Advanced Multicore Programming

Dr Andrew Moss

[1]BTH Karlskrona

August 28, 2011

Overview

Sequential Machines

- Performance limitations; clock speed, bus width . . .
- Relatively hard limits: 3Ghz · 4 ops 12Gflop/s
- Increase performance by doing more per clock cycle

Parallel Programming

- MIMD (multi-core) machines
  - Multiple Instruction Multiple Data
  - Similar programming model (plus locks and barriers)
- SIMD (many-core) machines
  - Vastly different programming model
  - Single Instruction Multiple Data
  - Different tradeoffs in scalability
  - 100s of cores, multiple Tflop/s performance
- Modern computers use a hybrid approach
  - Multi-core processor with a many-core graphics card

Focus

This course will focus on programming many-core machines

- Achieves the performance necessary for games
- Expect a transition from rasterisation to ray-tracing (next 5 years?)
- Beyond that path-tracing

Once a reasonable level of graphical fidelity has been reached

- Diminishing returns
- How geometry animates has more of an impact than the rendering
- Humans are very good at spotting unrealistic motion

Adding realism to a virtual world

- Simulate the behaviour of complex systems
- Gravity, Rigid bodies, fluids, gases, combustion . . .

Administration

Mixed Lecture / Lab sessions

- Four two-hour slots per weeks (days differ slightly)
- Mostly lab work, some lectures amongst that time
- The lectures are just an introduction

Advanced Level Course

- One criteria is an introduction to research-level material
- No textbook
- The assignments are selected from GPU Gems 3
  - Slightly out of date, worth reading anyway
  - Programming model is a bit different — translate what you read

Bricolege

Hands-on experience

- Also covers the annoying details
- Painful but necessary experience
  - Learning to interpret compiler error messages
  - Figuring out why the wrong libraries are linking
  - etc . . .

Try to avoid it on this course

- You've already gained those skills by this point
- Can't avoid all of it — GPGPU is still mainly used in research labs
- Beware the cutting-edge — unstable tool-chains, dodgy drivers . . .

To minimise the pain

- Lab machines with a working installation
- You will be using GCC in a unix environment
  - Currently the most stable system for this work

UNIX in 30 seconds

Two main applications are on the taskbar

- Terminal
- Firefox

Basic commands

- cd
- ls
- evince
- g++ file.cc -o progname
- make
- some kind of editor . . .
  - if you've never used a UNIX editor try nano

Approach

You will be writing code for GPUs

- This will not be fragment-shaders as you have seen before
- These will be OpenCL kernels
- New language, but looks set to become **the** standard for GPGPU

During your careers GPGPU will cross-over into the mainstream

- Apple are pushing OpenCL heavily via Grand Central
- AMD have standardised on it completely
- Nvidia seem to like it although they don't want to drop CUDA yet

Learning Outcomes

Official wording

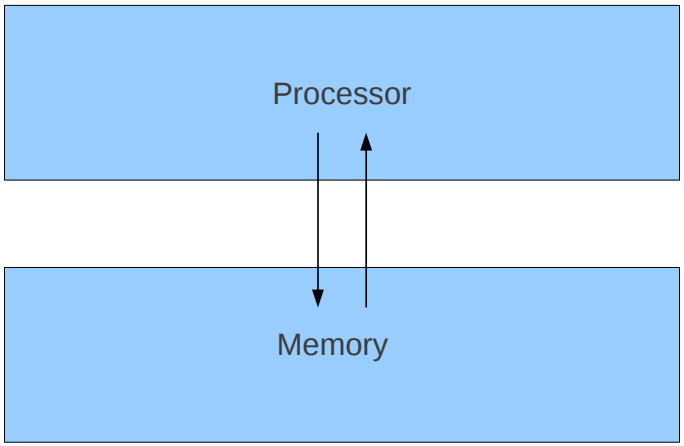
- Translate parallel programs into OpenCL.
- Analyse the data-flow in a problem to partition it into parallel pieces.
- Explain the impact of architecture design on performance.
- Design and implement programs to achieve high performance.

Translation

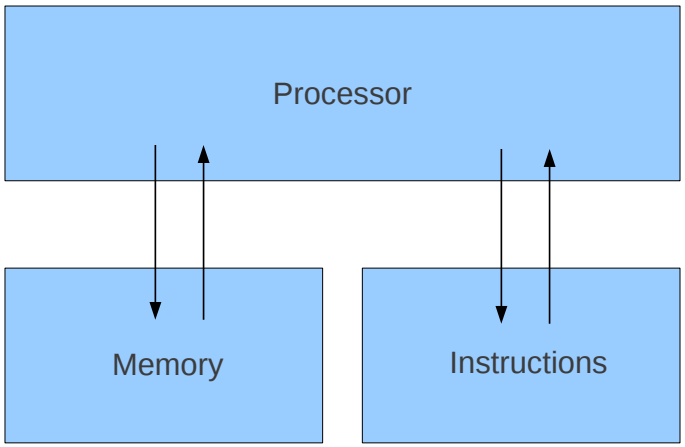
- Understand enough about how a GPU operates to write high-performance code.

Recap on Computer Architecture

Von Nuemann (1950s)



Harvard Architecture



Very simple models

- Both expose the main bottleneck in computing

Some architecture background

FACT: Memory will always be too slow for your application

- Terje Mathisen (optimization guru): "All programming is an exercise in caching."

Memory has two problems to overcome

- Bandwidth — build more wires.
- Latency — increase the speed of light?

Ok, so latency is hard

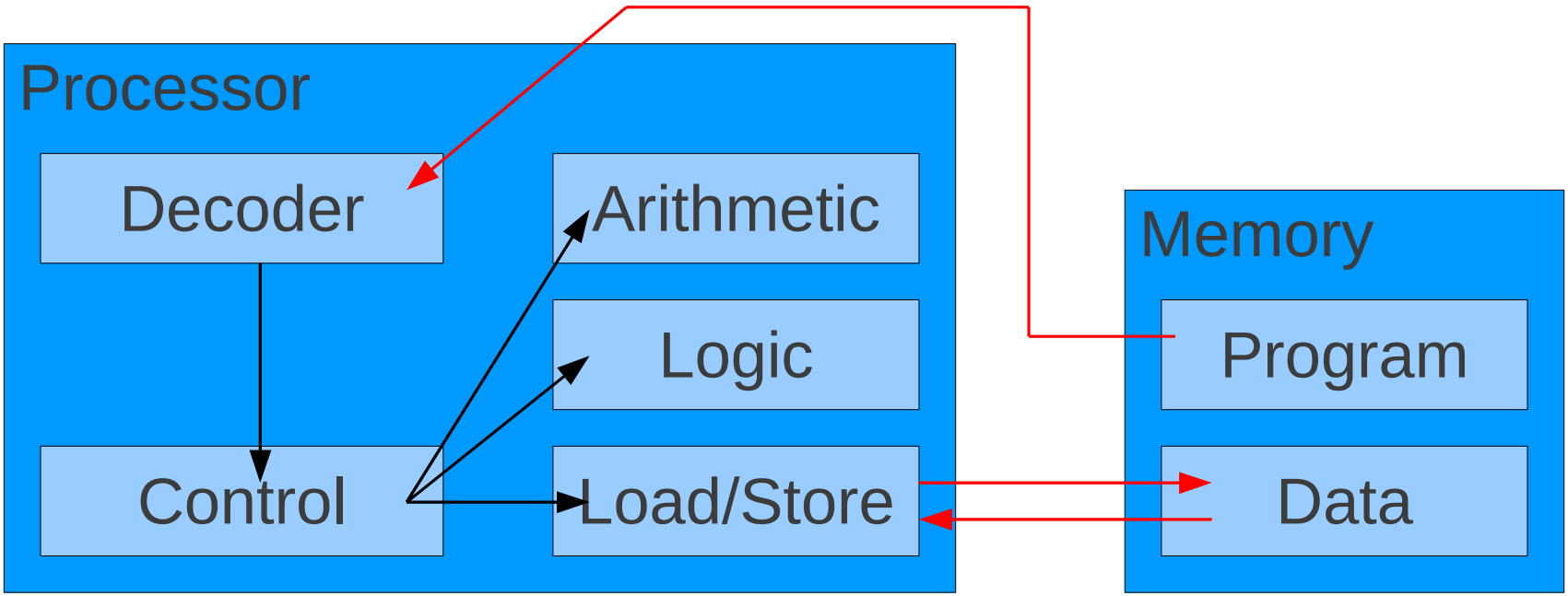
- If you can't decrease it the next best thing is to hide it
- How can a processor hide the latency of its memory operations?

Processor Bottlenecks

Black wires are fast (very small, low capacitance)

- Red wires are slow (very large, high capacitance)

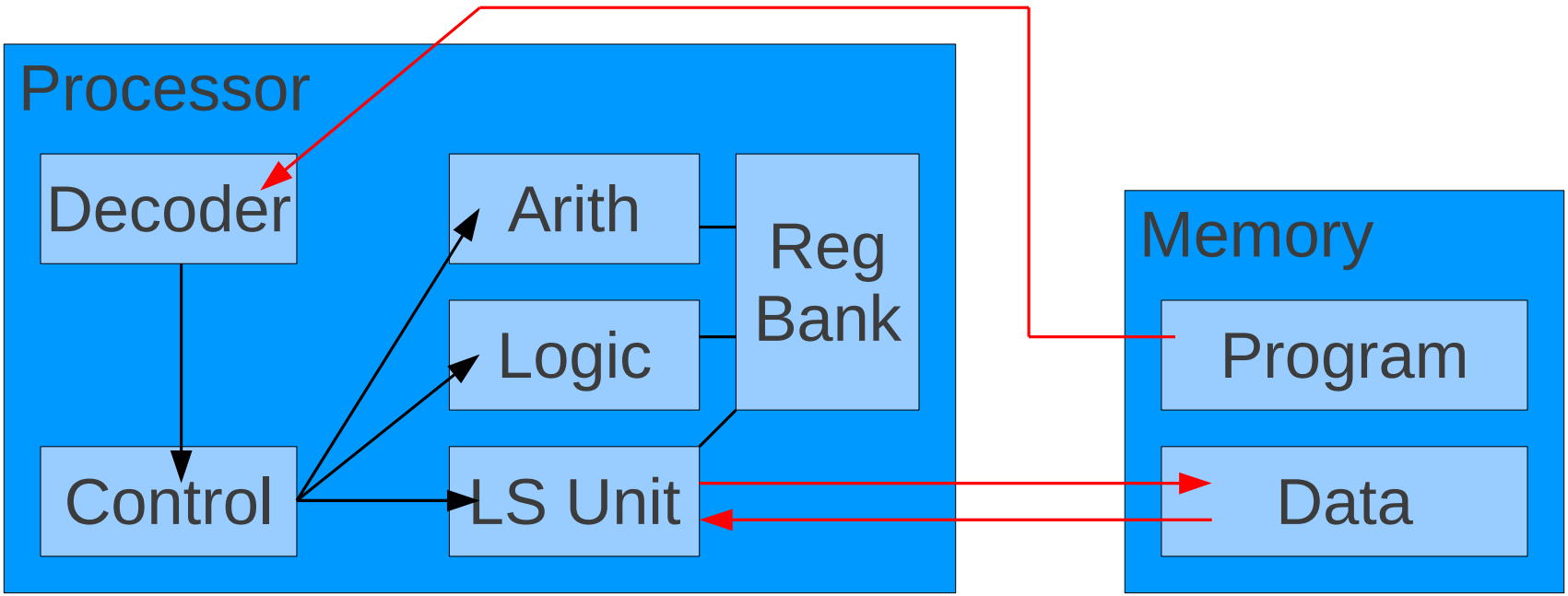
Move some memory across the slow red wires . . .



Processor Bottlenecks II

Registers are expensive

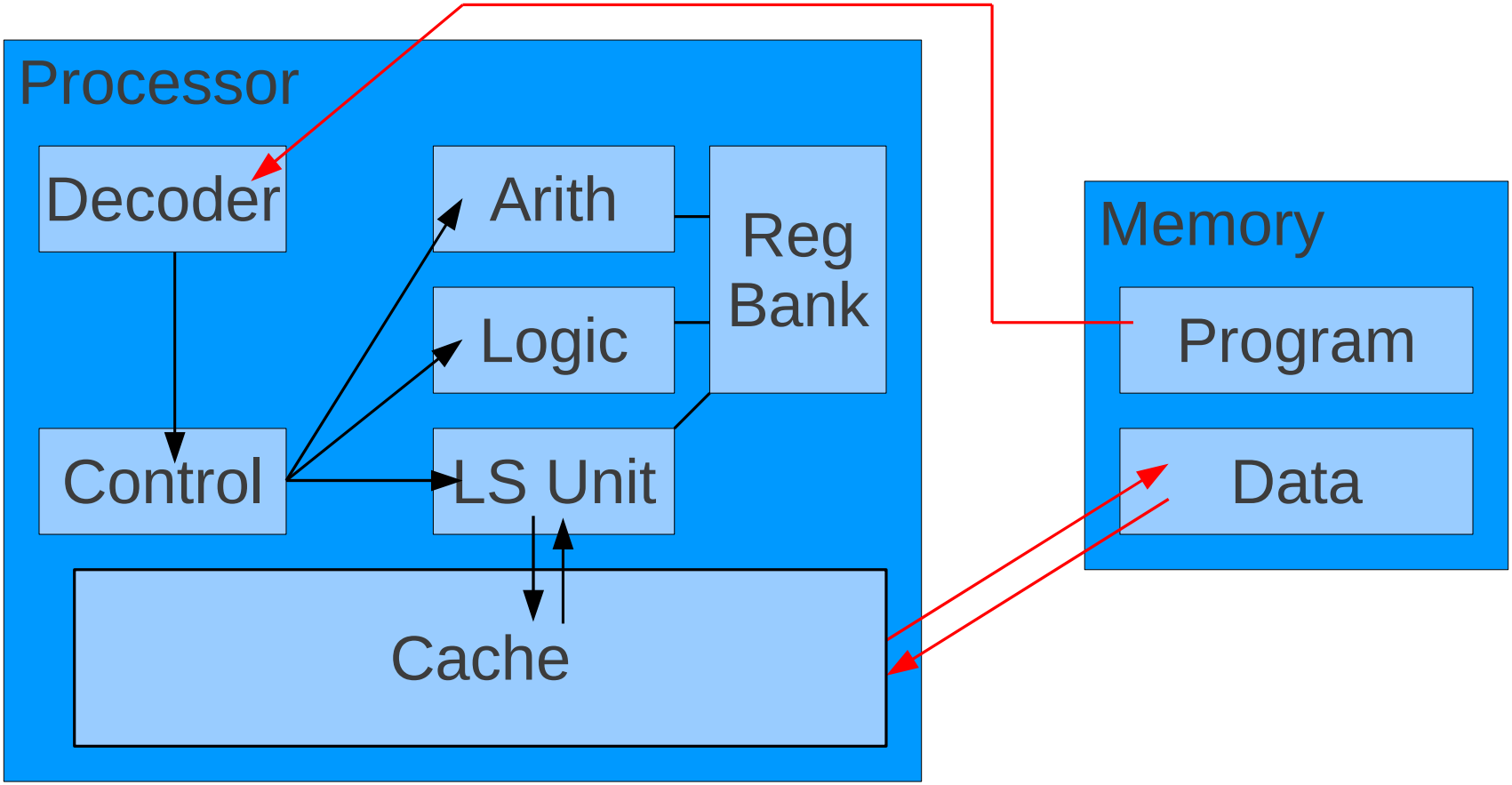
- Very fast memory on the processor die
- Need to be connected to everything
- Programmer control of how to hold (cache) data



Processor Bottlenecks III

Cache is expensive

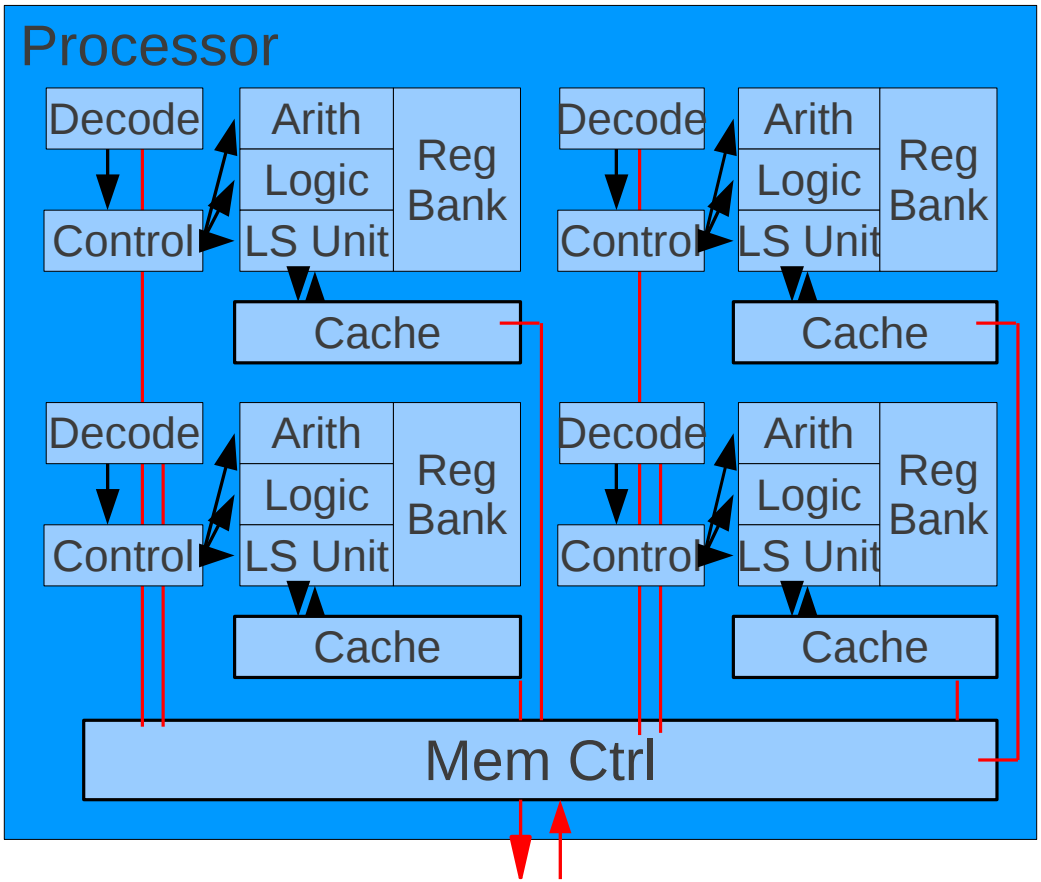
- Slightly slower and cheaper than registers
- Processor controls what is stored
- Anticipates usage based on space / time



Parallelism

Still not fast enough

- We always need more speed
- What about running different parts of the program at once?
- Chip designers know about cut'n'paste ...



Parallelism II

Arbitration is expensive

- Many things competing for the same resource
- Something has to decide on their order of access

Execution is discrete

- In one clock-cycle, one part can only perform one task
- As before, bandwidth is cheap, latency is expensive
- If several parts have to wait for their access slot latency increases

If we cannot remove a problem (physically)

- It appears in the programming model
- Abstractions are leaky
- There is no software fix for latency issues in the hardware

Parallelism IV

We've pushed our MIMD architecture as far it will go (this year...)

- Large die = High cost (low yields)
- High clock-speed = High power consumption = Hot

Two problems to solve

- How to cram more cores into the same area
- How to handle latency

SIMD

Single Instruction Multiple Data

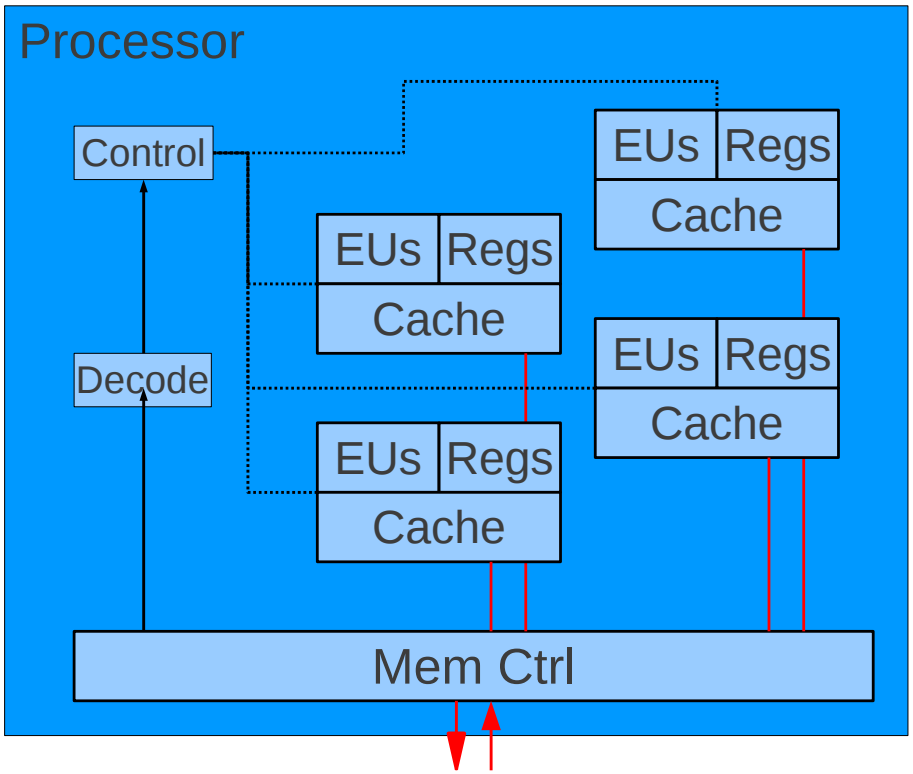
- Also called a vector architecture

Each processing core execute the *same* program

- Lots of copies of the program; each core in *lock-step*
- Only the data varies

Different programming model

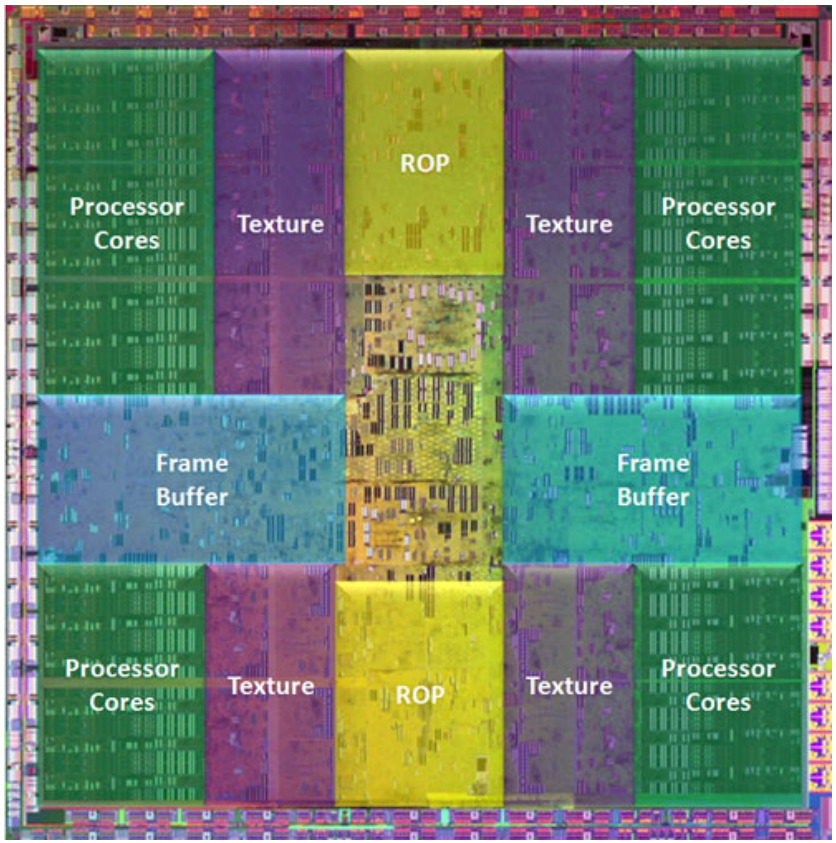
- Not suitable for every problem
- Good at Linear Algebra
- Tricky part is fitting other problems into the restricted model

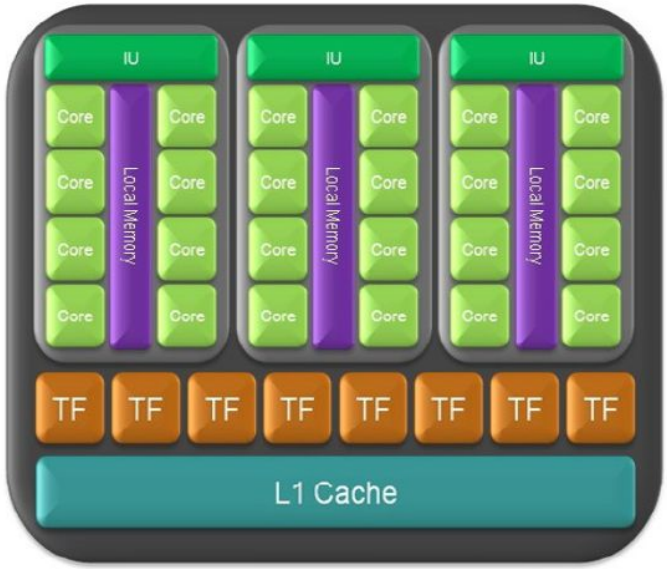


SIMD II

Reusing control / decode removed half the arbitration problem

- The other half is the caches
- But each core is now running the same program. . .
- There should be a similarity between the memory access patterns
- So the caches can cooperate. . .





Threading

The chip is *full* of processing cores

- Caches reduce the memory arbitration as much as possible
- But. . . memory is still far too slow
- About 500 cycles to main memory

Stop execution, start something else

- Exactly how an OS hides I/O latency by multi-tasking
- In the O/S case a full context-switch is needed (expensive)
- Require hardware support for fast thread-switching

The Multi-processors in a GPU

- Use register banks with tens of thousands of registers
- New thread is start by changing a base pointer
- Zero-cycle switching between threads
- But you still need enough to hide the latency