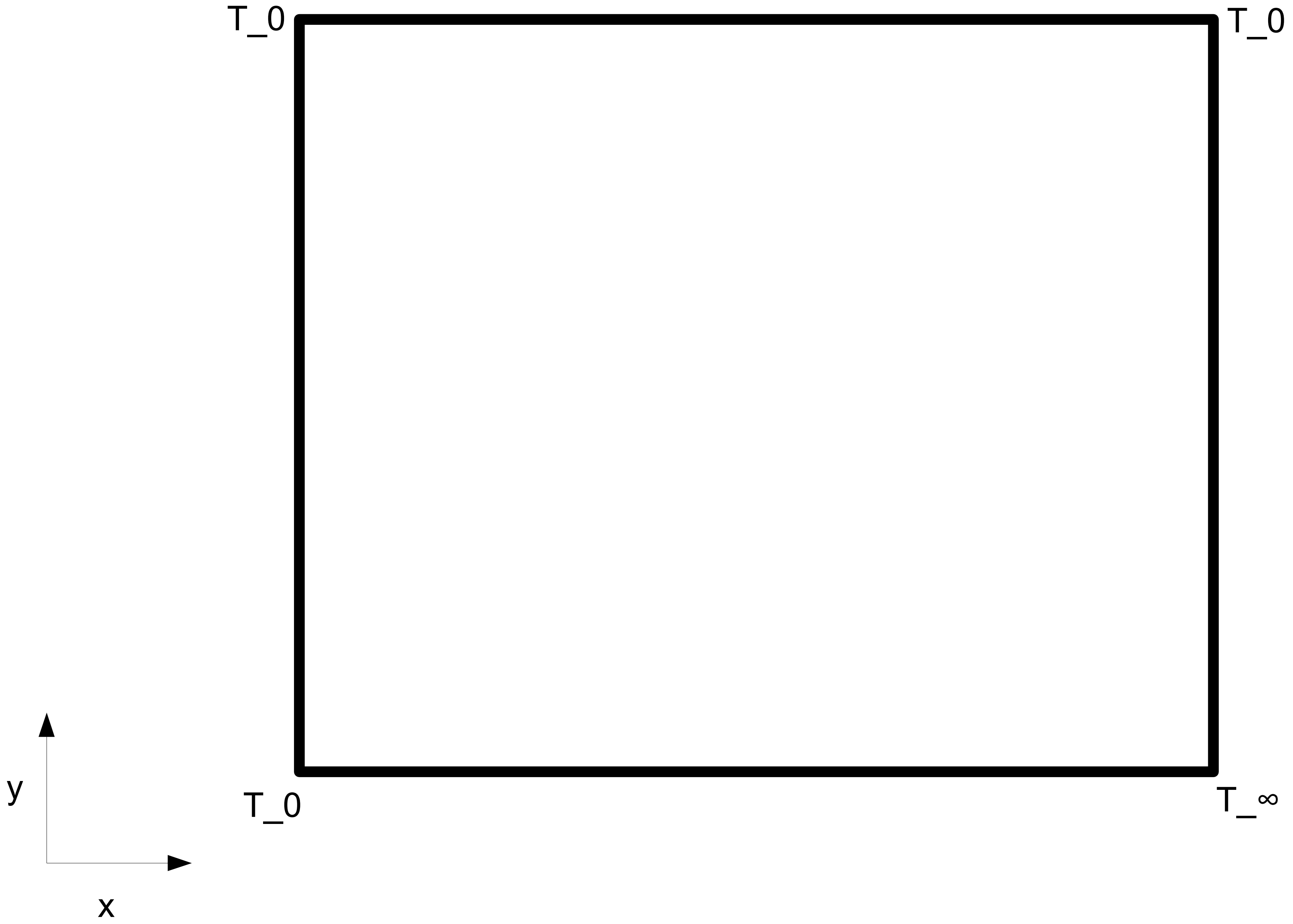
High Performance Scientific computing
Lecture 8

S. Gopalakrishnan

Steady State Heat Conduction

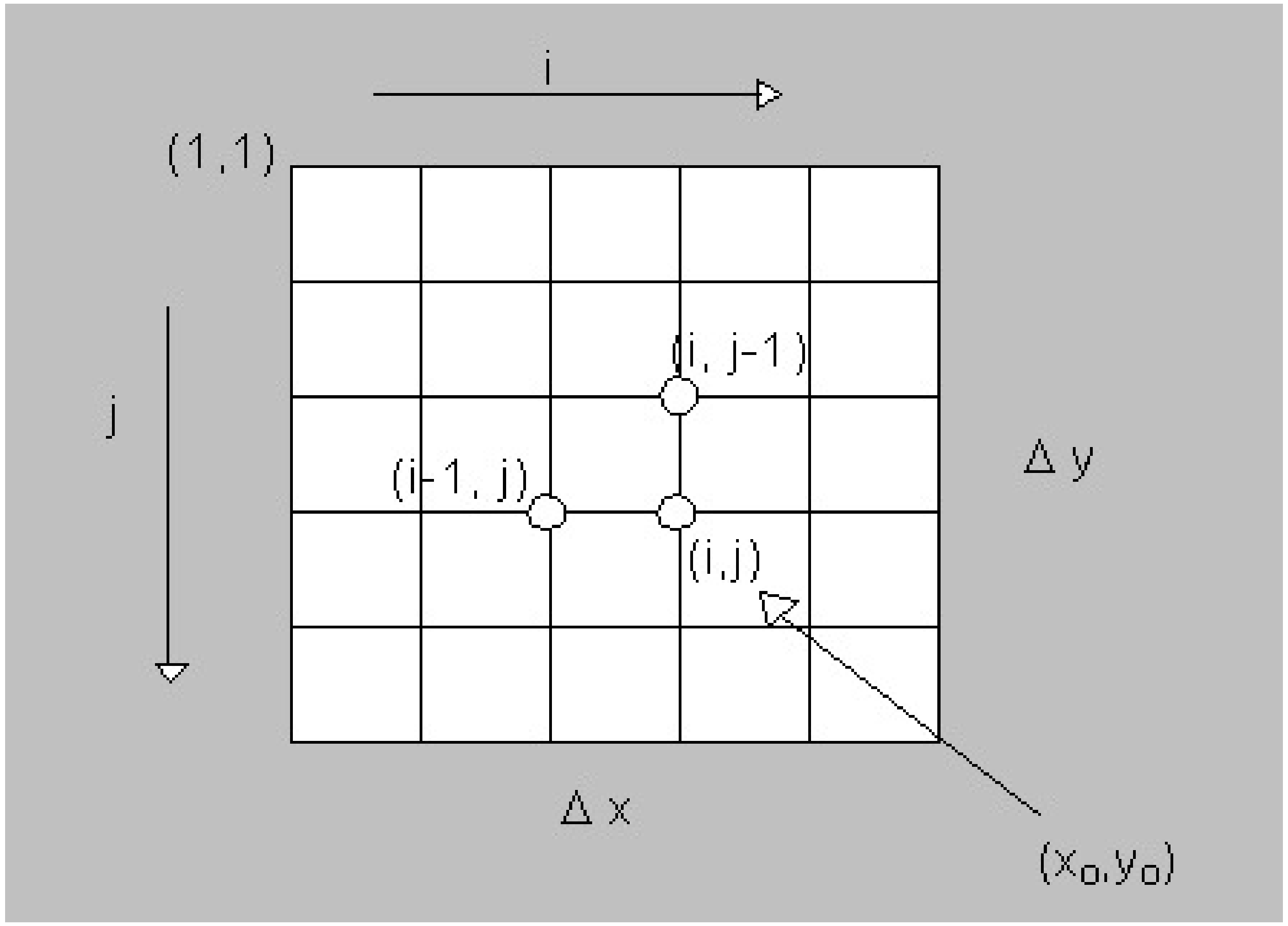


Steady State Heat Conduction

Phenomenon is modelled using Laplace's equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \nabla^2 T = 0$$

which can be discretised on a grid as,



Non-Blocking Sends and Receives

All of the receives that we will use today are blocking. This means that they will wait until a message matching their requirements for source and tag has been received.

Contrast this with the Sends, which try not to block by default, but don’t guarantee it. To do this, they have to make sure that the message is copied away before they return in case you decide to immediately change the value of the sent variable in the next line. It would be messy if you accidentally modified a message that you thought you had sent.

It is possible to use non-blocking communications. This means a receive will return immediately and it is up to the code to determine when the data actually arrives using additional routines (MPI_WAIT and MPI_TEST). We can also use sends which guarantee not to block, but require us to test for a successful send before modifying the sent message variable.

There are two common reasons to add in the additional complexity of these non-blocking sends and receives:

- **Overlap computation and communication**
- **Avoid deadlock**

The first reason is one of efficiency and may require clever re-working of the algorithm. It is a technique that can deal with high-latency networks that would otherwise have a lot of deadtime (think Grid Computing).

Non-Blocking Sends and Receives

This second reason will often be important for large codes where several common system limits can cause deadlocks:

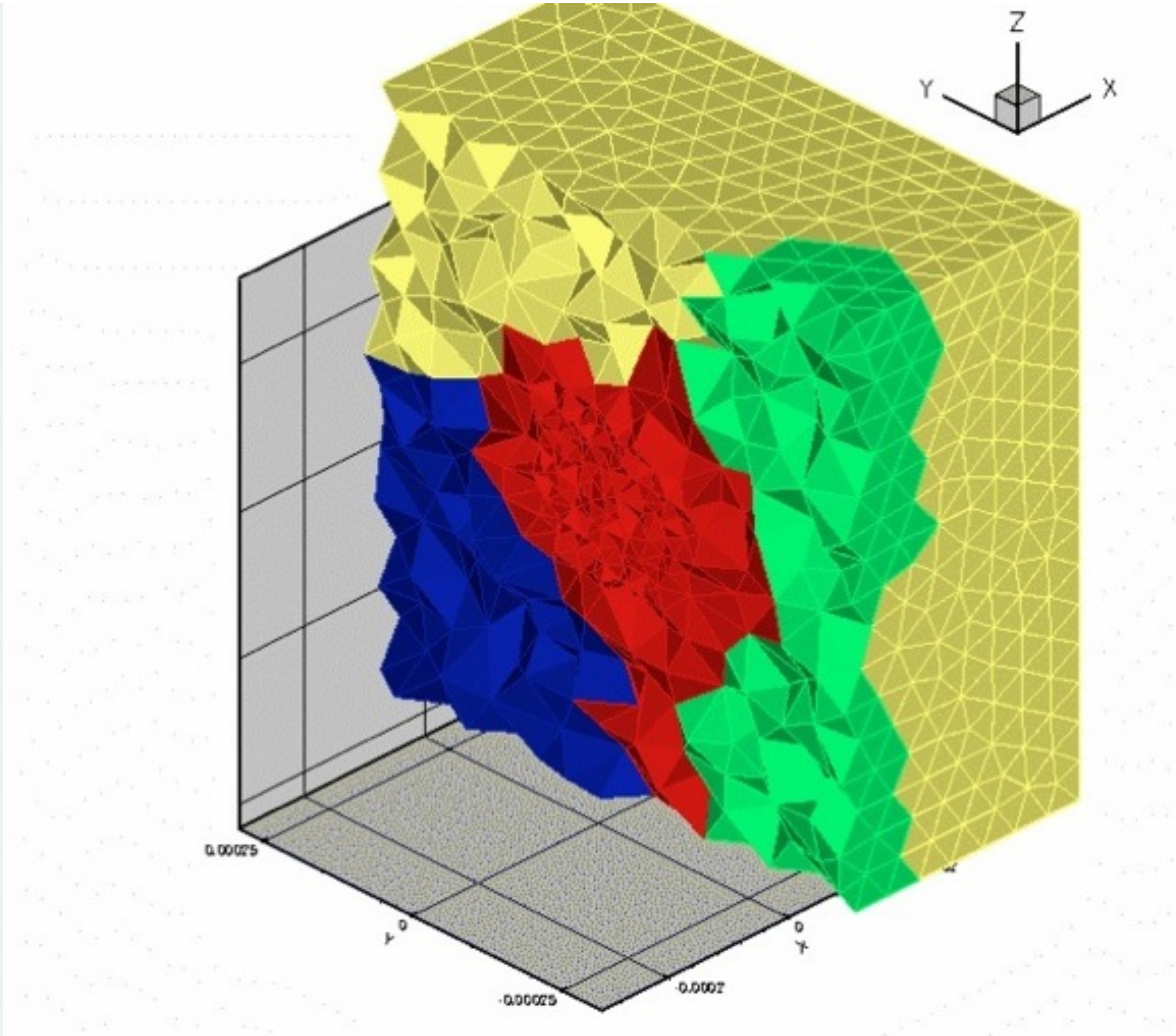
1)**Very large send data**
Large sections of arrays are common, as compared to the single integer that we used in this last example) can cause the MPI_SEND to halt as it tries to send the message in sections. This is OK if there is a read on the other side eating these chunks. But, what happens if all of the PEs are trying to do their sends first, and then they do their reads?

2)**Large numbers of messages**
This tends to scale with large PE counts) overload the network's in-flight message limits. Again, if all nodes try to send a lot of messages before any of them try to receive, this can happen. The result can be a deadlock or a runtime crash.

Note that both of these cases depend upon system limits that are not universally defined. And you may not even be able to easily determine them for any particular system and configuration. Often there are environment variable that allow you to tweak these. But, whatever they are, there is a tendency to aggravate them as codes scale up to thousands or tens of thousands of PEs.

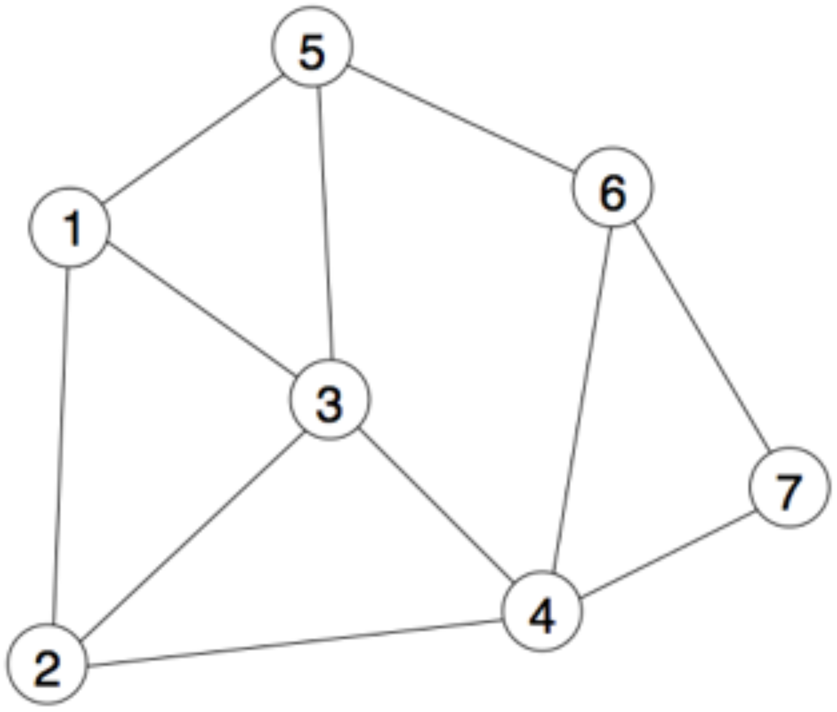
Domain Decomposition

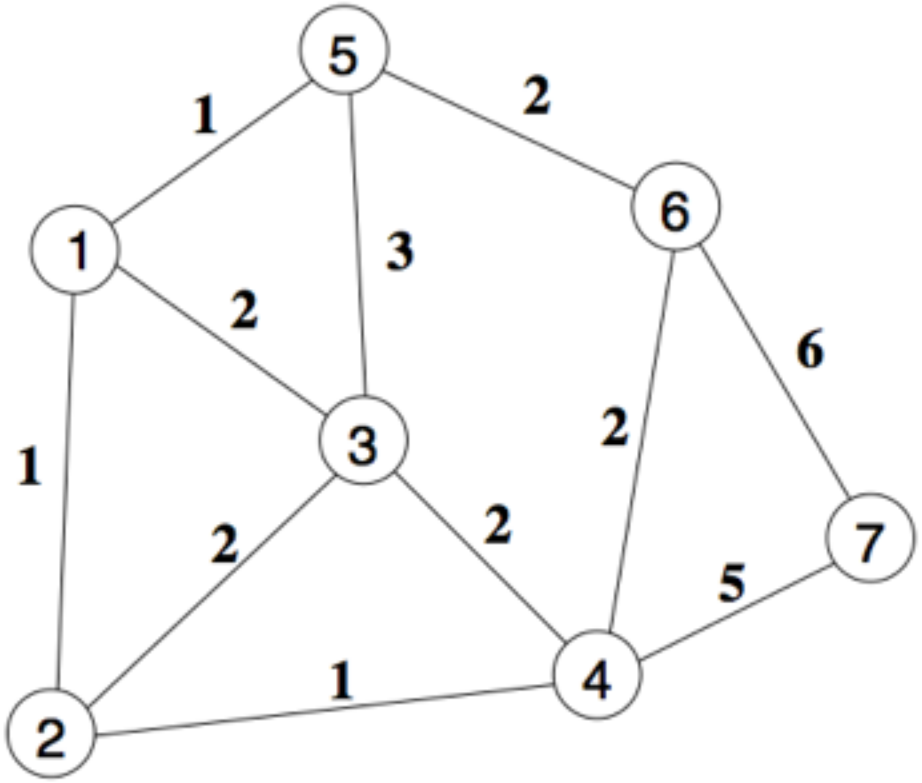
Domain decomposed by METIS



Metis is a Library for partitioning unstructured graphs / meshes

Mesh representation as Connected graphs





Unweighted graph

Weighted graph

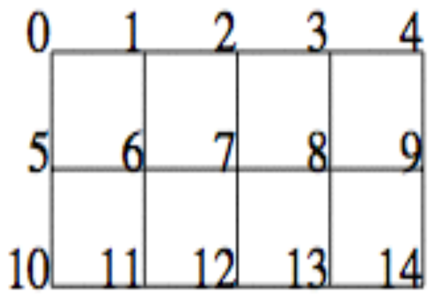
*Images: From METIS manual

CSR format and adjacency graph

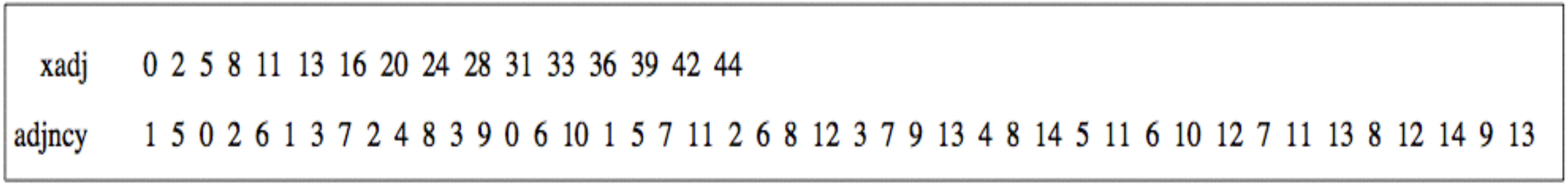
“The adjacency structure of the graph is stored using the compressed storage format (CSR). The CSR format is a widely used scheme for storing sparse graphs. In this format the adjacency structure of a graph with **n** vertices and **m** edges is represented using two arrays **xadj** and **adjncy**. The **xadj** array is of size **n + 1** whereas the **adjncy** array is of size **2m** (this is because for each edge between vertices **v** and **u** we actually store both **(v, u) and (u, v))**.”*Q

Quote: From METIS manual

CSR format and adjacency graph



(a) A sample graph



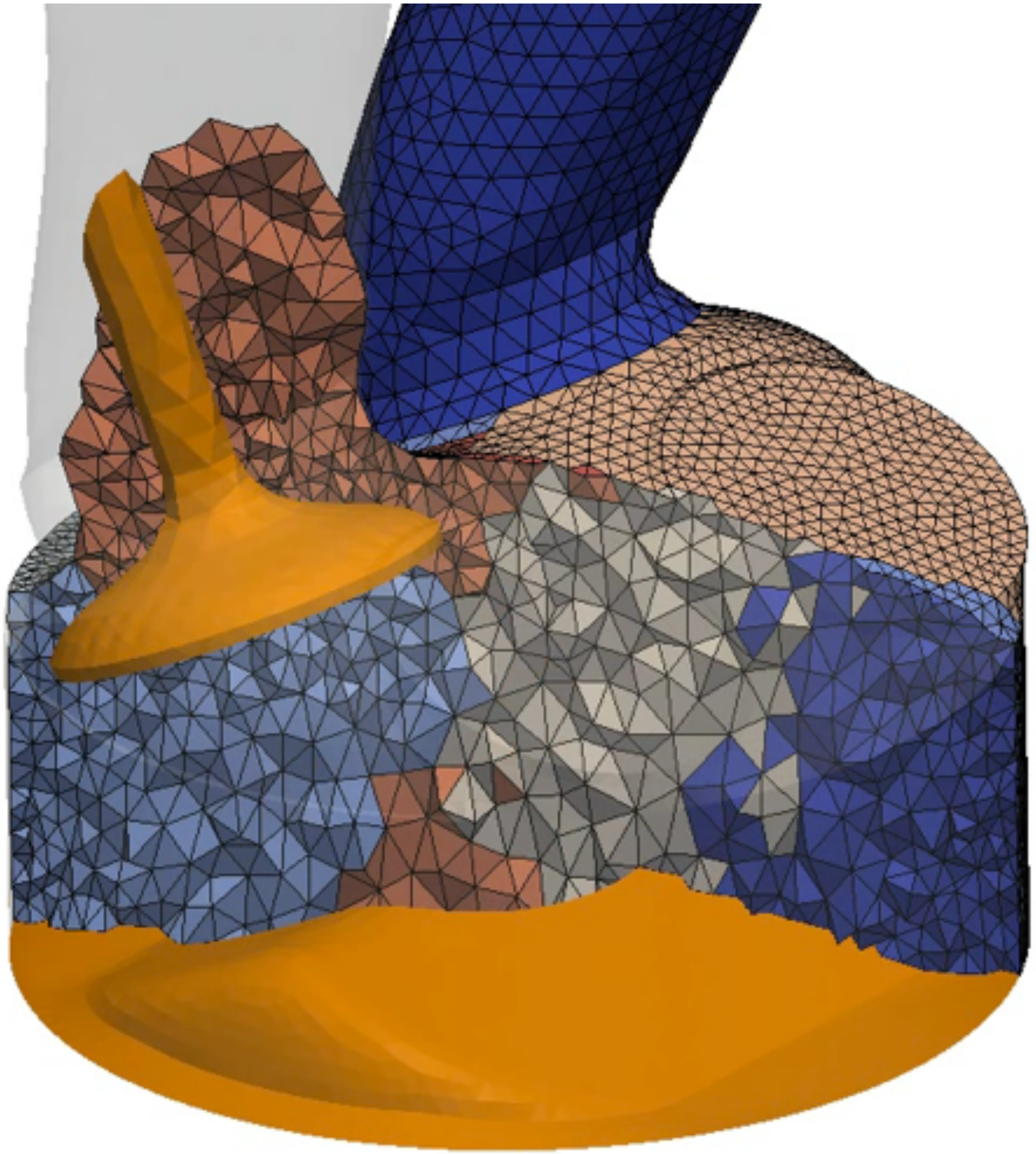
xadj 0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44

adjncy 1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

(b CSR format

*Images: From METIS manual

Domain decomposed by METIS



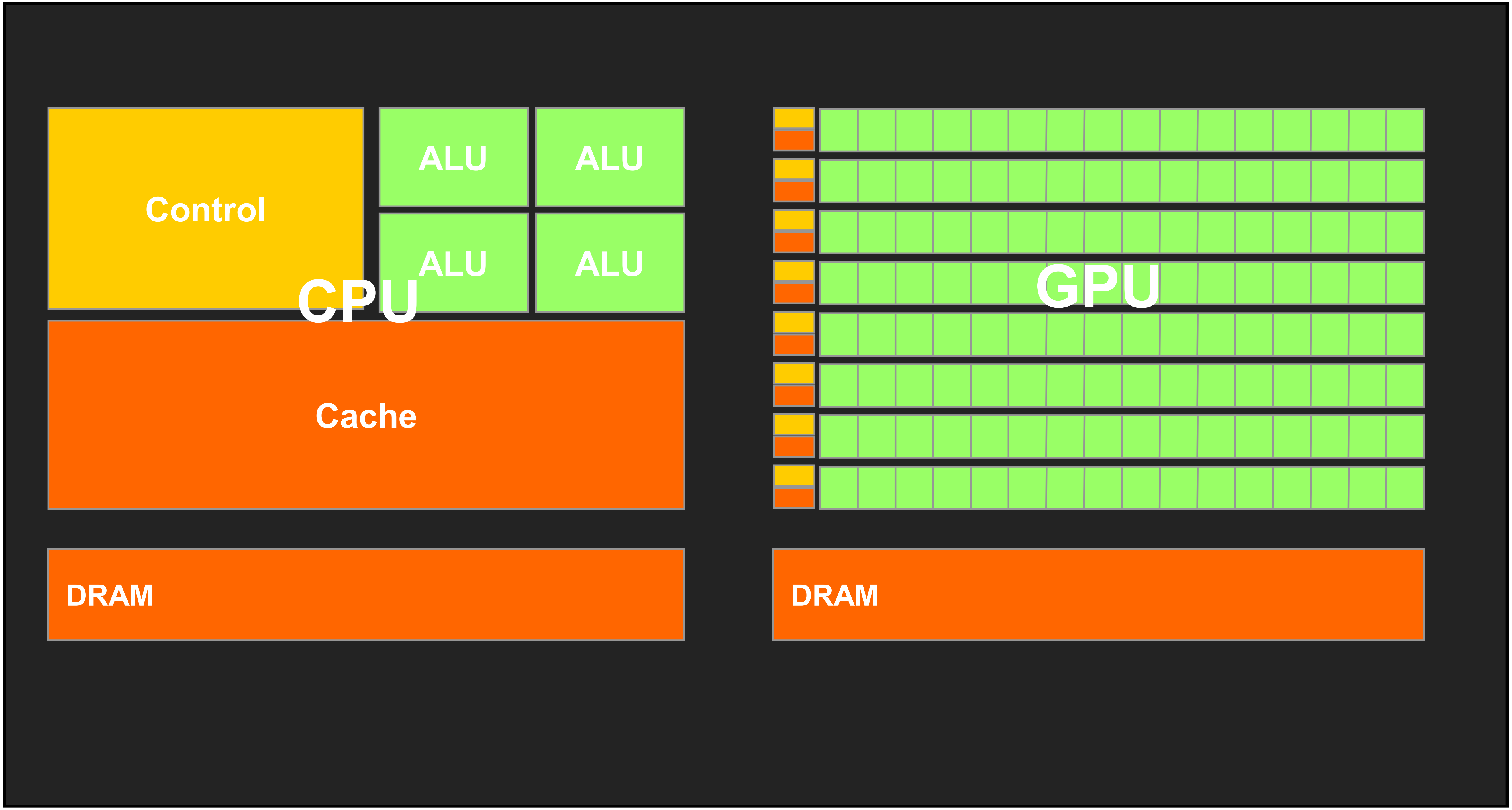
Computational domain of a automotive engine cylinder. S. Menon [2011]

GPGPU Computing
(General Purpose Graphics Processing unit)

GPU Evolution

- The **Graphic Processing Unit** (GPU) is a processor that was **specialized** for processing graphics.
- The GPU has recently **evolved** towards a **more flexible** architecture.
- Opportunity: We can implement ***any algorithm*,** not only graphics.
- Works on SIMD (Single Instruction Multiple Data) approach.
- Challenge: obtain **efficiency** and **high**

Comparison of CPU vs GPU Architecture



Source: Prof. Wen-mei W. Hwu UIUC

GPU CPU Analogy

GPU

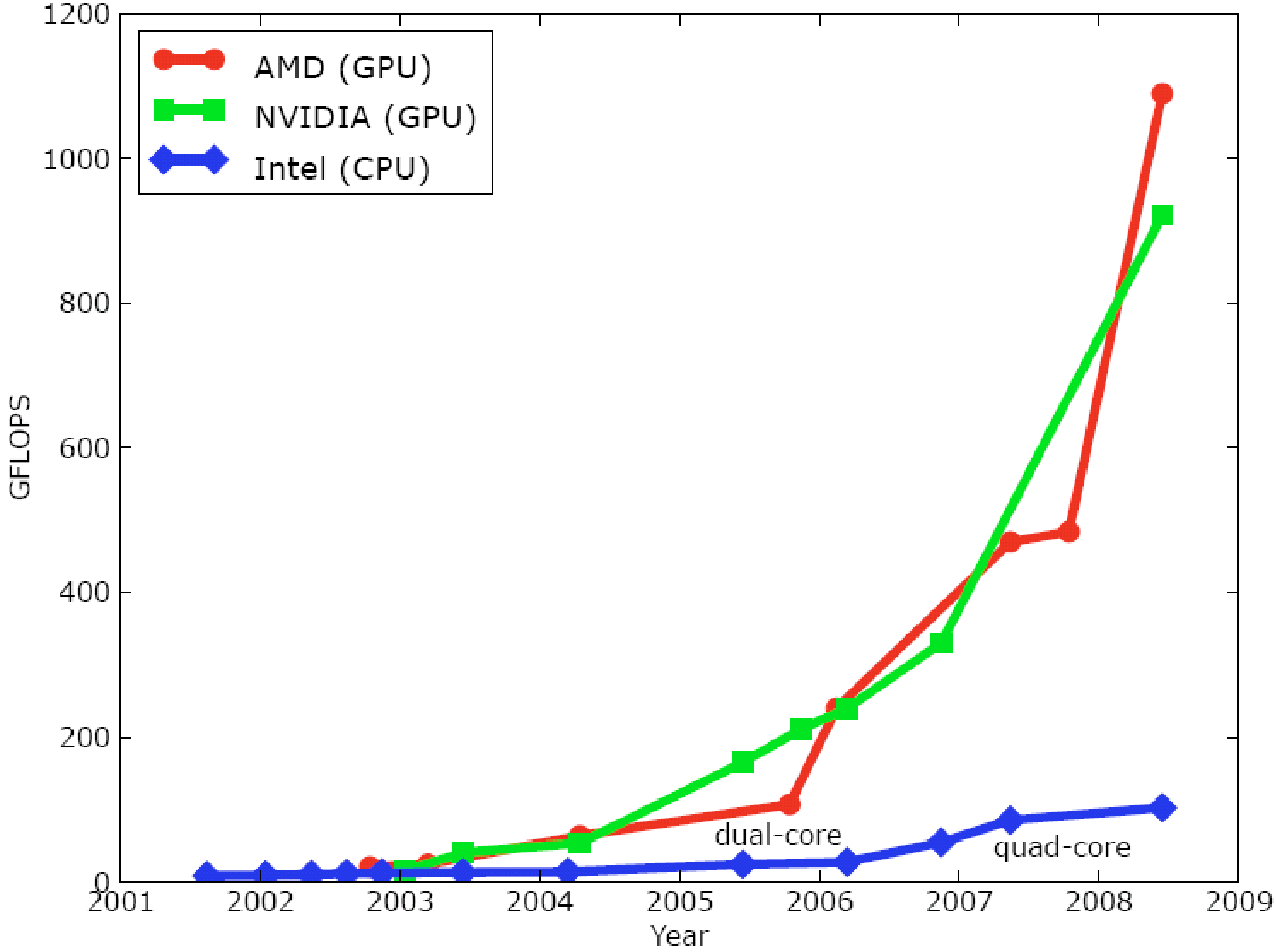
CPU

It is more effective to deliver Pizza's through light duty scooters rather than big truck. Similarly effective to use several lightweight GPU processors for parallel tasks.

GPU Performance

Peak performance increase

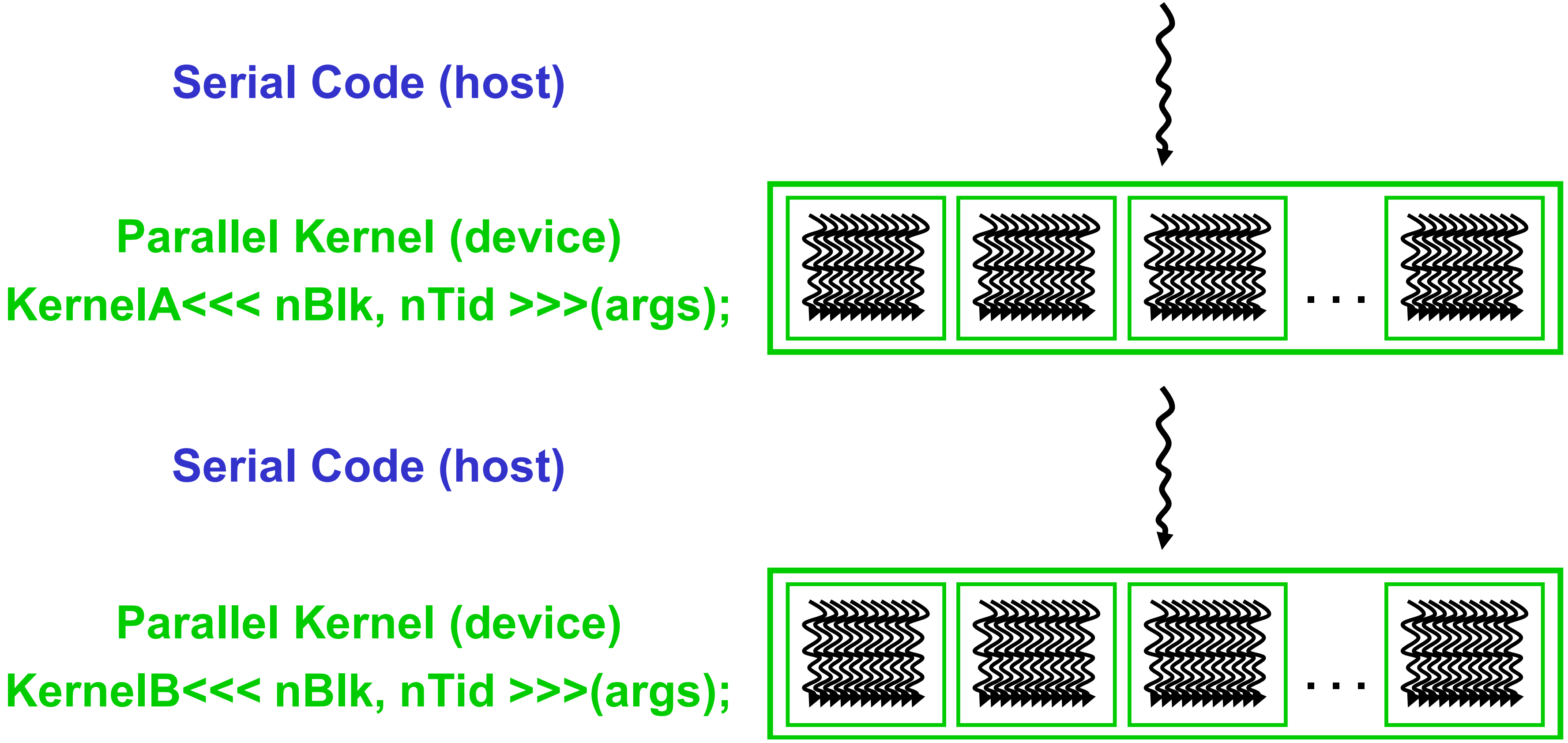
- Calculation ~ 1 TFlop on Desktop
- Memory Bandwidth ~ 150 GB/s



Courtesy: John Owens

CUDA/ OpenCL Execution Model

- Combination of Host (CPU) and Device(GPU) code
- Parallel portions of the code are expressed as device kernels and they run on many threads



Serial Code (host)

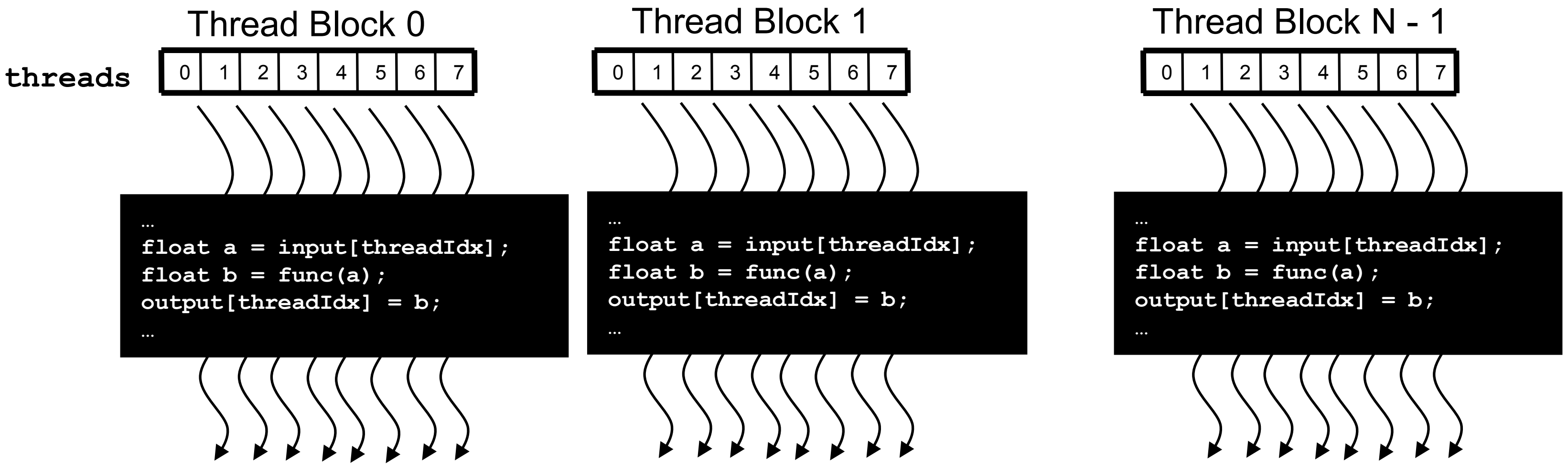
Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

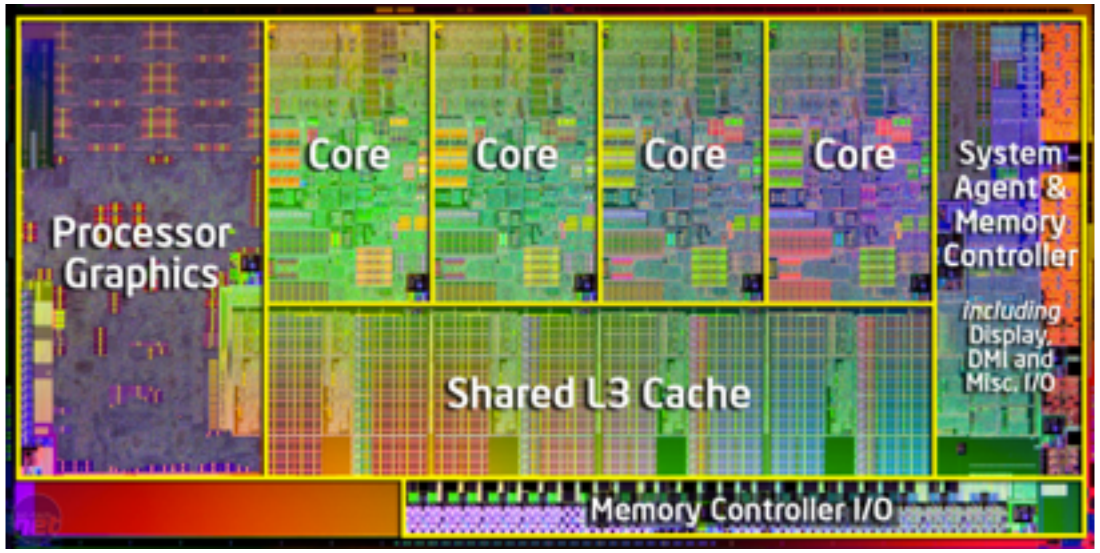
GPU threads vs CPU threads

- GPU threads are very lightweight and hence have less overheads
- For efficiency hundreds of GPU threads are required. Few in case of CPU
- We can subdivide threads into multiple blocks for shared memory cooperation.



```
…
float a = input[threadIdx];
float b = func(a);
output[threadIdx] = b;
…
```

OpenCL devices



Compute Unified Device Architecture (CUDA)

- CUDA set of APIs (application program interface) to use GPU's for general purpose computing
- Developed and released by NVIDIA Inc. Works only on NVIDIA GPU hardware
- Works on commercial GPU's and as well as specialized ones for scientific computing (Tesla)
- CUDA compiler supports C programming language. Extensions to FORTRAN are possible.
- Opensource alternative is OpenCL.

CUDA and OpenCL Advantages

- Abstracting from the hardware
- Abstraction by the CUDA / OpenCL API. You don't see every little aspect of the machine.
- Gives flexibility of vendors. Change hardware but keep legacy code. (CUDA —> only NVIDIA)
- Forward compatible.
- Automatic Thread management (can handle +100k threads)
- Multithreading: hides latency and helps maximize the GPU utilization.
- Transparent for the programmer (you don't worry about this.)
- Limited synchronization between threads is provided.
- Difficult to dead-lock. (No message passing!)

Source : Felipe Cruz, PASI workshop 2011

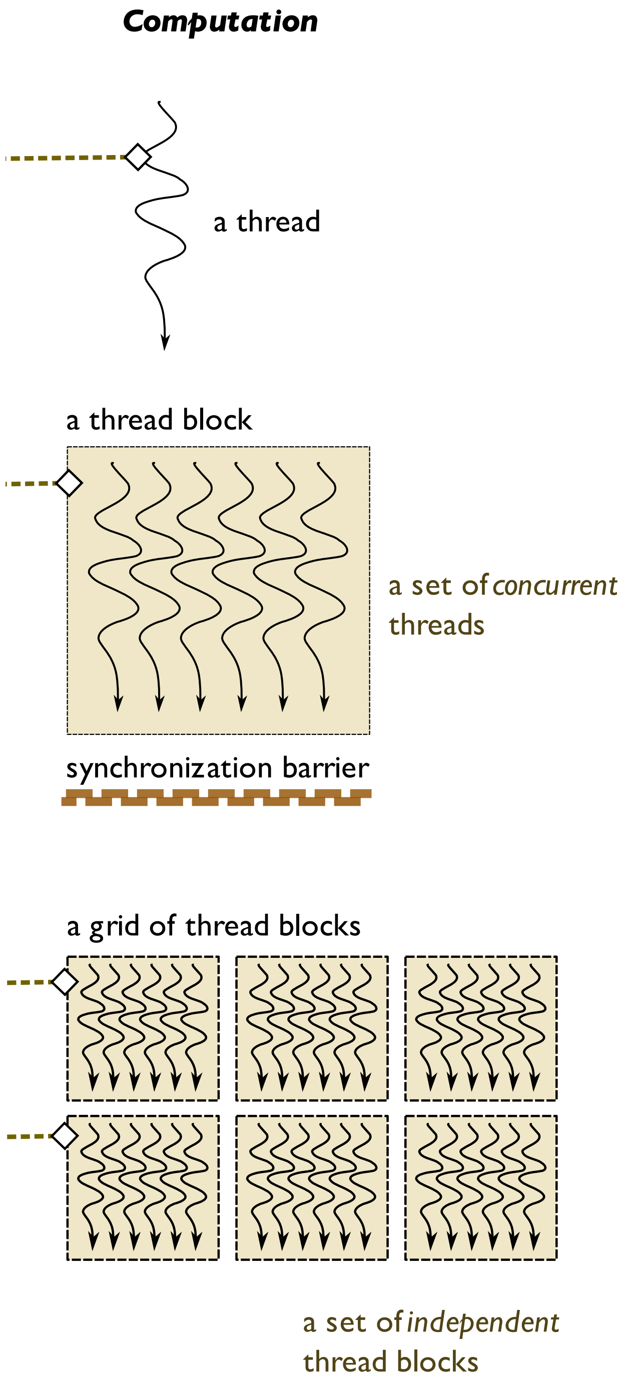
Programming effort

- Analyze algorithm for exposing parallelism:
  - Block size
  - Number of threads
  - Tool: pen and paper
- Challenge: Keep machine busy (with limited resources)
  - Global data set (Have efficient data transfers)
  - Local data set (Limited on-chip memory)
  - Register space (Limited on-chip memory)
  - Tool: Occupancy calculator

Source : Felipe Cruz, PASI workshop 2011

Thread Hierarchy

- Kernels are executed by thread.
- A kernel is a simple C program.
- Each thread has it own ID.
- Thousands of threads execute same kernel.
- Threads are grouped into blocks.
- Threads in a block can synchronize execution.
- Blocks are grouped in a grid.
- Blocks are independent (Must be able to be
- executed in any order.)



Source : Felipe Cruz, PASI workshop 2011

Memory Hierarchy

Three **types** of memory in the graphic card:

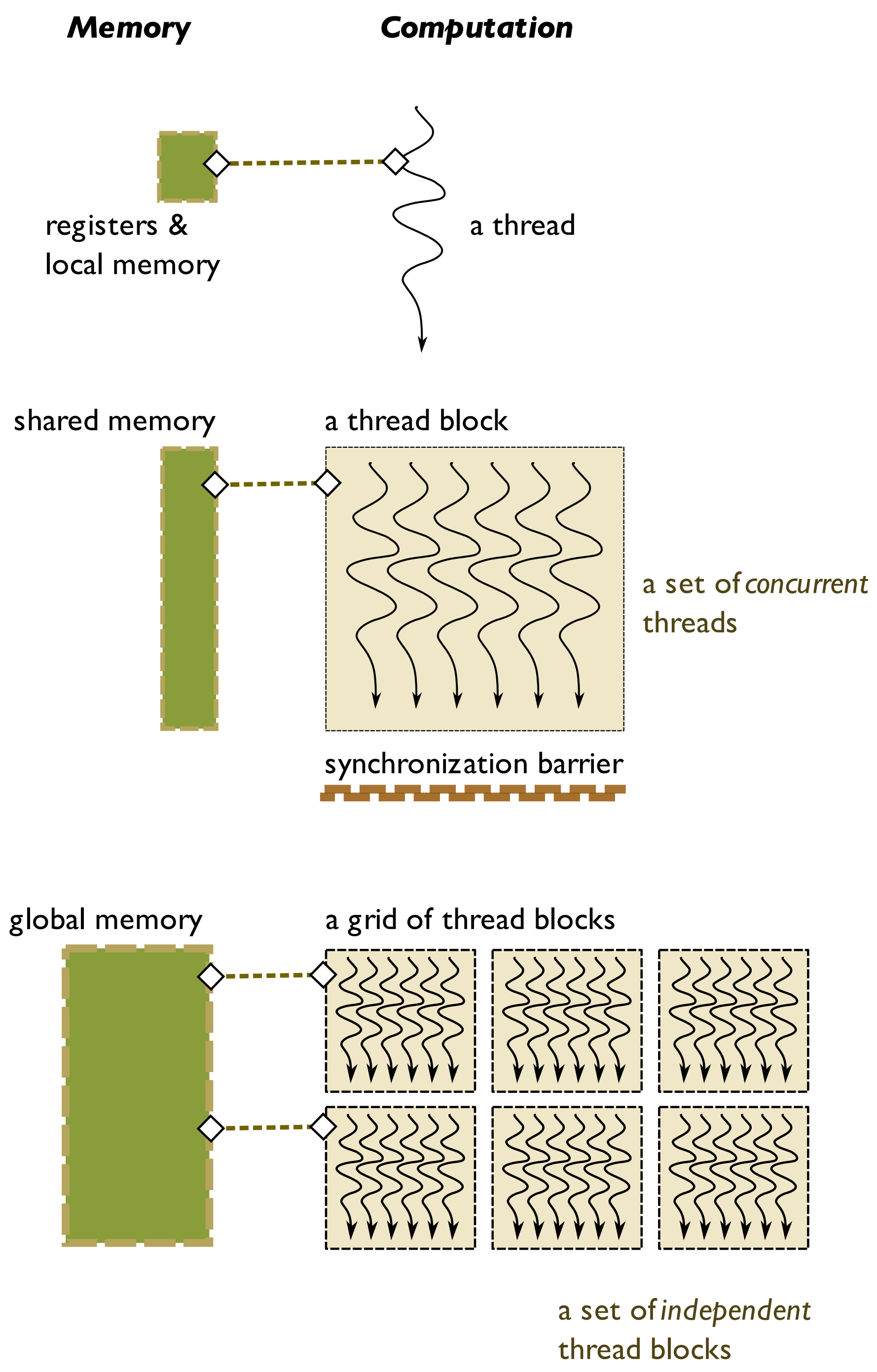
Global memory: 4GB
Shared memory: 16 KB
Registers: 16 KB

Latency:
Global memory: 400-600 cycles

Shared memory: Fast
Register: Fast

Purpose:
Global memory: IO for grid
Shared memory: thread collaboration

Registers: thread space



Source : Felipe Cruz, PASI workshop 2011

Basic C extensions

Function modifiers

Basic C extensions

- __global__ : to be called by the host but executed by the GPU.
- __host__ : to be called and executed by the host.

Kernel launch parameters

- Block size: (x, y, z). x*y*z = Maximum of 768 threads total. (Hw dependent)
- Grid size: (x, y). Maximum of thousands of threads. (Hw dependent)

Variable modifiers

- __shared__ : variable in shared memory.
- __syncthreads() : sync of threads within a block.

Source : Felipe Cruz, PASI workshop 2011

Target Metrics

Throughput (measured in FLOP/s):

Average number of floating point operations per second than can be executed on the GPU.

Bandwidth (measured in GigaBytes/s):

Rate at which data is transferred between memory and the processor per second. All read and write memory transactions must be considered.

Source : Felipe Cruz, PASI workshop 2011

Design notions

Computational intensity:

Ratio of floating point operations to memory accesses.

Concurrency:

Sections of the algorithm that can be executed concurrently.

Can be organized into levels: fine to coarse grained concurrency.

Homogeneity of calculations:

Degree at which concurrent computations are the same, input independent.

Data-locality:

The way in which physically stored data is accessed by the algorithm.

Spatial data locality: data is physically adjacent.

Temporal data locality: data is temporally adjacent.

Source : Felipe Cruz, PASI workshop 2011