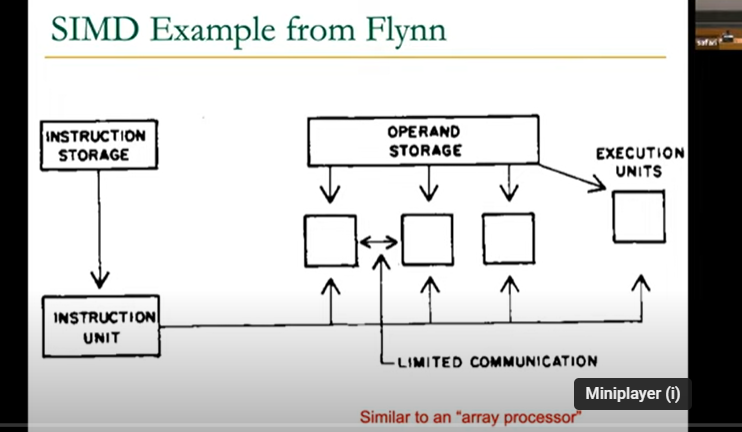
SISD: single instruction operates in single data element   
SIMD:   
2 types: array processor and vector processor   
MISD:  
closet form :systolic array processor , streaming processor

MIMD:  
  
  
some have all these 4  
multi processor and multi threaded processor   
  


DATA PARALELLISM  
concurrency arises from performing same operation on different pieces of data  
different operation in parallel (in data driven manner

Contrast with thread control parallelism

SIMD exploits operation level parallelism in diff data

*SIMD processing paradigm*

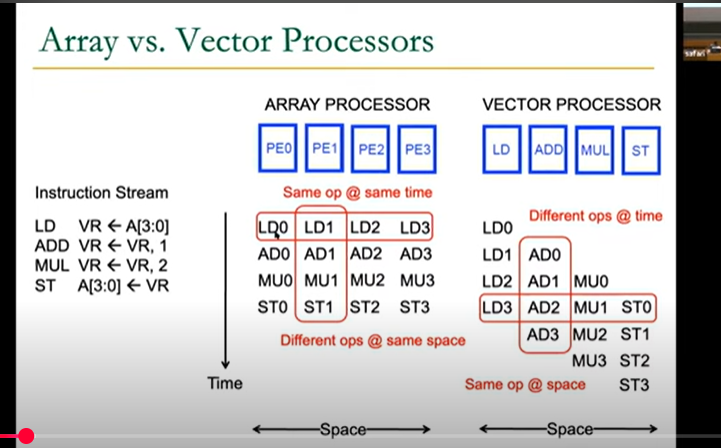
Multiple processing units, execution units

Time space duality

Array processor : intr operates on multiple data elemants at same time using diff spaces

Vector processor: intr operates on multiple data elemants at consecutive time steps using same space  
- need to load /store vector registers  
-need to operate on vector of diff length - vector length register (VLEN)  
-elements of a vector might be stored from each other in memory(VSTR) vector stride register   
stride should be 1 is good but it is not always 1.

Each vector data register hold N M-bit value



Vector Processor (II)

A vector instr performs operation on each element in consecutive cycles   
- vector functional units are pipelined   
- each pipeline stage operates on a different data element   
  
Vector instr allow deeper pipelines   
- no intra vector dependencies , no hardware interlocking needed with a vector   
-no control flow within a vector   
- known stride allows easy address calculation   
- enable easy loading of vector into register   
  
why GPU are energy efficient ? amortizes instr fetch and control overhead over many data   
no need to explicitly code loops highly regular memory access pattern   
  
there are disadvantages of vector processor too-  
- parallelism should be regular   
  
Amdahl’s law   
f: paralleizable fraction of program   
N: number of processor   
speedup = 1/(1-f+f/n)  
max speed up limited by serial portion - serial bottleneck

Notes from the pdf tutorials

- the exploitation of the GPUs vast floating-point throughput as a means of speeding up certain elements of our software.

- unified memory architectures, allowing the GPU and CPU to readily communicate and update data, in order to leverage the computational power of the GPU portion of the chip

- both of these hardware solutions featured octa-core set-ups backed up with multi-compute-unit graphical solutions strongly suggests that multi- and many-core computation will be a significant area of games-related research

**GPU Computation Overview**

concept of number-crunching is very old.

Early solutions revolved around the idea of manipulating pixel data through shader language,

in rendering we perform per-pixel operations in the context of colour space,

physical points are often handled as three element vectors.

if the instruction sets needed to be generalised in terms of vertex, pixel and geometry shader need, why not generalise them as far as possible, this was the context after shader

with advent of CUDA and firestream it was easier to deploy and access to programmable APIs

CUDA is a API

C++ AMP might have claim to be both the most accessible cross-platform API available,

CUDA as it is the most straightforward API through which to implement GPU computation without completely abstracting the GPU hardware (C++ AMP is easier to write in, but does not require us to think about the machine were deploying our code on, OpenCL is less accessible to the novice GPU programmer

**CUDA Hardware**

Kepler CUDA hardware architecture, which maps to the GTX 780Ti graphics processors present in most of the MSc machines

Maxwell-architecture chip

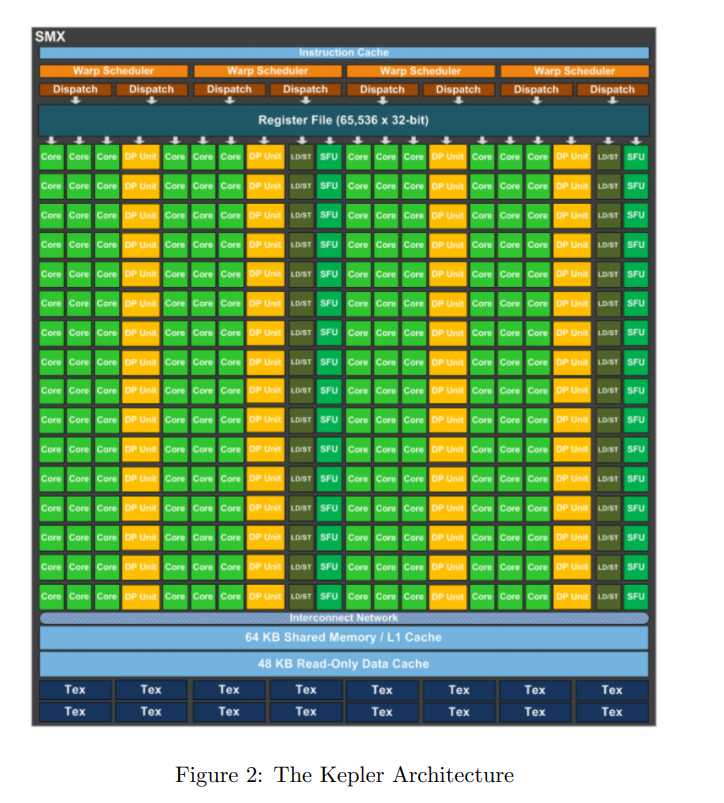
GPU is subdivided into several units (referred to in NVIDIA literature as streaming multiprocessors, or SMX). These units share the L2 Cache and, through that, access to the VRAM (analogous to system memory when programming for the GPU)

An SMX features 192 single-precision cores and 64 double-precision, along with 32 special function units (SFUs units optimised for common mathematical functions).

48KB of Read-Only Data Cache, and 64KB of memory labelled ”Shared Memory/L1 Cache”.

This 64KB is a pool of memory that you can, through the CUDA API, control to favour one or the other (L1 Cache, or Shared Memory) 16KB L1 and 48KB Shared; 16KB Shared and 48KB L1; or, 32KB of each. Shared Memory is a store for variables that can be accessed and updated by any core in the SMX, at any time. The L1 Cache pool is a shared cache pool which is used by every core in an SMX.

The instruction cache for a single SMX is used by all cores in that SMX (meaning that all cores will execute the same set of instructions). The Warp Scheduler handles the initiation of cores to execute their ’instance’ of the instruction (the kernel instance, discussed later). If instruction sets branch significantly (if-then conditions which make their completion time varied), the warp scheduler will not be able to leverage maximum efficiency from the cores in the SMX.



The CPUs in your desktop have as much L1 cache per core as is allocated by default to all 192 single precision cores in the SMX combined. They also enjoy more versatile instruction cache, optimised for resolving cache misses more rapidly (not something the GPU can claim, regrettably).

the GPU is intended to do: it executes shaders, which are themselves very simple functions (in terms of instructions if not theory), across all cores simultaneously. Its memory architecture is optimised towards that purpose. And if we are going to leverage this hardware to perform computationally intensive tasks for us, we need to keep that firmly in mind.

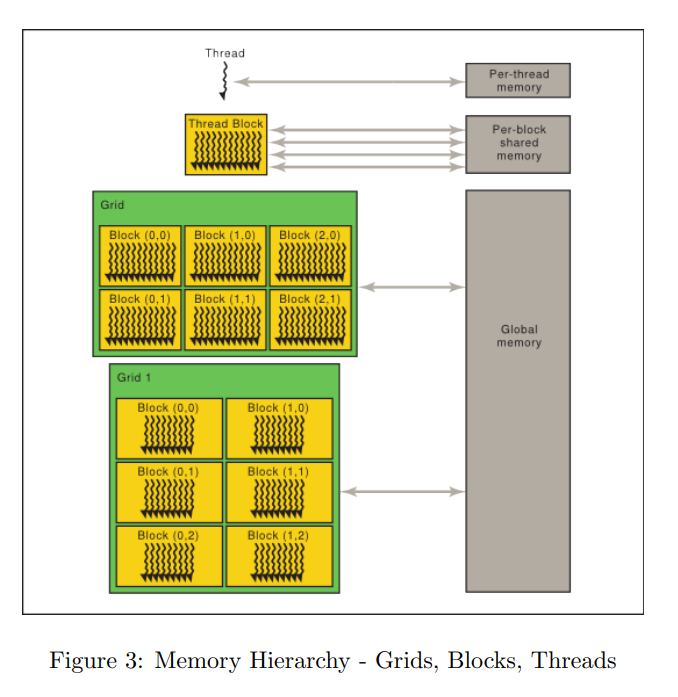
**CUDA Software**

CUDA is a C-styled language that permits the deployment of programs on the GPU. CUDAs syntax is relatively straightforward (and documented in the CUDA API).

The CUDA programming model is built on the idea of a grid execution; within the grid are a number of blocks; within a block, are a number of threads. A thread is a single instance of a kernel. It accepts a set of variables, and performs a set of instructions using those variables. A thread has a block ID within its thread block and grid; this is used to determine the threads unique ID, which normally maps to the data element it is accessing. IE, threadID 103 accesses the 103rd element of the arrays that have been sent to the GPU.

A block is a set of concurrently executing threads. These threads cooperate with each other through barrier synchronisation and shared memory. A block as a block ID within its grid. A grid is an array of thread blocks that execute the same kernel. The grid reads in inputs from global memory, writes results back out to global memory, and synchronises between multiple, dependent kernel calls.

You can consider initiating a kernel function as generating a grid, whose size is determined by the number of elements you have instructed the GPU to process. A constant in CUDA is stored in constant memory accessible by all threads. Arrays cannot be stored in constant memory. Shared memory is accessible to all threads in a block; arrays can be stored there. Similarly, read-only memory is accessible to all threads in a block.  
  
a set of threads executing the same instruction are dynamically grouped into a warp (wavefront) by the hardware.



Read the program flow and paradigms from the detailed PDF.

Tutorial 2 contain details about implementation read when doing.  
  
