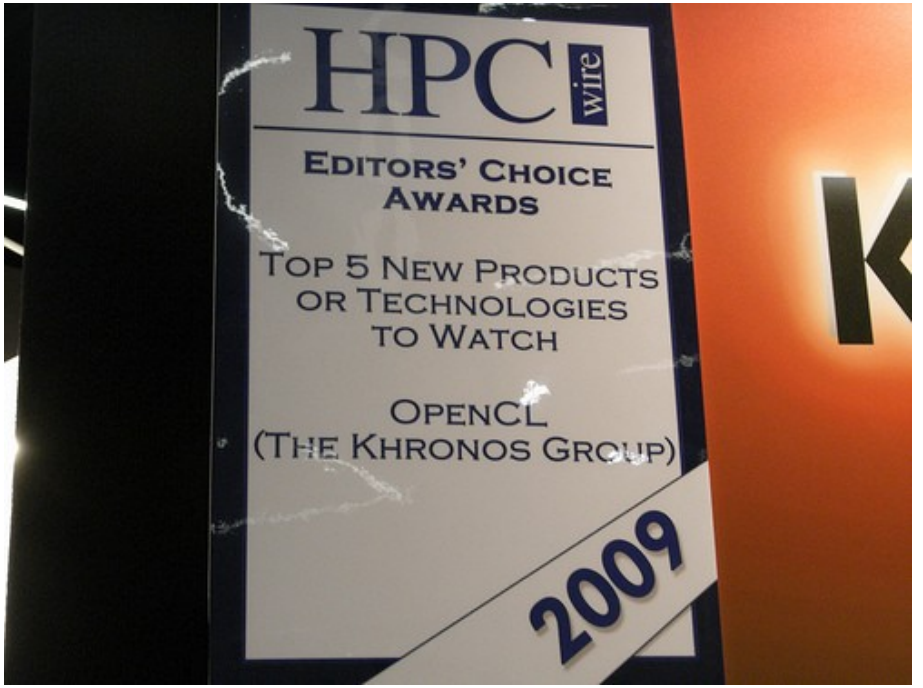
Instructions execution

- The instruction execution is organized in Threads.
- Each SM creates, manages, schedules and executes threads in groups of 32 threads named **wraps**.
- The threads of the same wrap start from the same program address but evolve independently
- The maximum in efficiency occurs when all threads of a wrap have the same path (no branch).
- Nvidia calls such architecture SIMT (Single Instruction Multiple Thread

Details in OpenCL Programming Guide for the CUDA Architecture vers 3.1.

OpenCL by Kronos Group



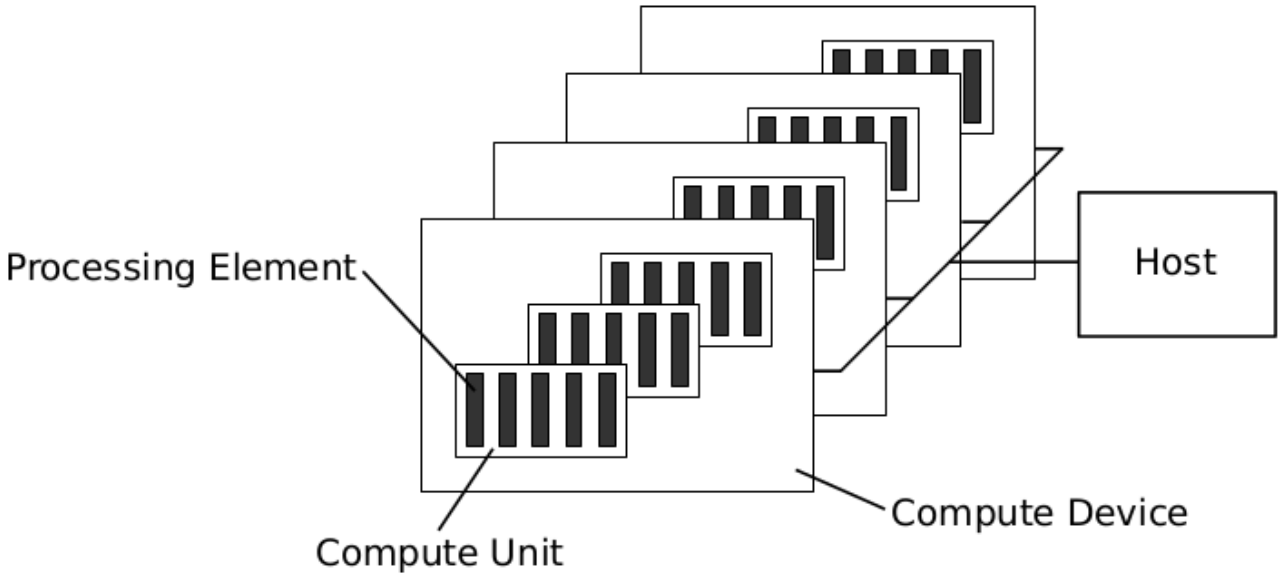


- Multi/Many Core Heterogeneous Computing Standard
- Runs on several devices (CPU, GPU, DSP, etc)
- Cross vendor (nVidia, AMD, Intel, etc)
- Portable (Linux, Windows, MacOS)

OpenCL architecture

- **Platform model**: abstraction of computing devices managed by a single host
- **Execution model**: defines the instructions set to be executed by the OpenCL devices (kernel) and the instructions initializing and controlling the kernels' execution (host program).
- **Memory model**: defines the memory objects, types of memory and how the host and the devices access them.
- **Programming model**: defines the type of parallel execution performed (on data or on tasks).
- **Framework model**: set of APIs and C99 extensions to implement host and kernel programs.

The Platform model

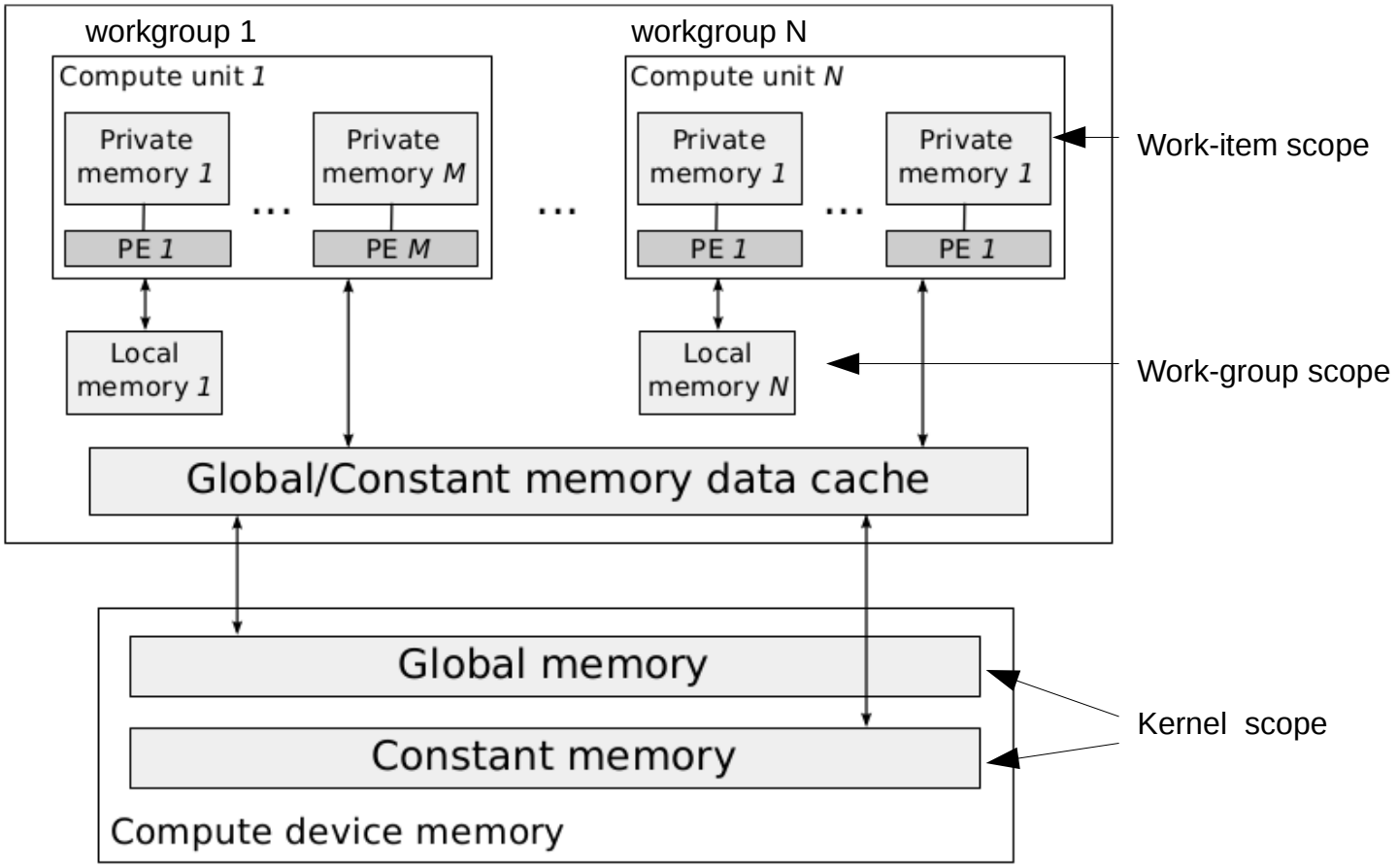


The platform model defines the roles of the host and the devices and provides an abstract hardware model for devices

The execution model

- **Host program**: set of instructions which initialize and manage the execution environment of the Compute Device
- **Kernel program**: set of instructions executed by the Compute Devices
- The Host prepares the various kernels' execution
- Each Compute Device executes the kernel
- Calculations are made by the **Work-items** (which are grouped in Work-groups) each work item executes the same program on different data

The memory model



The Framework model

- Extensions to C99:
  - Vector data type
  - Image data type
  - Conformance to the IEEE-754 - IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)
  - Memory objects
- Limitations respect C99:
  - No recursion
  - No standard libraries

A case study: AES

- The Advanced Encryption Standard (AES) algorithm plays a big role in the current encryption communications and security technologies.
- Standard FIPS-197
- The algorithm has been developed by **Joan Daemen** and **Vincent Rijmen** that submitted it with the ”Rindael” codename.
- The algorithm is symmetric, iterate, block based.
- Data blocks of 128 bit, Keys of 128, 192 or 256 bits.
- Due to its characteristics, it can greatly benefit from a parallel implementation and in particular from a GPU implementation.

The AES algorithm

```
State = input
AddRoundKey ( State , RoundKey [ 0 ] )
for  r = 1 to rounds−1
     SubBytes ( State )
     ShiftRows ( State )
     MixColumns ( State )
     AddRoundKey ( State , RoundKey [ r ] )
end
SubBytes ( State )
ShiftRows ( State )
AddRoundKey ( State , RoundKey [ rounds ] )
output = State
```

each byte of the state is combined with the round key using bitwise XOR

a non-linear substitution step where each byte is replaced with another according to a lookup table

a transposition step where each row of the state is shifted cyclically a certain number of steps.

a mixing operation which operates on the columns of the state, combining the four bytes in each column

each byte of the state is combined with the round key using bitwise XOR

a non-linear substitution step where each byte is replaced with another according to a lookup table

a transposition step where each row of the state is shifted cyclically a certain number of steps.

each byte of the state is combined with the round key using bitwise XOR

Implementation

- Read input file (plain text or ciphered)
- Read AES parameters
- Transfer memory objects to device global memory
- Key expansion
- Perform kernel on the OpenCl device
- Transfer memory objects from device global memory

Performance tests

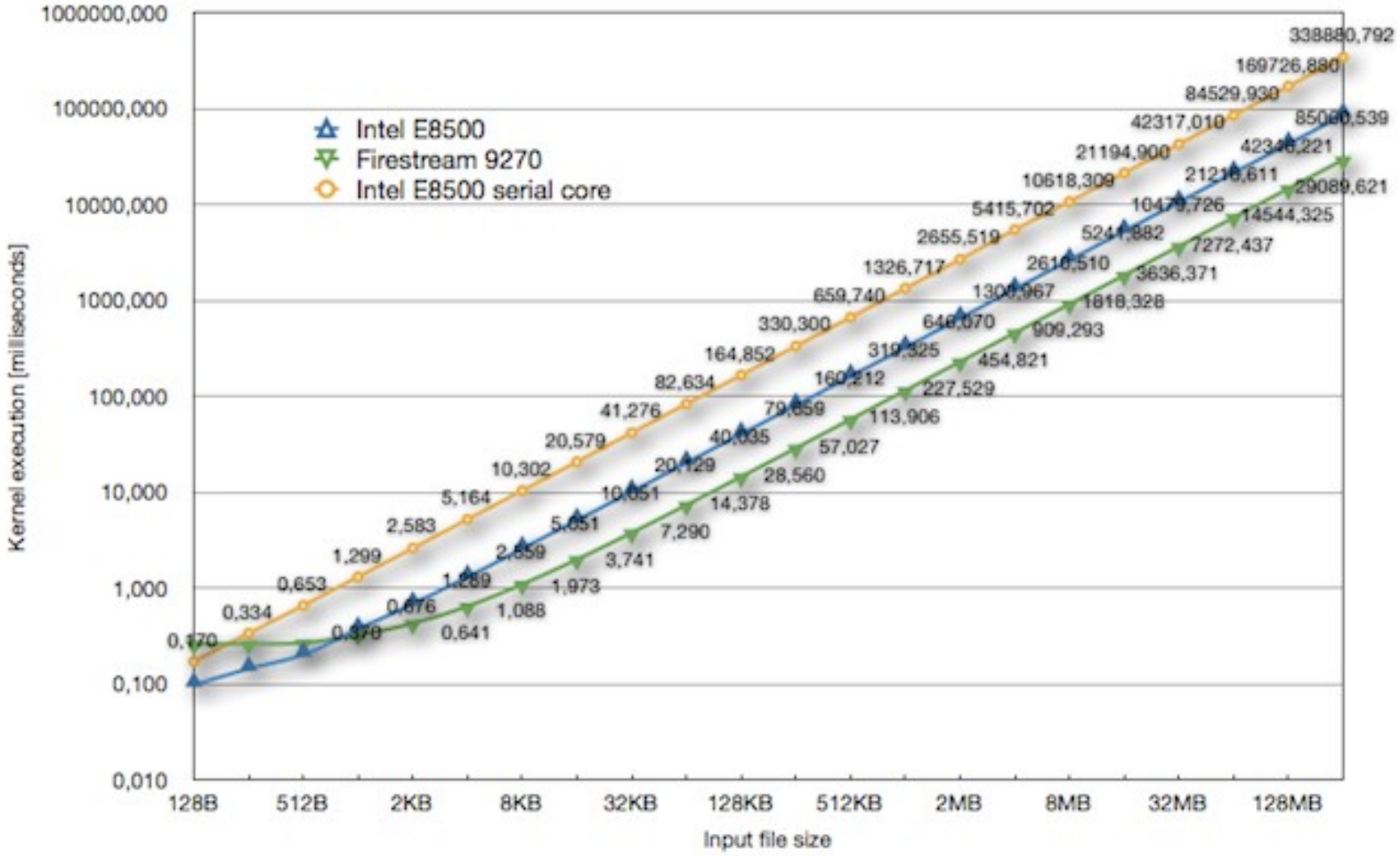
- Hardware description
  - ATI Firestream 9270 (vendor implementation of OpenCL)
  - Nvidia GeForce 8600 GT (vendor implementation of OpenCL)
  - CPU Intel Duo E8500 (AMD OpenCL driver)

Device **ATI RV770**
CL_DEVICE_TYPE: CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS: 10
CL_DEVICE_MAX_WORK_ITEM_SIZES: 256 / 256 / 256
CL_DEVICE_MAX_WORK_GROUP_SIZE: 256
CL_DEVICE_MAX_CLOCK_FREQUENCY: 750 MHz
CL_DEVICE_IMAGE_SUPPORT: 0
CL_DEVICE_GLOBAL_MEM_SIZE: 512 MByte
CL_DEVICE_LOCAL_MEM_SIZE: 16 KByte
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 256 Mbyte
CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_PROFILING_ENABLE

Device Intel(R) **Core(TM)2 Duo CPU E8500 @ 3.16GHz**
CL_DEVICE_TYPE: CL_DEVICE_TYPE_CPU
CL_DEVICE_MAX_COMPUTE_UNITS: 2
CL_DEVICE_MAX_WORK_ITEM_SIZES: 1024 / 1024 / 1024
CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
CL_DEVICE_MAX_CLOCK_FREQUENCY: 3166 MHz
CL_DEVICE_IMAGE_SUPPORT: 0
CL_DEVICE_GLOBAL_MEM_SIZE: 1024 MByte
CL_DEVICE_LOCAL_MEM_SIZE: 32 KByte
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 512 MByte
CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_PROFILING_ENABLE

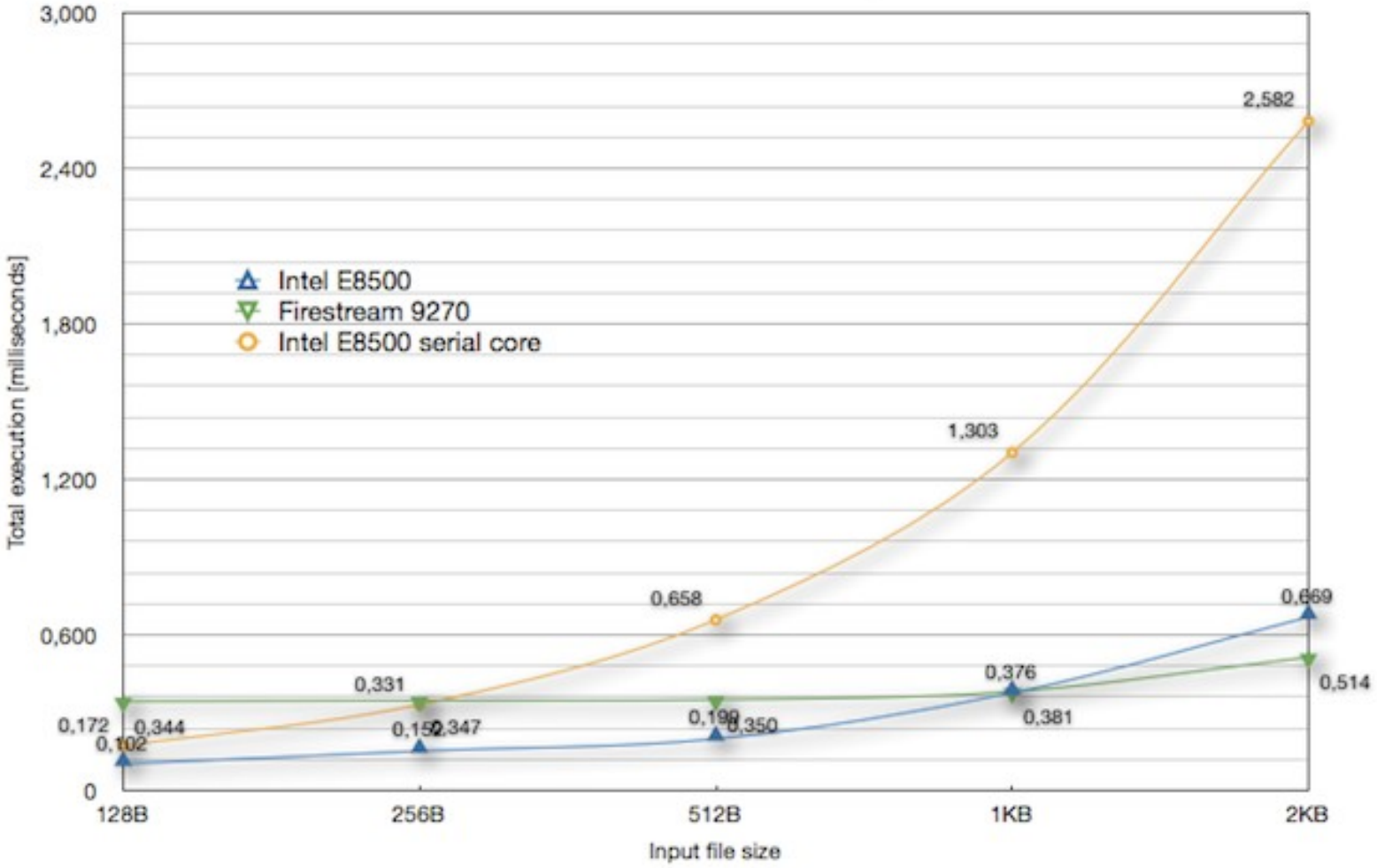
Device **GeForce 8600 GT**
CL_DEVICE_TYPE:
CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS: 4
CL_DEVICE_MAX_WORK_ITEM_SIZES:
512 / 512 / 64
CL_DEVICE_MAX_WORK_GROUP_SIZE:
512
CL_DEVICE_MAX_CLOCK_FREQUENCY:
1188 MHz
CL_DEVICE_IMAGE_SUPPORT: 1
CL_DEVICE_GLOBAL_MEM_SIZE: 255
MByte
CL_DEVICE_LOCAL_MEM_SIZE: 16
KByte
CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_PROFILING_ENABLE

Performance tests



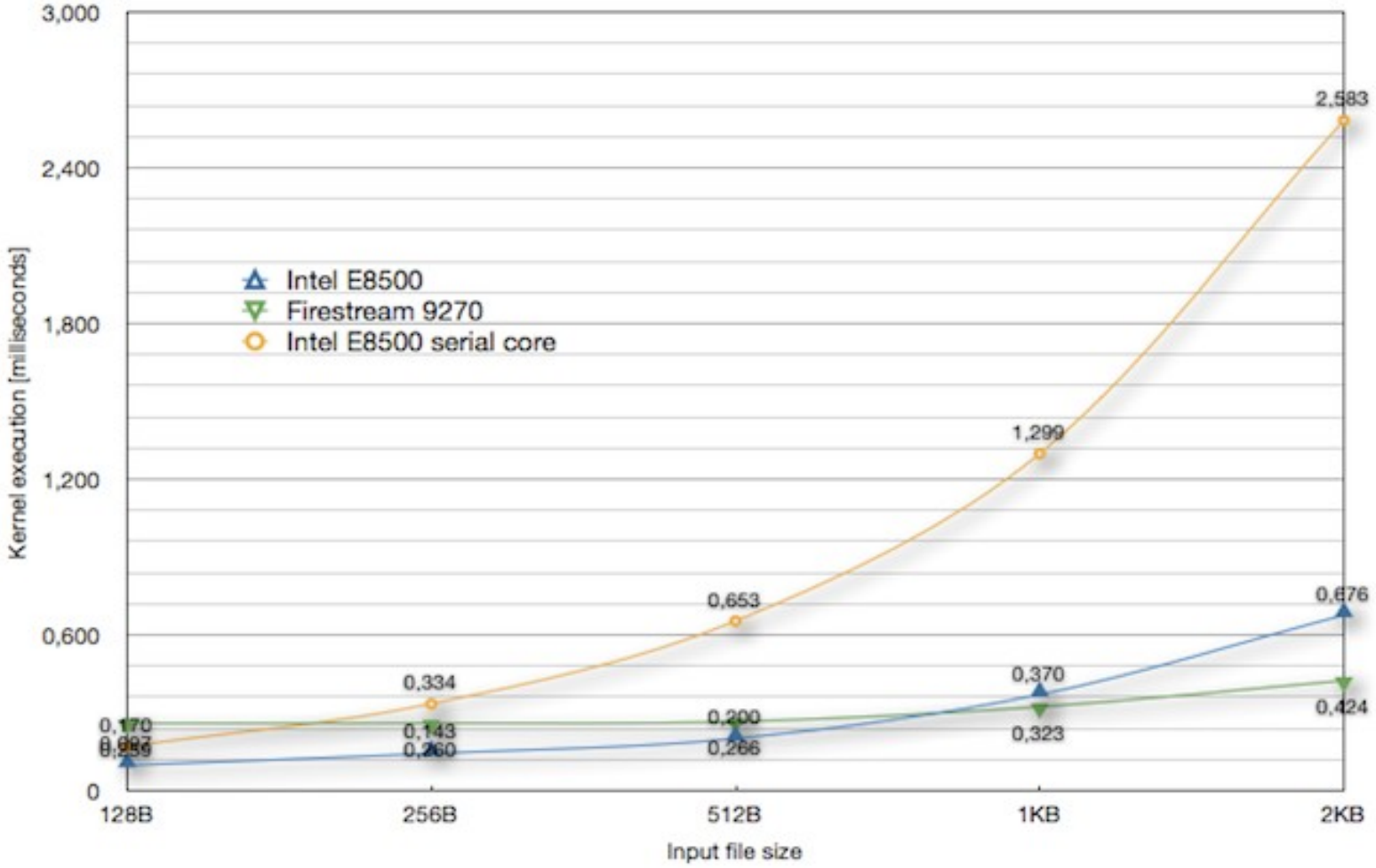
Performance tests

Ignoring the time spent to copy data in memory

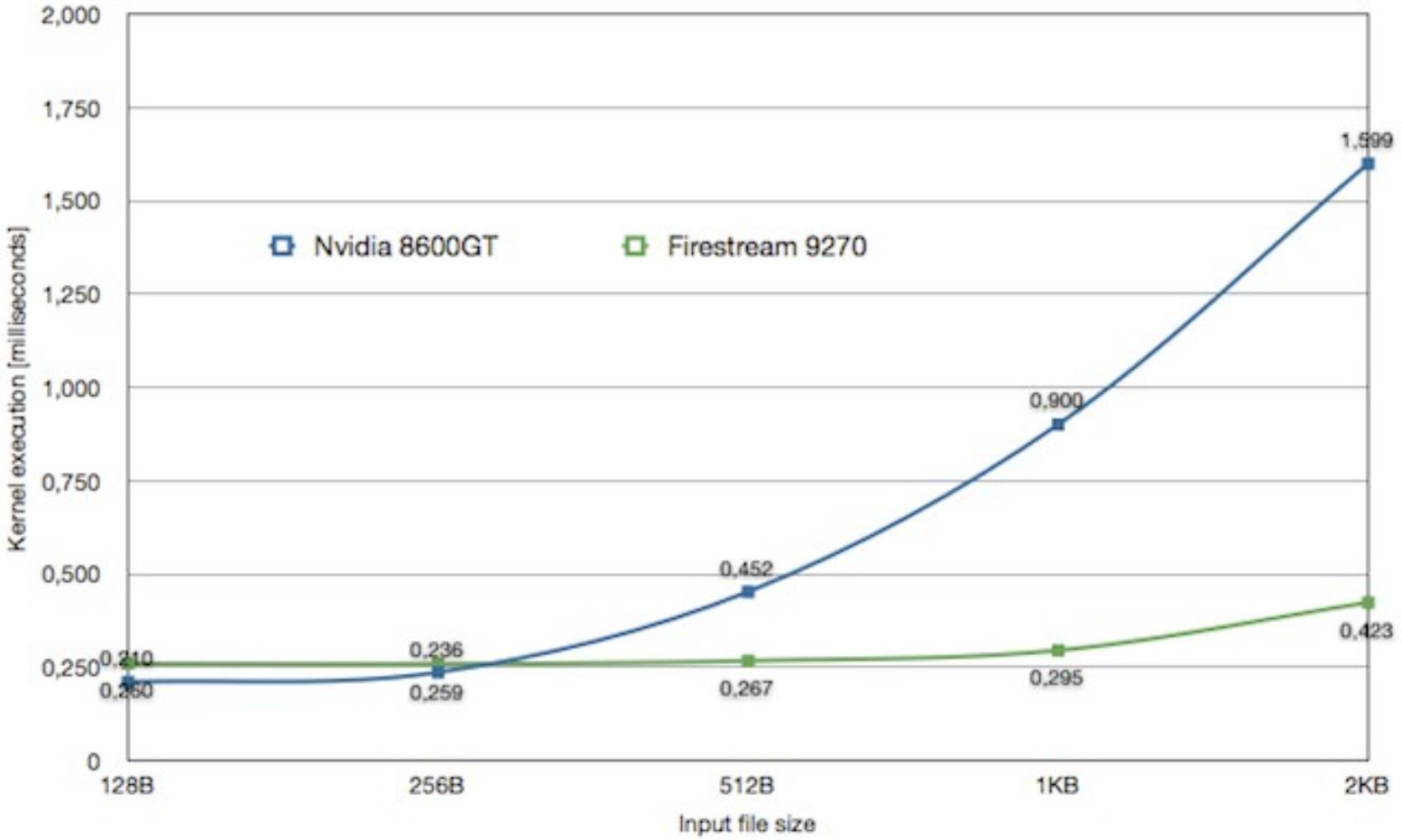


AES performance tests

Including the time spent to copy data in memory



AES performance tests



OpenSSL library

- FLOSS Security since 1998
- SSL TSL Tookit
- Projects based on openssl:
  - Apache (mod_ssl)
  - OpenVPN
  - SSH

  .....

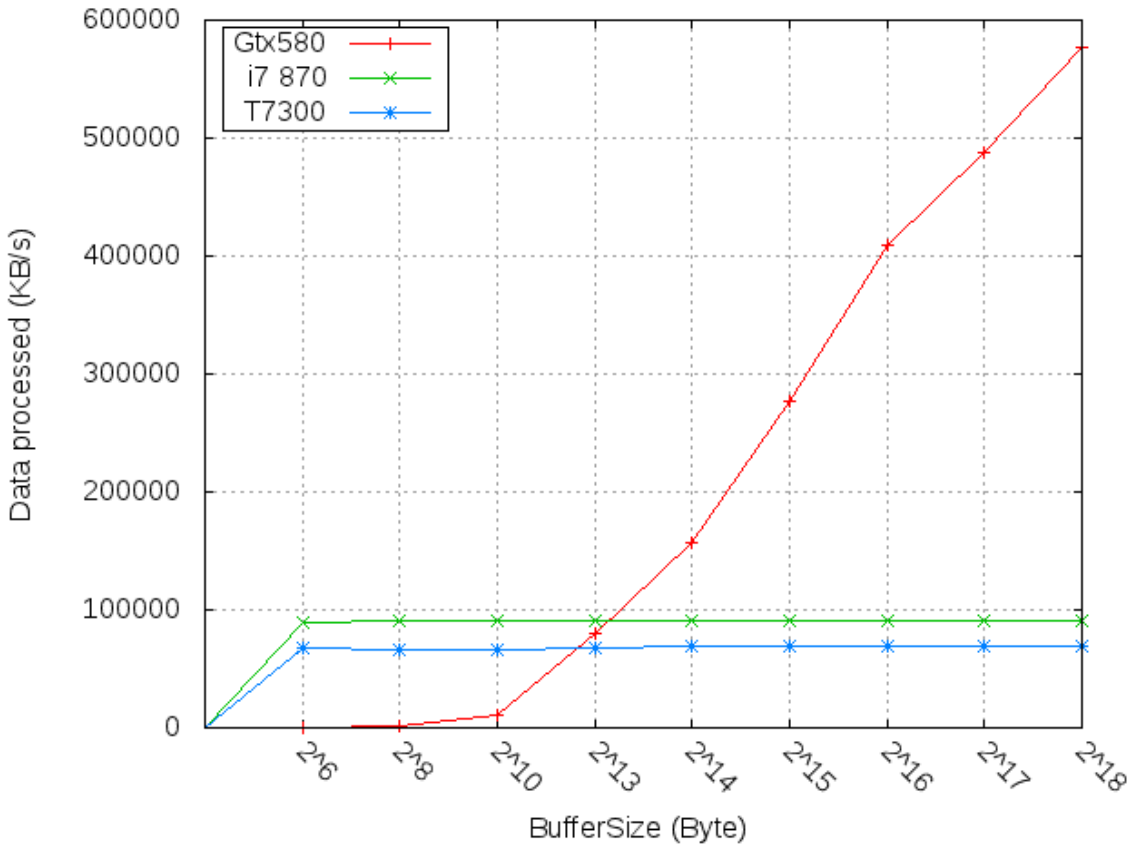
An OpenSSL Engine based on OpenCL has been created

Performance tests

- The performance tests have been carried out using the speed benchmark tool distributed with the openssl library.
- We used the following hardware:
  - Intel T7300 Core 2 **Duo** a 2.00GHz, 2 GB RAM DDR2
  - Intel i7 870 **Quad** Core (Hyperthreading) 3.0GHz, 4 GB di RAM DDR3
  - **Nvidia GTX 580** (16 Compute Units) 772MHz (512 Processing Element or Stream Processors at 1.5GHz) 1.5 GB VRAM DDR5
  - Two versions of the algorithm have been impleted: Sbox defined in the Constant Memory and the same installed on the Global Memory.

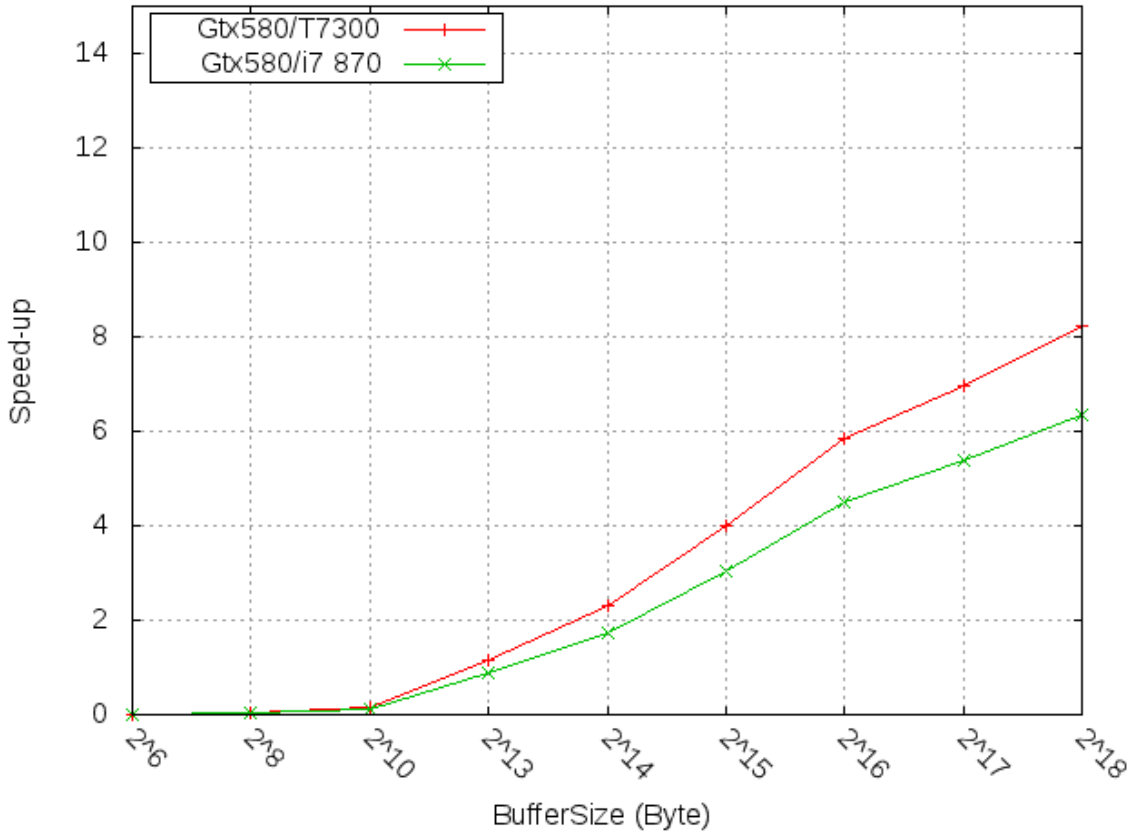
Performance tests

Data processed as a function of the packet size



Performance tests

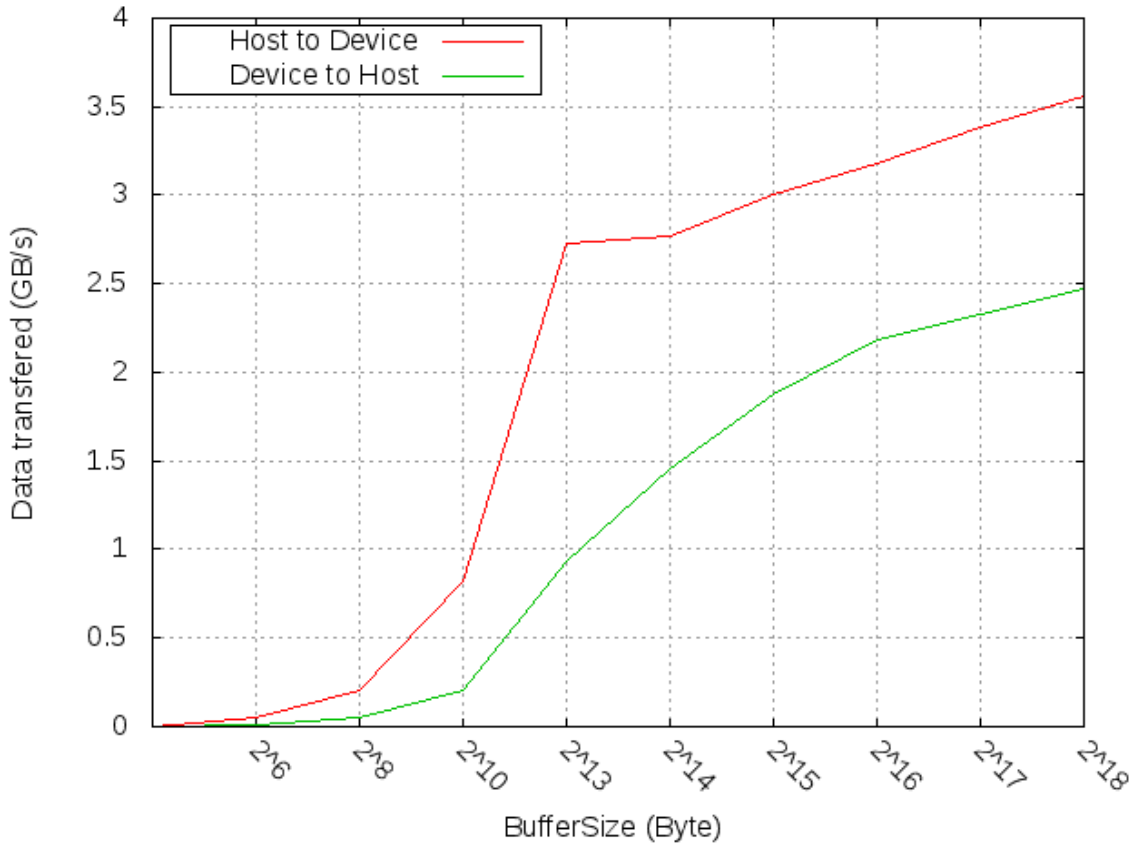
Speed-up of the same GPU running on 2 separate CPUs



Performance tests

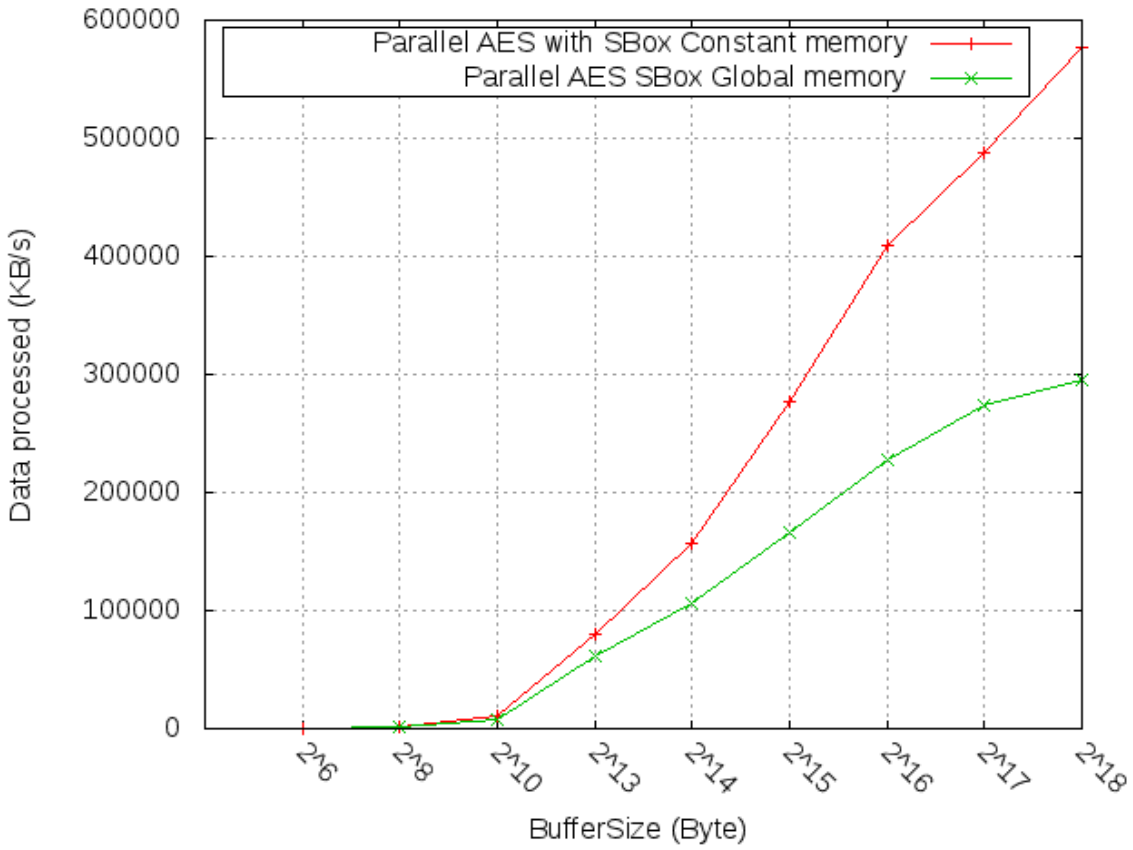
Measure of the data transfer from the host memory (RAM) to the device memory (VRAM) and vice-versa:

This is a measure of the overhead of the memory transfer



Performance tests

Speed-up of two variants of the algorithm
(Sbox defined in Constant Memory or in Global memory)



Scheduling issues

- The impressive amount of resources available through the GPGPU approach addresses important issues related to the efficiency of scheduling of modern operating systems in hybrid architectures.
- Usually it is up to the user decide the type of device to use. This is resulting in an inefficient or inappropriate scheduling process and to a not optimized usage of hardware resources.
- We are studying an H-system simulator to test scheduling algorithms for hybrid systems.

The simulator HPSim

- The model aims to simulate a H-system composed by:
  - a **set of processors** (CPUs) and graphics cards (GPUs) used as compute units to execute heterogeneous jobs
  - a **classifier** selecting the type of compute device (CPU or GPU)
  - a **scheduler** which implements the policy to be evaluated.

The simulator HPSim

- The proposed CPU-GPU simulation model is defined in terms of:
  - a set of **state variables** describing the system
    - Devices
    - Jobs
    - Queues
    - Scheduler
  - a **state transition function** which determines its progression through a finite set of discrete events

The simulator HPSim

- The simulator provides the following features:
  - Creation of the user-specified hardware in terms of number of CPUs and GPUs.
  - Generation of the system load, setting the number of jobs.
  - Tuning of the inter-arrival Job time.
  - Selection of the Job composition. It allows to specify the probability to generate a given number of Realtime, GPU User and CPU User Job.
  - Setting Classifier simulation.
  - Selection of qt strategy.

The simulator HPSim

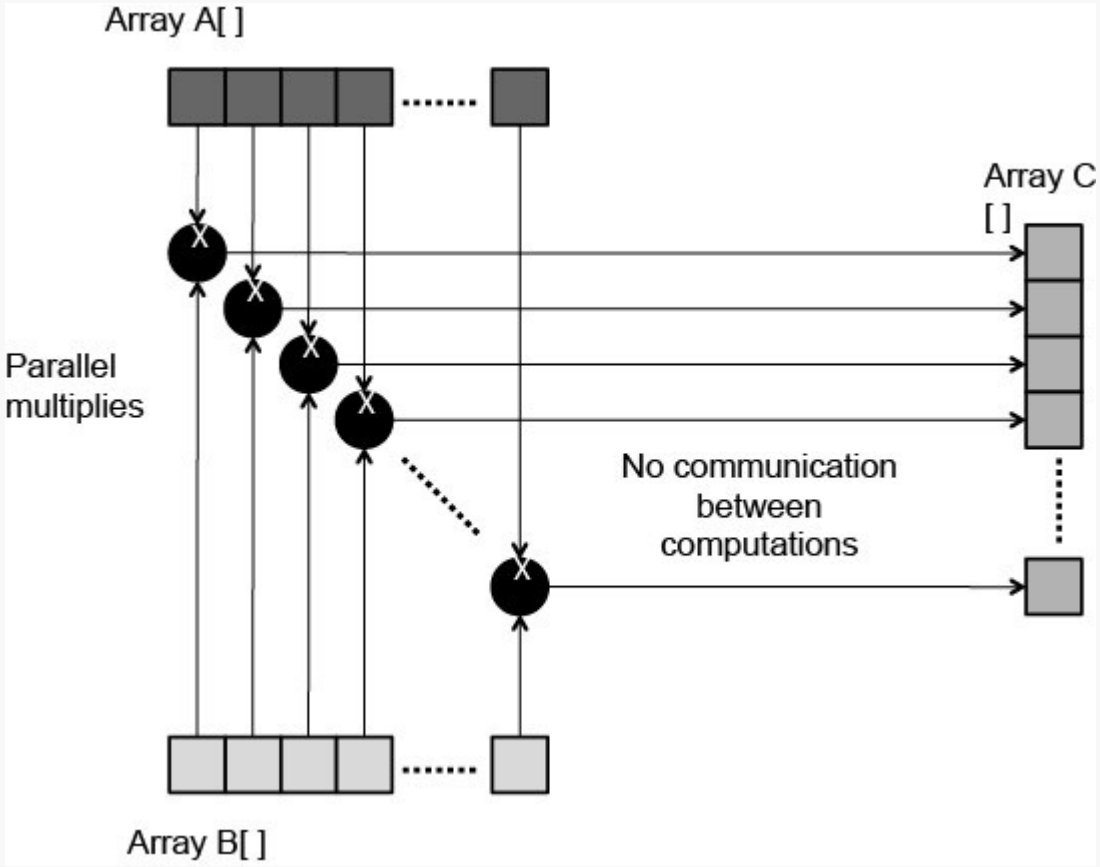
- We are focusing our work on three main aspects
  - We implemented a simple use-case considering a single non-preemptive priority queue. We are working to increase the possible cases.
  - We are carrying out a study of inter-arrival of real systems and the implementation of the linux scheduler (CFS) to validate the simulator.
  - We are adding new features to the simulator:
    - New scheduling policies
    - Implementation of a graphic interface
    - Automatic tools for the generation of charts for the analysis of the performance of the scheduling strategies.

Parallel computing

- Parallel Computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (i.e. in parallel)
- The degree of parallelism that can be achieved is dependent on the inherent nature of the problem at hand, and the skill of the algorithm or software designer is to identify the forms of parallelism present in the underlying problem.

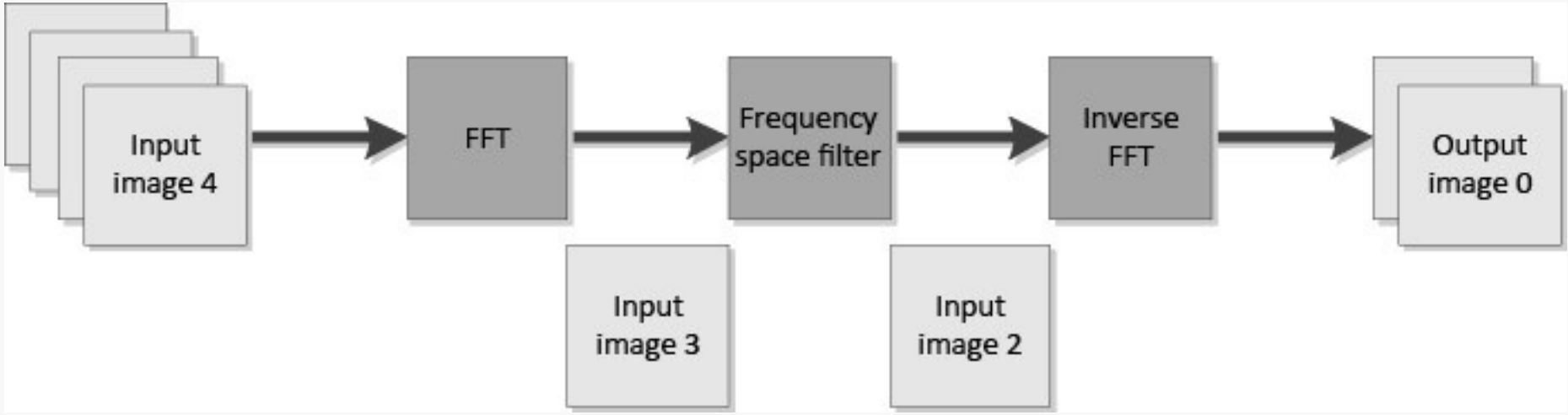
Multiplication of the elements of two arrays

- We carry out the multiplication of the elements of two vectors A and B, each with N elements, storing the result of each multiply in the corresponding array C



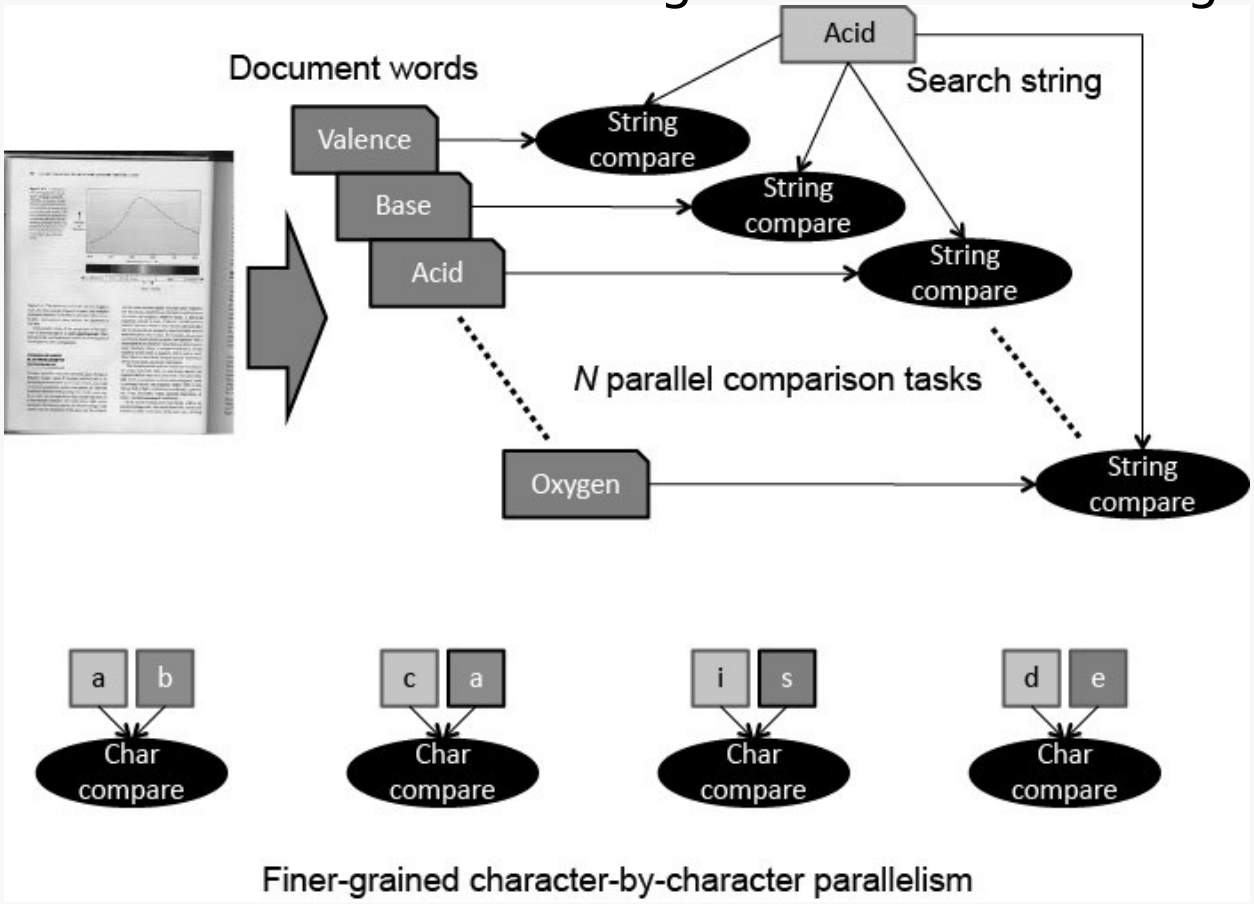
Filtering a series of images using FFT

- There is an high task parallelism on a series of tasks operating together in a pipeline to compute the overall result



Finding the occurrencies of a string in a text

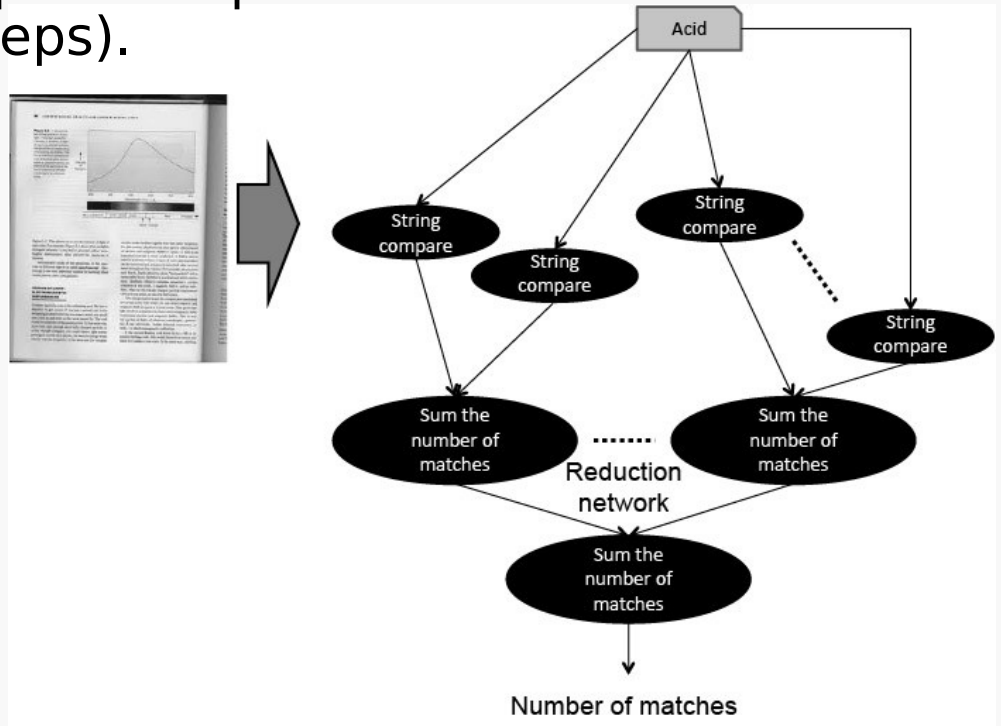
- We assume that the text body has been already parset in a set of N words. We divide the task of comparing the string against the N potential matches into N comparisons (i.e.: tasks), where each string of characters is matched against the text string.



Finer-grained character-by-character parallelism

Finding the occurrencies of a string in a text

- There is even further parallelism within single comparison task, where a matching on a character-by-character basis presents a finer-grained degree of parallelism. We observe both data-level parallelism and task-level parallelism.
- Once the number of matches is determined, we need to accumulate them to determine the total number of occurrences. We can again exploit the parallelism in the “reduction tree” (required logN steps).

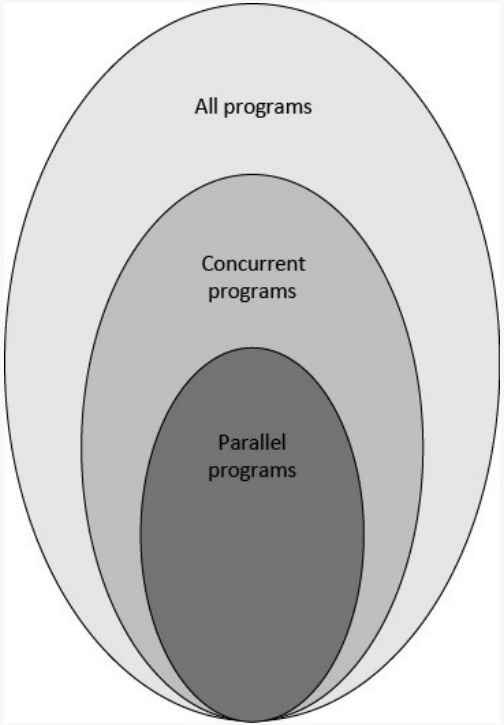


Concurrency

- Concurrency is concerned with two or more activities happening at the same time.
- We find concurrency in the real word every time we are thinking to something while doing something else with one's hands.
- When talking about concurrency in computer programming, we mean a single system performing multiple tasks independently.
- Although it is possible that concurrent tasks may be executed at the same time (i.e. in parallel) this is not a requirement.

Parallelism

- Parallelism is concerned with running two or more activities in parallel with the explicit goal of increasing the overall performances.
- Parallel programs must be concurrent, but concurrent programs need not be parallel.



Threads

- A running program may consist of multiple sub-programs that maintain their own independent control flow and that are allowed to run concurrently. These subprograms are defined as *threads*.
- Communication between threads is via updates and access to memory appearing in the same address space.
- Each thread has its own pool of local memory (variables), but all threads see the same set of global variables.
- A simple analogy may be the main program that includes a set of subroutines

Threads

- Threads communicate with each other through global memory. This can require synchronization constructs to ensure that more than one thread is not updating the same global address.
- A memory consistency model is defined to manage load and store ordering.
- Mechanisms such as locks/semaphores are commonly used to control access to shared memory that is accessed by multiple tasks.
- There is a significant cost to supporting a fully consistent shared memory model in hardware. Shared buses become bottlenecks in the design.

Message-passing communication

- The message-passing communication model enables explicit intercommunication of a set of concurrent tasks that may use memory during computation.
- Tasks exchange data through communication by sending and receiving explicit messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.
- The programmer is responsible for explicitly managing communications between tasks.

Different Grains of Parallelism

- In parallel computing, granularity is a measure of the ratio of computation to communication.
- Periods of computation are typically separated from periods of communication by synchronization events.
- The grain of the parallelism is constrained by the inherent characteristics of the algorithms constituting the application.
- It is important that the parallel programmer selects the right granularity in order to rip the full benefits of the underlying platform, because choosing the right grain size can help to expose additional degree of parallelism.

Data sharing and synchronization

- If two applications do not share any data, they can run concurrently and even in parallel.
- If halfway through the execution of one application is generated a result that will be subsequently rwquired by the second application, then we have to introduce some form of synchronization into the system, and parallel execution becomes impossible.
- When running concurrent software data sharing and synchronization play a critical role.
- Explicit synchronization primitives such as barriers or locks may be used.

The OpenCL specification

- The OpenCL specification is defined in 4 parts, called models:
  - Platform model: specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices). It defines an abstract hardware model that is used by programmers when writing OpenCL C functions (called kernels) the execute on the devices.
  - Execution model: defines how the OpenCL environment is configured on the host and how kernels are executed on the device. This includes defining a concurrency model used for kernels execution on devices.

The OpenCL specification

- Memory model: defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies, although this has non limited adoptability by other accelerators
- Programming model: defines how the concurrency model is mapped to physical hardware

Addition of the elements of two arrays

Serial code:

```
void vecadd(int *C, int *A, int *B, int N)
   {
      for (int i=0; i<N; i++) {
         C[i] = A[i] + B[i];
      }
   }
```

OpenCL Data Parallel

```
__kernel void vecadd(
    __global int *C,
    __global int *A,
    __global int *B){
       int tid = get_global_id(0); //OpenCL intrinsic function
       C[tid] = A[tid] + B[tid];
    }
```

Multiplication of the elements of two arrays

Serial code:

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

OpenCL Data Parallel

```
kernel void
    dp_mul( global const float *a,
            global const float *b,
            global float *c)
        {int id = get_global_id(0);
            c[id] = a[id] * b[id];
                    } // execute over “n” work-items
```

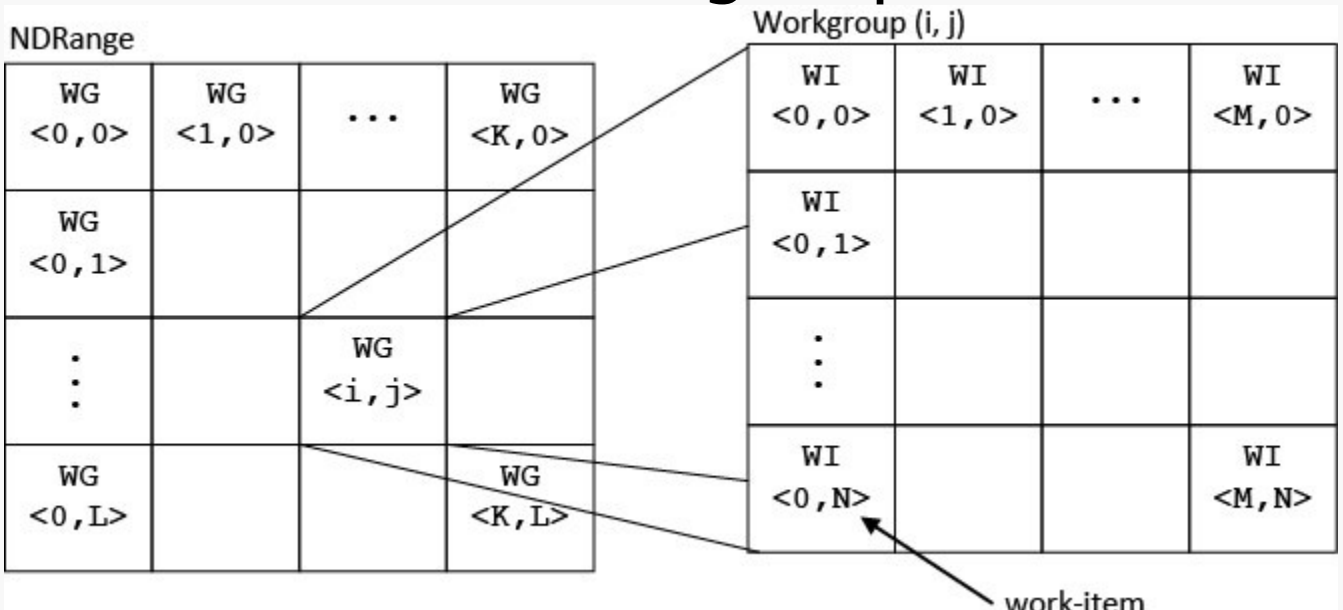
ND Range

- When a kernel is executed, the program specifies the number of work-items that should be created as an n-dimensional range (NDRange). A NDRange is a one-, two- or three-dimensional index space of work-items that will often map to the dimensions of either the input or the output data.
- The dimensions of the NDRange are specified as an N-element array of type size_t, where N represents the number of dimensions used to describe the work-items being created:

```
size_t indexSpaceSize[3] = [1024, 1, 1];
```

Work items

- Achieving scalability comes from dividing the work-items of an NDRange into smaller, equally sized workgroups. An index space with N dimensions requires workgroups to be specified using the same N dimensions; thus a three-dimensional index space requires three-dimensional workgroups.



Work items

- Work-items within a workgroup have a special relationship with one other: they can perform barrier operations to synchronize and they have access to a shared memory address space.
- Because workgroup sizes are fixed, this communication doesn't have a need to scale and hence does not affect scalability of a large concurrent dispatch.
- For the vector addition example, the workgroup size might be specified as

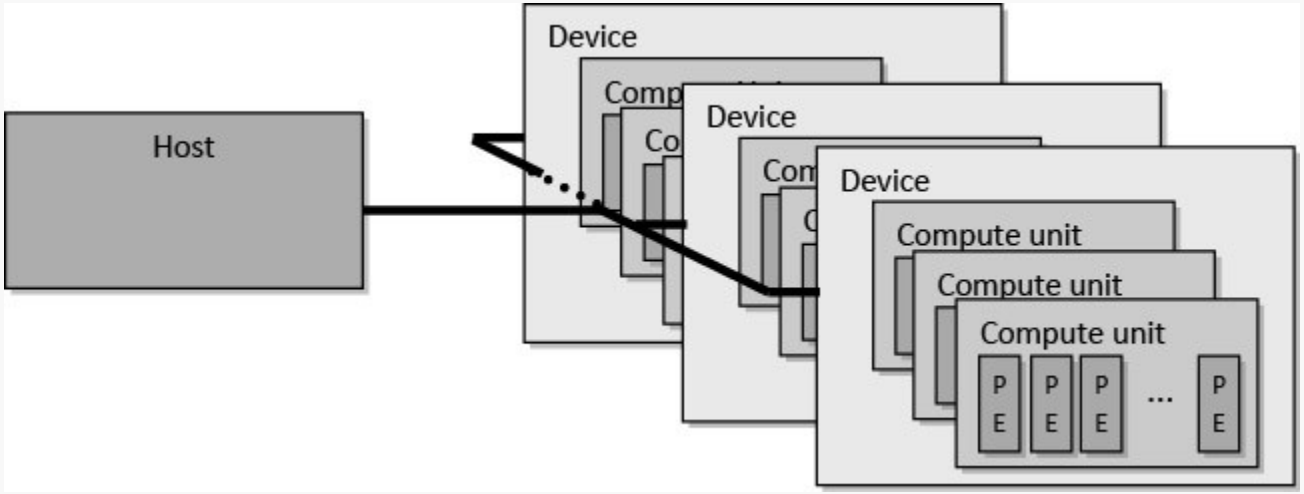
```
size_t workGroupSize[3] = [64, 1, 1];
```

Work items

- If the total number of work-items per array is 1024, this resulting in creating 16 workgroups (1024 work-items/(64 work-items per worksgroup) = 16 workgroup)
- OpenCL requires that the index space sizes are evenly divisible by the workgroup size in each dimension
- For programs such as vector addition in which work items behave independently (even within a workgroup), OpenCL allows the local workgroup size to be ignored by the programmer and generated automatically by the implementation. In this case the developer will pass NULL instead.

Platform and devices

- The OpenCL platform model defines the roles of the host and devices and provides an abstract hardware model for devices.
- A device is composed by a set of compute units, with each compute unit functionally independent from the rest. Compute Units are further divided into processing elements.



Platform and devices

- The platform device model closely corresponds to the hardware model of some GPUs.
- The API function `clGetPlatformIDs()` is used to discover the set of available platform for a given system

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

Contexts

- Before a host can request that a kernel be executed on a device, a context must be configured on the host that enables it to pass commands and data to the device.
- In OpenCL a context is an abstract container that exists on the host. A context coordinates the mechanism for host-device interaction, manages the memory objects that are available on the device, and keeps track of the program and kernels that are created for each device.
- The API function to create a context is **`clCreateContext()`**

  The `properties` argument is used to restict the scope of the context

Contexts

- Limiting the the context to a given platform allows the programmer to provide the context for multiple platforms and fully utilize a system comprising resources from a mixture of vendors.
- Next, the number and IDs of the devices that the programmer wants to associate with the context must be supplied.

```
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)
```

User callback provided by the user to report sdditional error informations that may be generated through its lifetime

Contexts

- The OpenCL specification also provides an API call that alleviates the need to build a list of devices. **`clCreateContextFromType()`** allows a programmer to create a context that automatically includes all devices of the specified type (e.g. CPUs, GPUs, and all devices).
- After creating a context the function **`clGetContextInfo()`** can be used to query information such as the number of devices present and the device structures.
- In OpenCL, the process of discovering platforms and devices and setting up a context is tedious. However, after code to perform these steps is written once, it can be reused for almost any project.

Command Queues

- Communication with a device occurs by submitting commands to a *command queue*.
- The command queue is the mechanism that the host uses to request action by the device.
- Once the host decides which device to work with and a context is created, one command queue needs to be created per device (each command queue is associated with only one device).
- Whenever the host needs an action to be performed by a device, it will submit commands to the proper command queue.

Command Queues

- The API **`clCreateCommandQueue()`** is used to create a command queue and to associate it to a device:

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

- Any API that specifies host-device interaction will always begin with *clEnqueue* and require a command queue as a parameter. For example the **`clEnqueueReadBuffer()`** command requests that the device send data to the host, and **`clEnqueueNDRangeKernel()`** requests that a kernel is executed on the device.

Events

- Any operation that enqueues a command into a command queue--that is, any API call that begins with `clEnqueue`—produces an *event*.
- Events have two main roles in OpenCL:
  - Representing dependencies
  - Providing mechanisms for profiling
- In addition to producing event objects, API calls that begins with `clEnqueue` also take a “wait list” of events as parameters. A `clEnqueue` call will block until all events on its wait list have completed.
- By generating an event for one API call and passing it as an argument to a successive call, OpenCl allows us to represent dependencies.

Memory objects

- OpenCL applications often work with large arrays of multidimensional matrices. These data need to be physically present on a device before execution can begin.
- In order for data to be transferred to a device, it must first be encapsulated as a *memory object*. OpenCL defines two types of memory objects: *buffers* and *images*.
- Buffers are equivalent to arrays in c, created using malloc(), where data elements are stored contiguously in memory
- Images are designed as opaque objects, allowing for data padding and other optimizations that may improve performance on devices.
- A memory object is valid only within a sigle context.

Memory Objects

Buffers

- It may help to visualize a memory object as a pointer that is valid on a device. This is similar to a call to `malloc` in C, or a C++ `new`'s operator.
- The API function clCreateBuffer() allocates the buffer and returns a memory object:

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

- Creating a buffer requires supplying the size of the buffer and a context in which the buffer will be allocated; it is visible to all devices associated with the context.

Memory Objects

- Optionally, the caller can supply flags that specify that the data is read-only, write-only, read-write. Other flags allow to specify additional options for creating and initializing a buffer. One simple option is to supply a host pointer with data used to initialize the buffer.
- Data contained in host memory is transferred to and from an OpenCL buffer using the command **`clEnqueueWriteBuffer()`** and **`clEnqueueReadBuffer()`**.
- If a kernel that is dependent from such a buffer is executed on a GPU, the buffer may be transferred to the device. The buffer is linked to a context, not to a device, so it is the runtime that determines the precise time the data is moved.

Memory Objects

- The API calls for reading and writing to buffers are very similar.

```
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)
```

- Similar to other enqueue operations, reading or writing a buffer requires a command queue to manage the execution schedule. The enqueue function requires the buffer, the number of bytes to transfer, and an offset within the buffer.
- The `blocking_write` option should be set to `CL_TRUE` if the transfer into an OpenCL buffer should complete before the function returns – will block until operation has completed.

Memory Objects

Images

- Images are a type of OpenCL memory object that abstract the storage of physical data to allow for device-specific optimization. They are not required to be supported by all OpenCl devices, and an application is required to check, using **`clGetDeviceInfo()`** if they are supported or not.
- Unlike buffers, images cannot be directly referenced as if they were arrays.
- Furthermore, adjacent data elements are not guaranteed to be stored contiguously in memory.
- The purpose of using images is to allow the hardware to take advantage of spatial locality and to utilize hardware acceleration available on many devices.

Memory Objects

- The elements of an image are represented by a format descriptor **`cl_image_format`**. The format descriptor specifies how the image elements are stored in memory based on the concept of channel.
- The channel order specifies the number of elements that make up an image element (up to four elements, based on the traditional RGBA pixels) and the channel type specifies the size of each element.
- These elements can be sized from 1 to 4 bytes and in various different formats (i.e. integer of floating point).
- Creating an OpenCL image is done using the command **`clCreateImage2D()`** or **`clCreateImage3D()`**
- In addition are required the height and the width of the image (and the depth in the 3D case). Furthermore the image pitch (N. of Bytes between the start of one image row and the start of the next) may be supplied if initialization data is provided.

Memory Objects

```
cl_mem clCreateImage2D(cl_context context,
                       cl_mem_flags flags,
                       const cl_image_format *image_format,
                       size_t image_width,
                       size_t image_height,
                       size_t image_row_pitch,
                       void *host_ptr,
                       cl_int *errcode_ret)
```

- There are also additional parameters when reading or writing an image. Read or Write operations take three element origin (similar to the buffer offset) that defines the location within the image that the transfer will begin and another three-element region parameter that defines the extent of the data that will be transferred.
- Within a kernel images are accessed with built-in functions specific of the data type, i.e.: **`read_imagef()`** for floats and **`read_imageui()`** for unsigned integers.

Flush and finish

- The flush and finish commands are two different types of barrier operations for a command queue.
- The **`cl_Finish`**`()` function blocks until all the commands in a command queue have completed; its functionality is synonymous with a synchronization barrier.
- The **`cl_Flush`**() function blocks until all the commands in a command queue have been removed from the queue.
- This means that the commands will definitely be in-flight but will not necessarily have completed.

```
cl_int clFlush(cl_command_queue command_queue);
cl_int clFinish(cl_command_queue command_queue);
```

Creating an OpenCL Program Object

- OpenCL C code, written to run on an OpenCL device, is called a *program*. A program is a collection of functions called kernels, where kernels are units of execution that can be scheduled to run on a device.
- OpenCL programs are compiled at runtime through a series of API calls. This runtime compilation gives the system an opportunity to optimize for a specific device.
- There is no need for an OpenCL application to have been prebuilt against the vendor (NVIDIA,AMD, Intel) runtimes.
- OpenCL software links only to a common runtime layer (called the ICD); all platform-specific SDK activity is delegated to a vendor runtime through a dynamic library interface.

Creating an OpenCL Program Object

- The process of creating a kernel is as follows:
  - The OpenCL C source code is stored in a character string. If the source code is stored in a file on a disk, it must be read into memory and stored as a character array
  - The source code is turned into a program object, **`cl_program`**, by calling **`clCreateProgramWithSource()`**
  - The program object is then compiled, for one or more OpenCL devices, with **`clBuildProgram()`**. If there are compile errors, they will be reported here.
- The precise binary representation used is very vendor specific. In the AMD runtime there are two main classes of devices: x86 CPUs and GPUs. For x86 CPUs **`clBuildProgram()`** generates x86 instructions that can be directly executed on the device. For the GPUs it will create AMD's GPU intermediate language (**IL**) a high-level intermediate language that represents a single work-item but that will be just-in-time compiled for a specific GPU's architecture later, generating a **ISA** (code for a specific instruction set architecture). NVIDIA uses a similar approach (calling it PTX).

Creating an OpenCL Program Object

- The advantage of using such an IL is to allow the GPU ISA itself to change from one device or generation to another in what is still a very rapidly developing architectural space.
- One additional feature of the build process is the ability to generate both the final binary format and various intermediate representations and serialize them (i.e.: write them out to disk).
- As with most objects, OpenCL provides a function to return information about program objects, **`clGetProgramInfo()`**. One of the flags to this function is **`CL_PROGRAM_BINARIES`**, which returns a vendor-specific set of binary objects generated by **`clBuildProgram()`**.
- In addition to **`clCreateProgramWithSource()`**, OpenCL provides **`clCreateProgramWithBinary()`**, which takes a list of binaries that matches its device list.
- The binaries are previously created using **`clGetProgramInfo()`**.

The OpenCL Kernel

- The final stage to obtain a **`cl_kernel`** object that can be used to execute kernels on a device is to extract the kernel from the **`cl_program`**. Extracting a kernel from a program is similar to obtaining an exported function from a dynamic library.
- The name of the kernel that the program exports is used to request it from the compiled program object. The name of the kernel is passed to **`clCreateKernel()`**, along with the program object, and the kernel object will be returned if the program object was valid and the particular kernel is found.
- Unlike calling functions in regular C programs, we cannot simply call a kernel by providing a list of arguments.
- Executing a kernel requires dispatching it through an enqueue function. Due both to the syntax of C language and to the fact that kernel arguments are persistent, we must specify each kernel argument individually using the function **`clSetKernelArgs()`**.

The OpenCL Kernel

- **`clSetKernelArgs()`** takes a kernel object, an index specifying the argument number, the size of the argument, and a pointer to the argument. When a kernel is executed this information is used to transfer arguments to the device.
- The type information in the kernel parameter list is then used by the runtime to unbox the data to its appropriate type.
- After any required memory objects are transferred to the device and the kernel arguments are set, the kernel is ready to be executed.
- Requesting that a device begin executing a kernel is done with the call:

```
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)
```

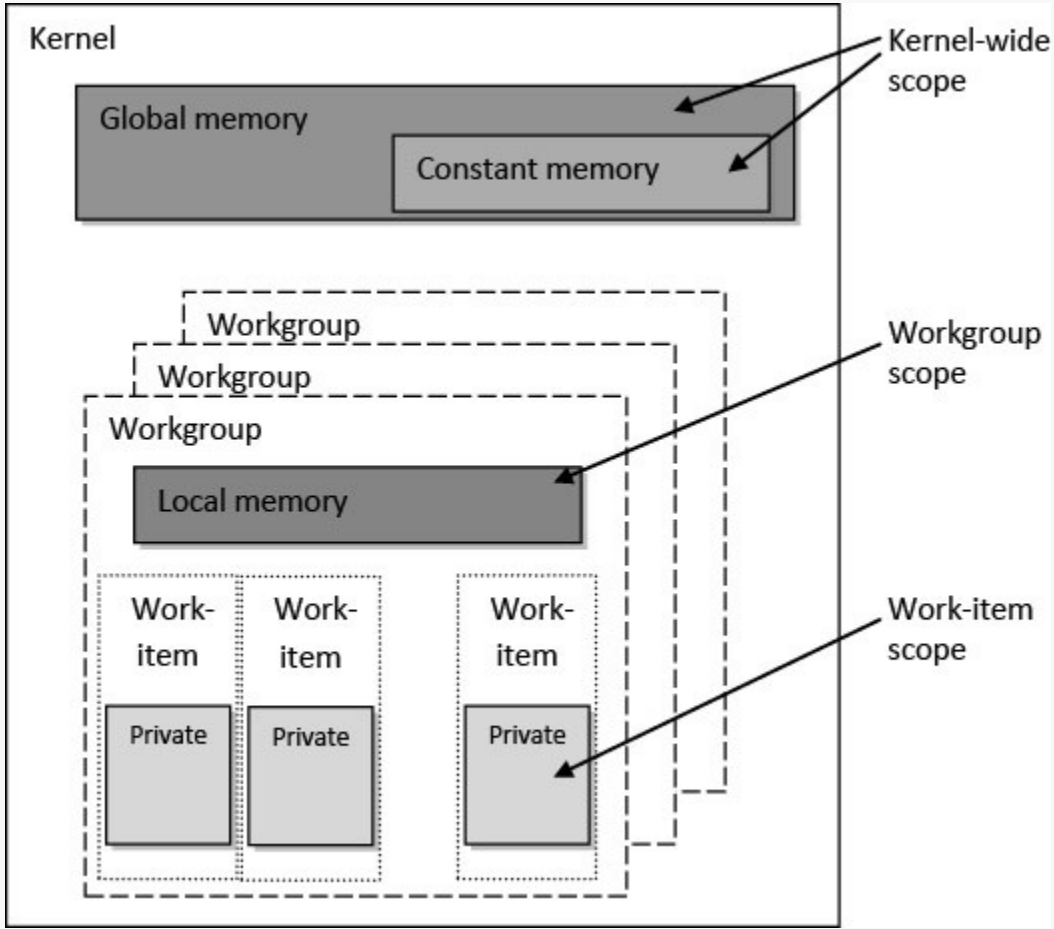
The OpenCL Kernel

- The **`work_dim`** parameter specifies the number of dimensions in which work-items will be created
- The **`global_work_size`** parameter specifies the number of work items in each dimension of the NDRange and **`local_work_size`** specifies the number of work-items in each dimension of the workgroups.
- The **`clEnqueueNDRangeKernel()`** call is asynchronous: it will return immediately after the command is enqueued in the command queue and likely before the kernel has even started execution.
- Either **`clWaitForEvents()`** or **`clFinish()`** can be used to block execution on the host until the kernel completes.

Memory Model

Memory subsystems vary greatly between computing platforms. For example. All modern CPUs support automatic caching, althoughmany GPUs do not. To support code portability, OpenCL's approach is to define an abstract memory model that programmers can target when writing code and vendors can map to their actual memory

These memory spaces are relevant within OpenCL programs. The keywords associated with each space can be used to specify where a variable should be created or where the data that it points to resides.



Memory Model

- **Global memory** is visible to all compute units on the device (similar to the main memory on a CPU-based host system). Whenever the data is transferred from the host to the device, the data will reside in global memory. Any data that is to be transferred back from the device to the host must also reside in global memory.
- The keyword **`__global`** is added to a pointer declaration to specify that data referenced by the pointer resides in global memory.
- For example in the OpenCL C code shown as an example (vector addition) **`__global float *A`**, the data pointed to by `A` resides in global memory (although we will see that A actually resides in private memory)

Memory Model

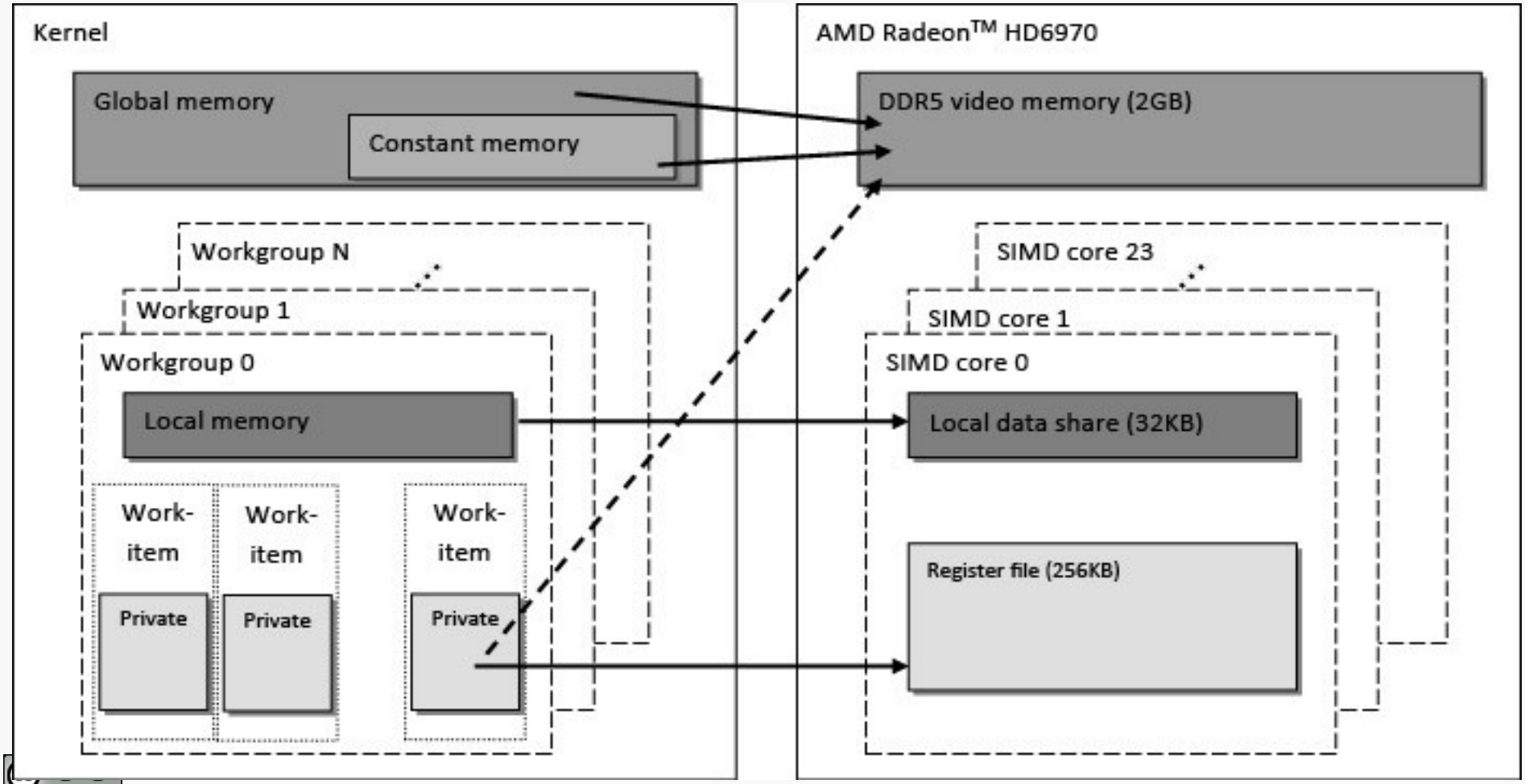
- **Constant memory** is designed for data where each element is accessed simultaneously by all work-items. Variables whose values never changes, also fall into this category.
- Constant memory is modelled as part of the global memory, so memory objects that are transferred to global memory can be specified as constant.
- Data is mapped to constant memory by using the **`__constant`** keyword.
- **Local memory** is a scratchpad memory whose address space is unique to each compute device. It is common for it to be implemented as on-chip memory, but there is no requirement that this be the case.
- Local memory is modelled as being shared by a workgroup. As such accesses may have much shorter latency and much higher bandwidth then global memory.

Memory Model

- Calling **`clSetKernelArg()`** with a size, but no argument, allow local memory to be allocated at runtime, where a kernel parameter is defined as a **`__local`** pointer (e.g.: **`__local float *shareData`**)
- Alternatively, arrays can be statically declared in local memory by appending the keyword **`__local`** (e.g.: **`__local float[64] sharedData`**), although this requires specifying the array size at compile time.
- Private memory is memory that is unique to an individual work-item. Local variables and non-pointer kernel arguments are private by default. In practice, these variables are mapped to registers, although private arrays and any spilled registers are usually mapped to an off-chip (i.e.: long latency) memory.

Memory Model

The memory spaces of OpenCL closely model those of modern GPUs:



Writing kernels

- OpenCL kernels are similar to C functions and can be thought of as instances of a parallel map operation.
- The function body, like the mapped function, will be executed once for every work-item created.
- Kernels begin with the keyword **`__kernel`** and must have a return type of `void`.
- The argument list is as for a C function with the additional requirement that the address space of any pointer must be specified.
- Buffers can be declared in global memory (**`__global`**) or constant memory (**`__constant`**).
- Images are assigned to global memory.
- Access qualifiers (**`__read_only`**, **`__write_only`**, **`__read_write`**) can also be optionally specified because they may allow for compiler and hardware optimizations.

Writing kernels

- When programming for OpenCL devices, particularly GPUs, performance may increase by using local memory to cache data that will be used multiple times by multiple work-items of the same workgroup.
- When developing a kernel, we can achieve this with an explicit assignment from a global memory pointer to a local memory pointer:

```
__kernel void cache(
    __global float* data,
    __global float* sharedData) {
    int globalId = get_global_id(0);
    int localId = get_local_id(0);
    //cache data to local memory
    sharedData[localId] = data[globalId];
    ……
}
```

Source code vector addition

```
// This program implements a vector addition using OpenCL

// System includes
#include <stdio.h>
#include <stdlib.h>

// OpenCL includes
#include <CL/cl.h>

// OpenCL kernel to perform an element-wise
// add of two arrays
const char* programSource =
  "__kernel                                          \n"
"void vecadd(__global int *A,                        \n"
"            __global int *B,                        \n"
"            __global int *C)                        \n"
"{                                                   \n"
"                                                    \n"
"   // Get the work-items unique ID                   \n"
"   int idx = get_global_id(0);                      \n"
"                                                    \n"
"   // Add the corresponding locations of            \n"
"   // 'A' and 'B', and store the result in 'C'.     \n"
"   C[idx] = A[idx] + B[idx];                        \n"
"}                                                   \n"
";

int main() {
    // This code executes on the OpenCL host

    // Host data
    int *A = NULL;  // Input array
    int *B = NULL;  // Input array
    int *C = NULL;  // Output array

    // Elements in each array
    const int elements = 2048;
```

Source code vector addition

```
// Compute the size of the data
size_t datasize = sizeof(int)*elements;

// Allocate space for input/output data
A = (int*)malloc(datasize);
B = (int*)malloc(datasize);
C = (int*)malloc(datasize);
// Initialize the input data
for(int i = 0; i < elements; i++) {
    A[i] = i;
    B[i] = i;
}

// Use this to check the output of each API call
cl_int status;

//---------------------------------------------------
// STEP 1: Discover and initialize the platforms
//---------------------------------------------------

cl_uint numPlatforms = 0;
cl_platform_id *platforms = NULL;

// Use clGetPlatformIDs() to retrieve the number of
// platforms
status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform
platforms =
    (cl_platform_id*)malloc(
       numPlatforms*sizeof(cl_platform_id));

// Fill in platforms with clGetPlatformIDs()
status = clGetPlatformIDs(numPlatforms, platforms,
           NULL);
```

Source code vector addition

```
//---------------------------------------------------------
// STEP 2: Discover and initialize the devices
//---------------------------------------------------------

cl_uint numDevices = 0;
cl_device_id *devices = NULL;

// Use clGetDeviceIDs() to retrieve the number of
// devices present
status = clGetDeviceIDs(
    platforms[0],
    CL_DEVICE_TYPE_ALL,
    0,
    NULL,
    &numDevices);

// Allocate enough space for each device
devices =
    (cl_device_id*)malloc(
        numDevices*sizeof(cl_device_id));

// Fill in devices with clGetDeviceIDs()
status = clGetDeviceIDs(
    platforms[0],
    CL_DEVICE_TYPE_ALL,
    numDevices,
    devices,
    NULL);
```

Source code vector addition

```
//---------------------------------------------------------
 // STEP 3: Create a context
 //---------------------------------------------------------

 cl_context context = NULL;

 // Create a context using clCreateContext() and
 // associate it with the devices
 context = clCreateContext(
     NULL,
     numDevices,
     devices,
     NULL,
     NULL,
     &status);

 //---------------------------------------------------------
 // STEP 4: Create a command queue
 //---------------------------------------------------------

 cl_command_queue cmdQueue;

 // Create a command queue using clCreateCommandQueue(),
 // and associate it with the device you want to execute
 // on
 cmdQueue = clCreateCommandQueue(
     context,
     devices[0],
     0,
     &status);
```

Source code vector addition

```
//------------------------------------------------------
// STEP 5: Create device buffers
//------------------------------------------------------

cl_mem bufferA;  // Input array on the device
cl_mem bufferB;  // Input array on the device
cl_mem bufferC;  // Output array on the device

// Use clCreateBuffer() to create a buffer object (d_A)
// that will contain the data from the host array A
bufferA = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    datasize,
    NULL,
    &status);

// Use clCreateBuffer() to create a buffer object (d_B)
// that will contain the data from the host array B
bufferB = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    datasize,
    NULL,
    &status);

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(
    context,
    CL_MEM_WRITE_ONLY,
    datasize,
    NULL,
    &status);
```

Source code vector addition

```
//---------------------------------------------------------
// STEP 6: Write host data to device buffers
//---------------------------------------------------------

// Use clEnqueueWriteBuffer() to write input array A to
// the device buffer bufferA
status = clEnqueueWriteBuffer(
    cmdQueue,
    bufferA,
    CL_FALSE,
    0,
    datasize,
    A,
    0,
    NULL,
    NULL);

// Use clEnqueueWriteBuffer() to write input array B to
// the device buffer bufferB
status = clEnqueueWriteBuffer(
    cmdQueue,
    bufferB,
    CL_FALSE,
    0,
    datasize,
    B,
    0,
    NULL,
    NULL);
```

Source code vector addition

```
  //---------------------------------------------------
// STEP 7: Create and compile the program
//---------------------------------------------------

// Create a program using clCreateProgramWithSource()
cl_program program = clCreateProgramWithSource(
    context,
    1,
    (const char**)&programSource,
    NULL,
    &status);

// Build (compile) the program for the devices with
// clBuildProgram()
status = clBuildProgram(
    program,
    numDevices,
    devices,
    NULL,
    NULL,
    NULL);

//---------------------------------------------------
// STEP 8: Create the kernel
//---------------------------------------------------

cl_kernel kernel = NULL;

// Use clCreateKernel() to create a kernel from the
// vector addition function (named "vecadd")
kernel = clCreateKernel(program, "vecadd", &status);
```

Source code vector addition

```
//---------------------------------------------------
// STEP 8: Create the kernel
//---------------------------------------------------

cl_kernel kernel = NULL;

// Use clCreateKernel() to create a kernel from the
// vector addition function (named "vecadd")
kernel = clCreateKernel(program, "vecadd", &status);

//---------------------------------------------------
// STEP 9: Set the kernel arguments
//---------------------------------------------------

// Associate the input and output buffers with the
// kernel
// using clSetKernelArg()
status  = clSetKernelArg(
    kernel,
    0,
    sizeof(cl_mem),
    &bufferA);
status |= clSetKernelArg(
    kernel,
    1,
    sizeof(cl_mem),
    &bufferB);
status |= clSetKernelArg(
    kernel,
    2,
    sizeof(cl_mem),
    &bufferC);
```

Source code vector addition

```
//---------------------------------------------------
// STEP 10: Configure the work-item structure
//---------------------------------------------------

// Define an index space (global work size) of work
// items for
// execution. A workgroup size (local work size) is not
// required,
// but can be used.
size_t globalWorkSize[1];
// There are 'elements' work-items
globalWorkSize[0] = elements;

//---------------------------------------------------
// STEP 11: Enqueue the kernel for execution
//---------------------------------------------------

// Execute the kernel by using
// clEnqueueNDRangeKernel().
// 'globalWorkSize' is the 1D dimension of the
// work-items
status = clEnqueueNDRangeKernel(
    cmdQueue,
    kernel,
    1,
    NULL,
    globalWorkSize,
    NULL,
    0,
    NULL,
    NULL);
```

Source code vector addition

```
//---------------------------------------------------------
 // STEP 12: Read the output buffer back to the host
 //---------------------------------------------------------

 // Use clEnqueueReadBuffer() to read the OpenCL output
 // buffer (bufferC)
 // to the host output array (C)
 clEnqueueReadBuffer(
     cmdQueue,
     bufferC,
     CL_TRUE,
     0,
     datasize,
     C,
     0,
     NULL,
     NULL);

 // Verify the output
 bool result = true;
 for(int i = 0; i < elements; i++) {
     if(C[i] != i+i) {
         result = false;
         break;
     }
 }
 if(result) {
     printf("Output is correct\n");
 } else {
     printf("Output is incorrect\n");
 }
```

Source code vector addition

```
    //---------------------------------------------------------
    // STEP 13: Release OpenCL resources
    //---------------------------------------------------------

    // Free OpenCL resources
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmdQueue);
    clReleaseMemObject(bufferA);
    clReleaseMemObject(bufferB);
    clReleaseMemObject(bufferC);
    clReleaseContext(context);

    // Free host resources
    free(A);
    free(B);
    free(C);
    free(platforms);
    free(devices);
}
```