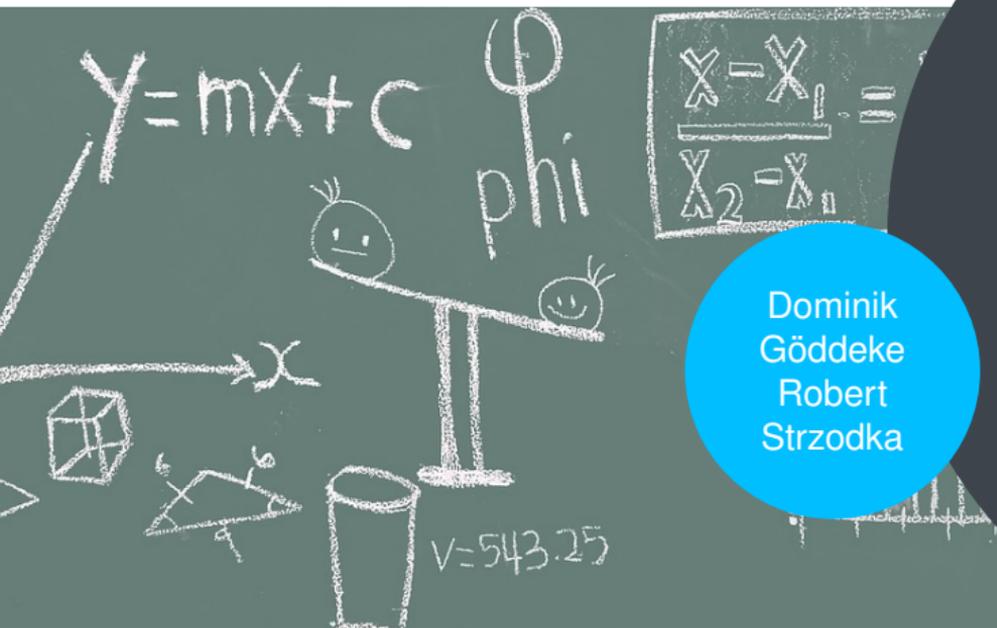


University of Stuttgart
Germany



Modern GPU Computing

Session 1

September 2019

Dominik
Göddeke
Robert
Strzodka

Introduction

Long tradition of GPU tutorials at PPAM

- Today: 12-year anniversary, 7th PPAM in a row

Presenters

- Dominik Göttsche
Chair Computational Mathematics, University of Stuttgart
- Robert Strzodka
Chair Application Specific Computing, Heidelberg University
- Manuel Ujaldón
Computer Architecture Department, University of Malaga

Introduction

Target audience

- Both beginners and experts, see next slide

Course goals

- Session 1: short overview and recap on CUDA
- Session 1: introduction to the DLI certificate offer
- Sessions 2+3: some advanced and recent concepts in detail, and their impact on programming and algorithmic thinking
- Large practical with experiments on recent hardware

Introduction

Course outline

- 11:30–13:00: Session 1, overview, info about the DLI certificate [entry-level]
- 13:00–13:30: Lunch
- 13:30–15:00: Session 2, unified memory, new Volta features [advanced]
- 15:00–16:30: Practical exercises, coffee break (15:30–16:00)
- 16:30–18:00: Session 3, optimizing for Volta and Turing and the DGX-2 architecture [advanced]
- 19:30: Welcome reception and concert [entry-level]

Why GPUs?

Why GPUs?

The exa-research answer

- GPUs are manycores, not multicores
- Contemporary architecture for algorithm design that requires massive degrees of concurrency for performance

The pragmatic answer

- GPUs are (can be) fast, some examples from our own work
- Seismic wave propagation modeling: 10–20x faster
- Sparse iterative solvers: 5–30x faster
- Lattice-Boltzmann methods: 5x
- Runge Kutta methods for multiplexing in optical fibres: 30–50x faster

CUDA overview

CUDA ecosystem, rationale why we focus on CUDA

CUDA ecosystem

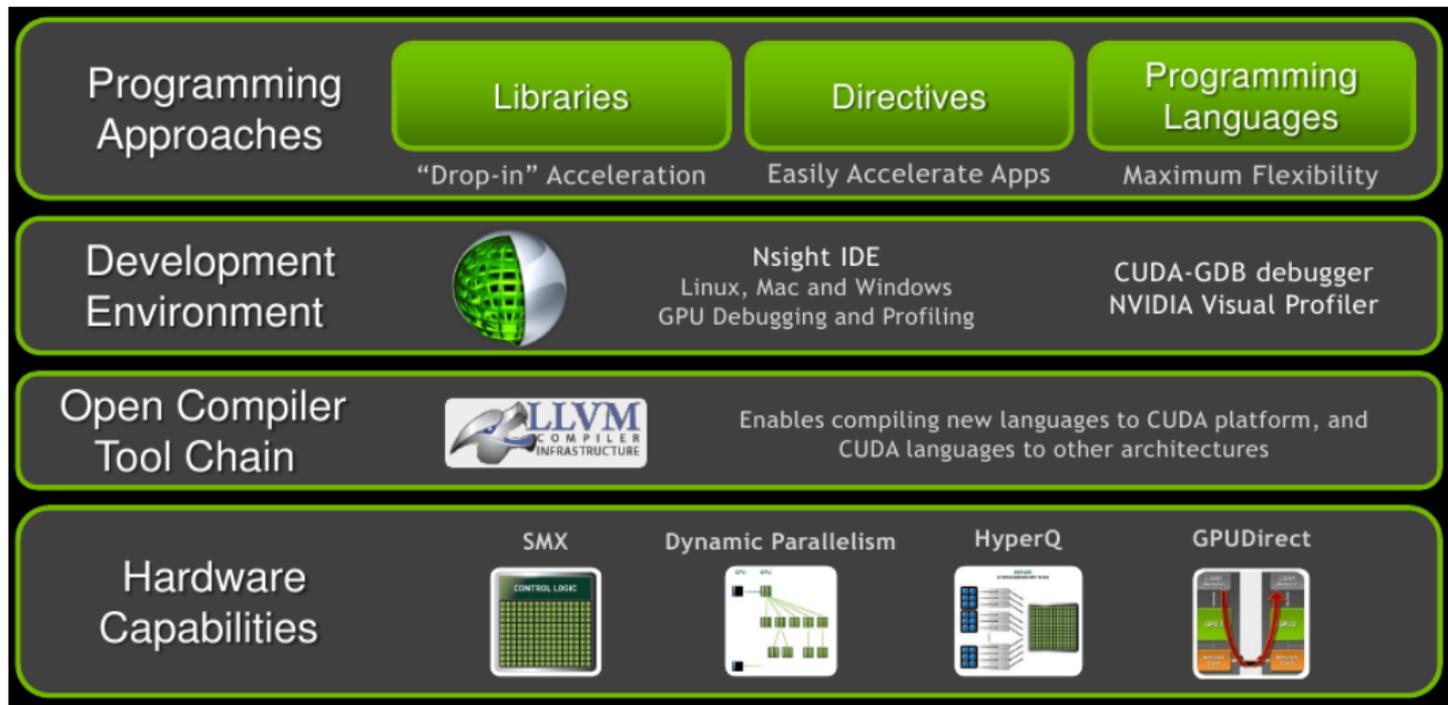


Figure courtesy Will Ramey, NVIDIA

GPU accelerated libraries

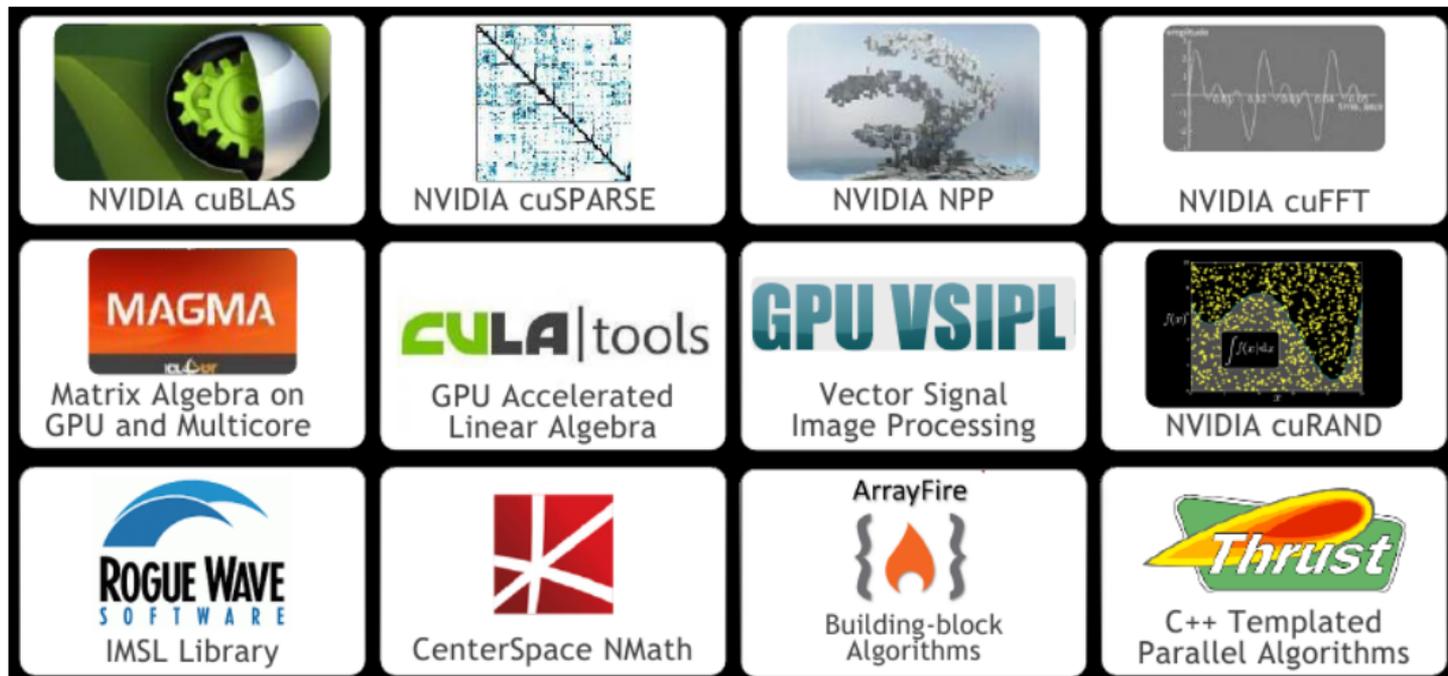


Figure courtesy Will Ramey, NVIDIA

GPU-aware MPI implementations

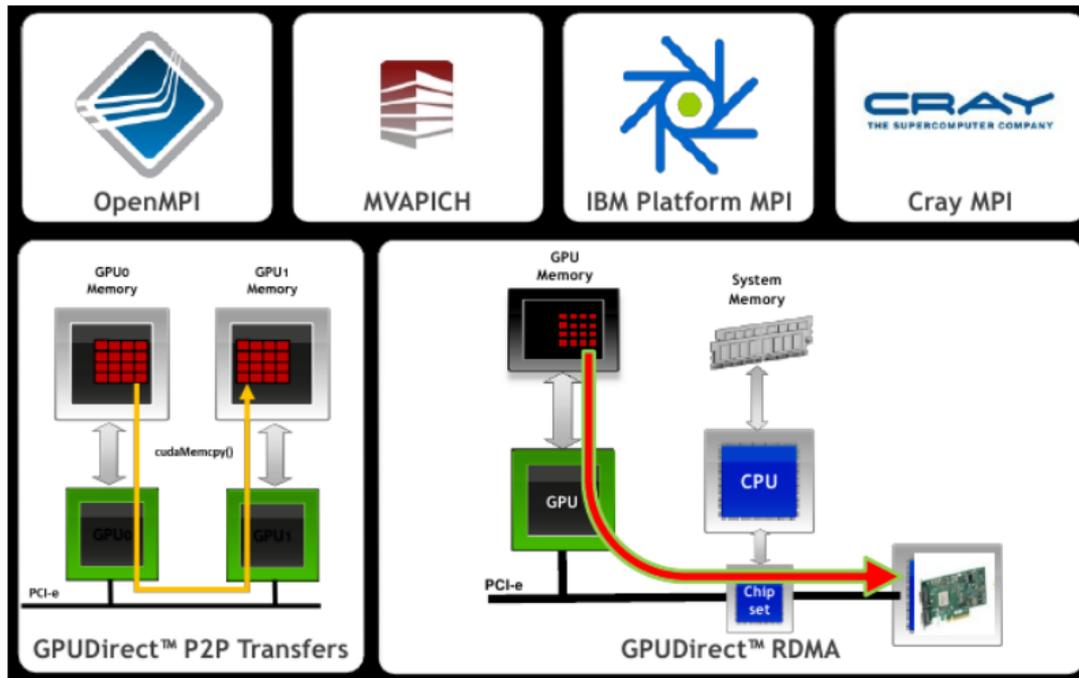
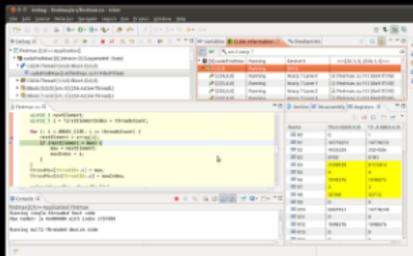
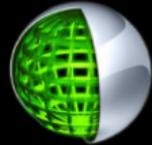


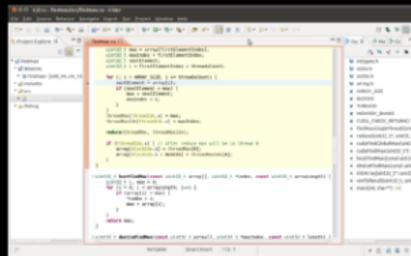
Figure courtesy Will Ramey, NVIDIA

NVIDIA® Nsight™, Eclipse Edition for Linux and MacOS



CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs



Nsight Debugger

- Simultaneously debug CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging



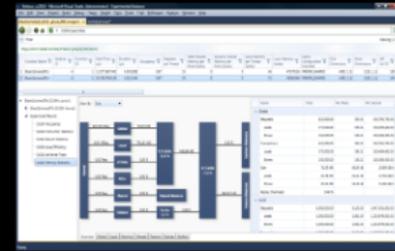
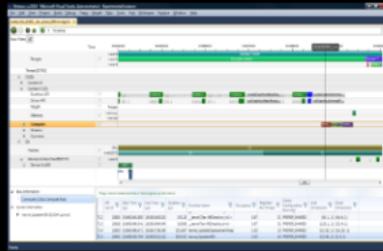
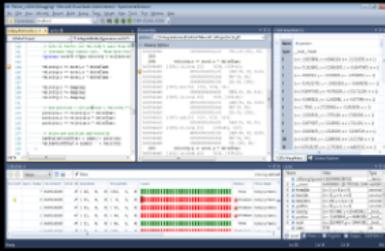
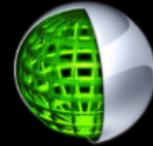
Nsight Profiler

- Quickly identifies performance issues
- Integrated expert system
- Source line correlation

developer.nvidia.com/nsight

Figure courtesy Will Ramey, NVIDIA

NVIDIA® Nsight,™ Visual Studio Edition



CUDA Debugger

- Debug CUDA kernels directly on GPU hardware
- Examine thousands of threads executing in parallel
- Use on-target conditional breakpoints to locate errors

CUDA Memory Checker

- Enables precise error detection

System Trace

- Review CUDA activities across CPU and GPU
- Perform deep kernel analysis to detect factors limiting maximum performance

CUDA Profiler

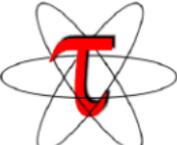
- Advanced experiments to measure memory utilization, instruction throughput and stalls

Figure courtesy Will Ramey, NVIDIA

Analysis tools

Performance Analysis Tools

Single Node to Hybrid Cluster Solutions

 <p>NVIDIA Nsight Eclipse & Visual Studio Editions</p>	 <p>NVIDIA Visual Profiler</p>	 <p>Vampir Trace Collector</p>
 <p>TAU Performance System Tuning and Analysis Utilities</p>	 <p>PAPI PAPI CUDA Component</p>	 <p>HPCToolkit Under Development</p>

developer.nvidia.com/performance-analysis-tools

Figure courtesy Will Ramey, NVIDIA

Debugging

Debugging Solutions Command Line to Cluster-Wide



NVIDIA Nsight
Eclipse & Visual Studio Editions



NVIDIA CUDA-GDB
for Linux & Mac



NVIDIA CUDA-MEMCHECK
for Linux & Mac



Allinea DDT with CUDA
Distributed Debugging Tool



TotalView for CUDA
for Linux Clusters

developer.nvidia.com/debugging-solutions

Figure courtesy Will Ramey, NVIDIA

CUDA recap

CUDA versions (software)

Compatibility

- Obviously, the more recent the version, the more features and GPUs are supported and available (we cover CUDA 10)
- Ideal world: same driver and toolkit version
- Real world: toolkit \leq driver (backward compatibility)
- Warning: support for \leq Fermi is deprecated since CUDA 8.x

CUDA GPU-sided 'functions' use extended C/C++ syntax

- Consequence: NVIDIA CUDA Compiler `nvcc` necessary
- `nvcc` is a compiler wrapper
- Must be able to 'see' the system compiler
 - Linux: default distro gcc compiler and kernel version
 - Similarly on Windows: fixed version of Visual Studio

CUDA versions (hardware)

Obvious fact

- Different generations of CUDA-capable GPUs support different features
- Rule of thumb: feature support is incremental

Dedicated compute capabilities reflect this

- Compute capability = fixed (guaranteed) feature set and fixed technical specification
- Example: Only 3.5 and above devices support unified memory
- Timeline: first two Tesla generations (cc 1.x), Fermi (cc 2.x), first generation Kepler (cc 3.0), second generation Kepler (cc 3.5), different Maxwell (cc 5.x), Pascal (cc 6.x), Volta (cc 7.0) and Turing (cc 7.5)

CUDA versions (hardware)

Compute capabilities cont.

- See Appendix G.1 in the CUDA C Programming Guide for the full mapping of features to capabilities
- See also Appendix G.2 for details on different compliance with the IEEE 754-2008 floating point standard
- We cover mostly Volta and Turing

Figuring out compute capabilities

- Impossible from brand names due to marketing strategies
- Full list: Appendix A of the programming guide
- Programmatically: next slides

Device properties and compute capabilities

CUDA API first contact

```
#include <iostream>
#include <cuda_runtime.h>
int main (int argc, char** argv) {
    int numDevices(-1);
    cudaError_t error = cudaGetDeviceCount(&numDevices);
    if (error != cudaSuccess) {
        std::cerr << cudaGetErrorString(error) << std::endl;
        return 42;
    }
    std::cout << "NumDevices:" << numDevices << std::endl;
    return 0;
}
```

- Minimalistic C++ program to query the number of GPUs using the runtime API
- A single CUDA include, not necessary for compilation with `nvcc`

CUDA API first contact

Compilation trajectory 1

- `nvcc 00-deviceinfo.cpp`
- Automagically sets all paths etc., include not even necessary
- Defaults to whatever default the host compiler (gcc etc.) uses, and to full optimisations for GPU code

Compilation trajectory 2

- Standard GCC include/lib approach: `g++ -I$CUDA_INC_PATH 00-deviceinfo.cpp -L$CUDA_LIB_PATH -lcudart`
- Other systems: installer informs you at the end (Linux/Mac), template project (MSVC), NSight IDE

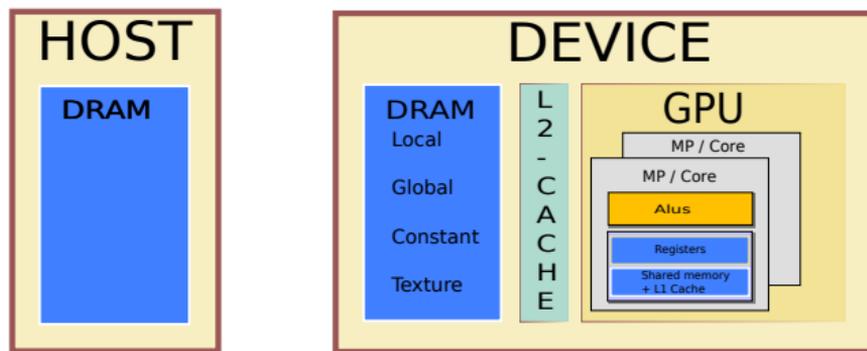
Device properties

```
[...]  
error = cudaSetDevice(2); // or whichever  
if (error != cudaSuccess) {...}  
cudaDeviceProp prop;  
error = cudaGetDeviceProperties (&prop, 2); // or ...  
if (error != cudaSuccess) {...}  
std::cout << prop.name;  
std::cout << " (cc " << prop.major << "." << prop.minor << ")";  
std::cout << std::endl;  
[...]
```

- Device selection defaults to first device if not used
- Warning: device enumeration may change between driver updates
- <http://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html>

CUDA memory model (old school)

CUDA memory model



- Host memory and device memory are separated
- Consequence: separate allocation, manual copying (or use unified managed memory, cc 3.5+, covered later)
- Performance consideration: moving data to/from device via PCIe is *very* expensive, both in terms of latency and bandwidth
- Physical memory is always addressed linearly, as on CPUs

Device memory allocation

`cudaError_t cudaMalloc((void**)&p, size);`

- Allocates a chunk of device memory of `size` bytes and returns a pointer to the starting address in `p`
- Example:

```
double *p; cudaError_t status =  
    cudaMalloc((void**)&p, 42*sizeof(double));  
if (status != cudaSuccess) // d'oh
```

- Note the similarity to standard C-style memory allocation:

```
double *q = (double*) malloc(42*sizeof(double));  
if (q == NULL) // d'oh
```

`cudaError_t cudaFree(p)`

- Releases the chunk of memory `p` points to

Memory copies

`cudaError_t cudaMemcpy(to,from,size,direction);`

- Copies `size` bytes from the memory location `from` to the memory location `to`
- Exactly the same as C `memcpy`, except for return type and direction
- `direction`: one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`
- `from` and `to`: pointers to host or device memory
- Example using the `p` and `q` arrays from the previous slide:

```
cudaError_t status =  
    cudaMemcpy(p,q,42*sizeof(double),cudaMemcpyHostToDevice);  
if (status != cudaSuccess) // something went wrong
```

and

```
cudaError_t status =  
    cudaMemcpy(q,p,42*sizeof(double),cudaMemcpyDeviceToHost);  
if (status != cudaSuccess) // something went wrong
```

Device memory management

Device pointers are standard pointers

- Can construct pointered structures easily, just like on the CPU
- Pointer arithmetic completely legal if you really need it, except of course pointer arithmetic involving host *and* device pointers
- Must not dereference device pointers on host or vice versa, unless when using unified memory (session 3)
- BSoD or segfault likely to occur otherwise

Address space

- 32-bit for compute capability 1.x, 49-bit for Pascal (cc6.x), 40-bit inbetween
- Keep this in mind as it restricts the maximum length of a chunk of memory

CUDA programming model

From OpenMP to CUDA

A very simple operation: vector addition

- Each iteration completely independent of all other iterations

```
for (int i(0); i<N; ++i)
    c[i] = a[i] + b[i];
```

Standard OpenMP implementation

```
#pragma omp parallel for default(shared) schedule(static)
for (int i(0); i<N; ++i)
    c[i] = a[i] + b[i];
```

- Under the hood, OpenMP partitions the iteration into $\lceil N/\text{numCores} \rceil$ contiguous chunks, each chunk is treated sequentially by exactly one core
- Assuming we haven't fiddled with `OMP_NUM_THREADS` etc.

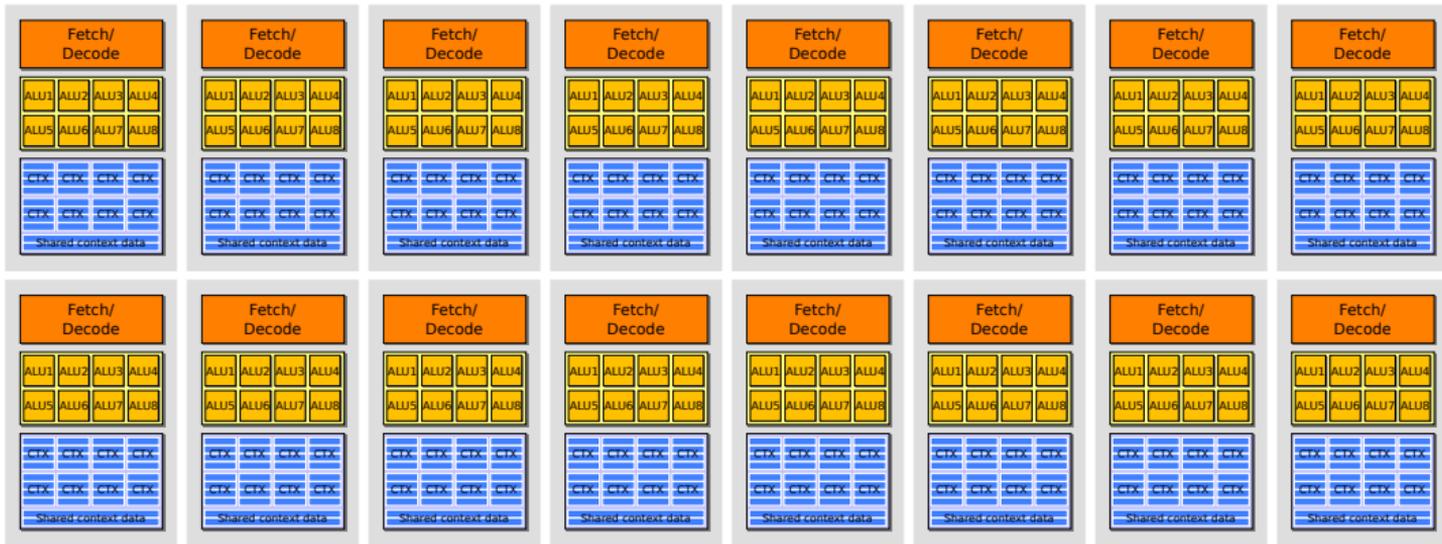
From OpenMP to CUDA

A slightly different OpenMP implementation

```
#pragma omp parallel for default(shared) schedule(static,256)
for (int i(0); i<N; ++i)
    c[i] = a[i] + b[i];
```

- Note: 256 completely arbitrary (for now), just a reasonable small-digit number
- Under the hood, OpenMP partitions the iteration into $\lceil N/256 \rceil$ contiguous chunks, each chunk is treated sequentially by exactly one available core, chunks are assigned to cores in arbitrary order (no guarantee whatsoever on the execution sequence of these 'blocks')
- Single most important statement in this part:
A-ha! This is almost exactly the way basic (old-school) CUDA works

Making sense of the bold similarity claim



- GPU idea 1+2: lots of simple SIMD cores (details in 15 minutes)

Making sense of the bold similarity claim

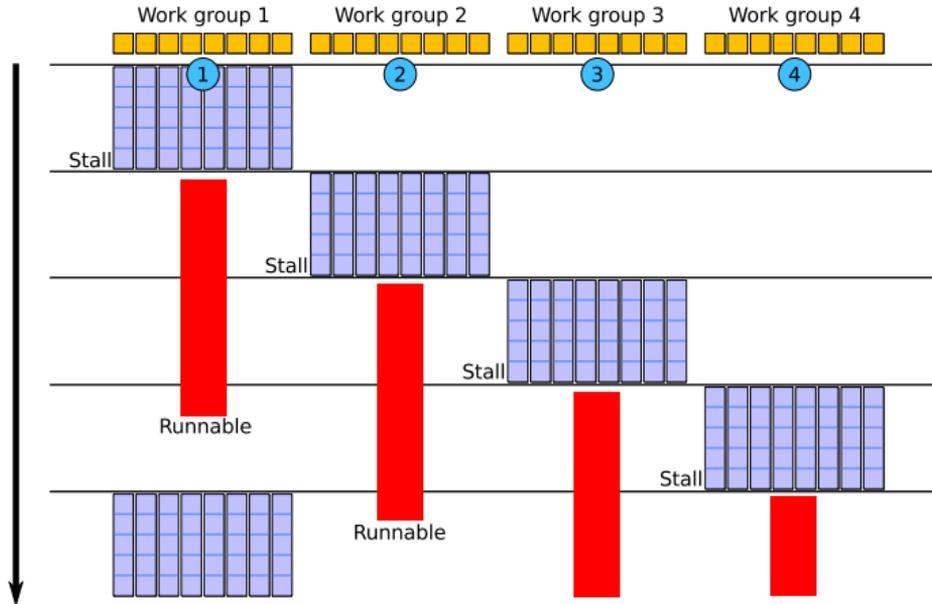
Only one relevant conceptual difference on GPUs

- Chunks (blocks, work groups, etc.) are not treated sequentially by one CPU thread on one CPU-style core
- But instead by (here) 256 conceptionally simultaneously running, concurrent GPU threads in one of these ‘simple SIMD GPU-style cores’

Differences in detail

- Must (instead of ‘can’) specify the number of threads in each block (and thus, the size of each block) manually
- Because we are responsible that resources in each SIMD ‘core’ (streaming multiprocessor) are not over-used
- OpenMP does this automagically

Making sense of the bold similarity claim



- GPU idea 3: latency hiding through interleaved execution

Making sense of the bold similarity claim

Latency hiding

- CPU cores (e.g., with OpenMP) run one block to completion
- GPUs interleave blocks on stalls (e.g., due to a pending load instruction)
- Consequence: kick off (launch) much more of these blocks than physical cores available

High-level roundup

- If you *know how to think in parallel under resource constraints* already, then CUDA is 'just another language extension to learn'
- We'll explain how to express this in CUDA C++ in a minute
- Getting something up and running is indeed easy
- As always: the challenge lies in getting performance right

General CUDA programming approach

Generic pattern

- Invest thinking into which computation(s) should be performed by which thread in which block
- Invest even more thinking into getting the indexing (data access in global arrays) right
- Write a kernel, i.e., scalar (per-thread) code to be executed by each GPU thread separately
- Batch threads together in blocks, batch blocks together into a grid
- Launch (execute) the grid on the GPU

Example: vector addition

Mapping computations to threads

- Each GPU thread computes one element in the result array

Indexing

- Data structure is 1D linear array, so use 1D linear indexing
- Thread 0 computes $c[0]$, thread 1 computes $c[1]$, etc.
- Luckily, $c[i] = a[i] + b[i]$: same indexing in all arrays

Blocks and grids

- Threads on GPUs are not enumerated globally contiguous
- But instead in a hierarchical fashion
- Recall OpenMP `schedule(static,256)` example:
 - Computation within one 256-sized block now 'instantaneous'
 - Threads arranged into blocks, blocks into a grid

CUDA kernel for vector addition

```
__global__ void mykernel(int *c, const int *a,
                        const int *b, const int N)
{

    c[i] = a[i] + b[i];
}
```

- Kernel: function that executes on the GPU and is callable only from the CPU (for now)
- Keyword `__global__` indicates a kernel
- Kernel functions must be `void`, no way to get information back except through writes to GPU memory and `cudaMemcpy()`'ing (for now)
- Pointer arguments: device pointers by `cudaMalloc()`

CUDA kernel launch configuration for vector addition

```
dim3 block(256, 1, 1);  
dim3 grid( (unsigned)ceil(N/(double)block.x), 1, 1 );  
mykernel<<<grid,block>>> (c_gpu, a_gpu, b_gpu, N);
```

- dim3: triple of unsigned int, standard CTOR, public members: .x, .y, .z
- Define 1D blocks of size 256, i.e., of 256 threads each
- Create a grid with as many blocks as needed to add all N elements
- Use both in kernel launch (triple-chevron syntax)
- Creates $\lceil N/256 \rceil$ blocks, linearly enumerated but not necessarily linearly executed (same as in OpenMP)
- Creates $\lceil N/256 \rceil \cdot 256$ threads, linearly enumerated in each block

Back to vector addition kernel

```
__global__ void mykernel(int *c, const int *a,
                        const int *b, const int N)
{
    const int i(threadIdx.x + blockDim.x*blockIdx.x);

    c[i] = a[i] + b[i];
}
```

- Use enumeration of blocks in the grid, and of threads in the block, to compute a global thread index (let's call it i as usual)
- Use this index to access the arrays: each thread computes one element
- `threadIdx`: index of the current thread in the block (dim3)
- `blockDim`: size of the current block (dim3)
- `blockIdx`: index of the current block in the whole grid (dim3)
- **Everything built-in and filled automatically**

Back to vector addition kernel

```
__global__ void mykernel(int *c, const int *a,
                        const int *b, const int N)
{
    const int i(threadIdx.x + blockDim.x*blockIdx.x);
    if (i<N)
        c[i] = a[i] + b[i];
}
```

- Problem if $\lceil N/256 \rceil \neq N/256$: too many threads
- Out-of-bounds memory accesses to all arrays
- A simple conditional does the trick for the first N threads

First meaningful CUDA program

Demo code

- Allocate three large arrays on both CPU and GPU
- Perform vector addition on both architectures
- Copy GPU result back to CPU and perform comparison to make sure everything works correctly
- Source code: `02-vecadd.cu`

First meaningful CUDA program

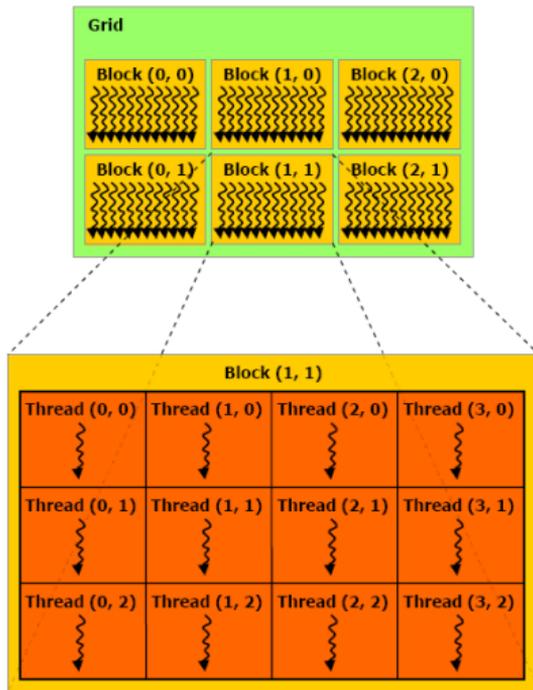
Important: error checking

- CUDA kernel launches are always asynchronous
- CPU free to do other work while GPU computes
- Consequence: kernel launches do not return a `cudaError_t`

Correct approach

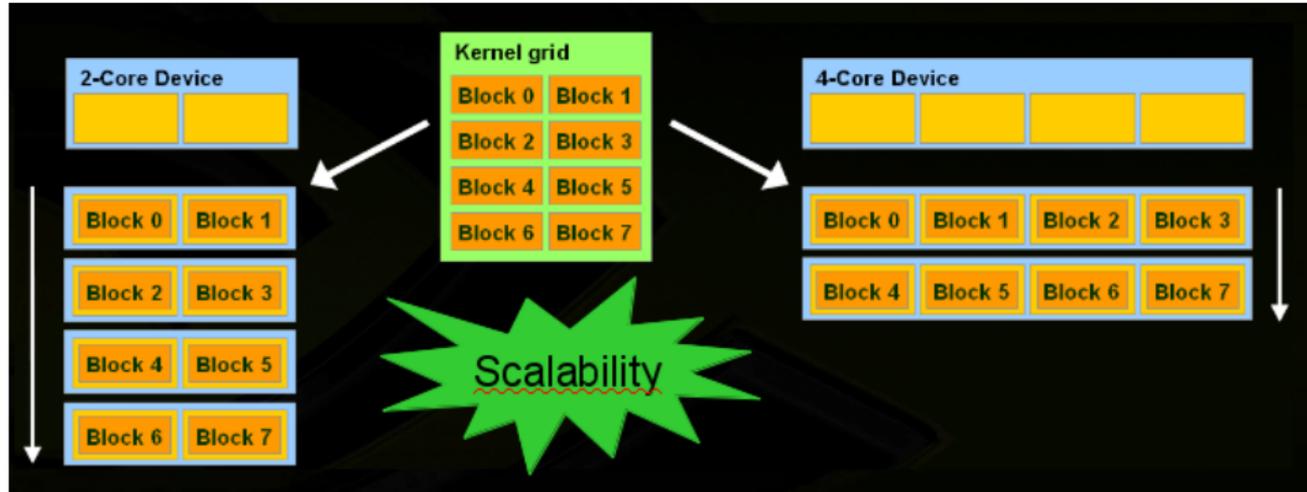
- Use `cudaGetLastError()` to query if an error occurred
- Manually synchronise with `cudaDeviceSynchronize()` beforehand to make sure the kernel is completed
- Also useful for CPU-sided timings of kernels (otherwise, kernel executes seemingly in 0.00000001 seconds)
- Better, more fine-grained synchronisation and timings: CUDA Events

The programming model in a nutshell: Grid of blocks of threads



Copyright: NVIDIA, source: CUDA C Programming Guide

The programming model in a nutshell: Blocks must be independent



Copyright: NVIDIA, source: CUDA C Programming Guide

More details: DLI course

Summary

Write code for a single thread

- Parameterise code by values from built-in variables so that each thread knows the data portions it is responsible for
- Make sure computations and memory accesses are SIMD-able (later)

Arrange threads into independent blocks

- Blocks must execute in arbitrary order
- Threads in a block may synchronise during execution, threads from different blocks cannot (more later)
- Respect physical resources, maximise interleaving of blocks (later)

Launch all blocks simultaneously in a grid

- Kernel launches are asynchronous on the CPU, control returns immediately so that CPU can do something else in between

Building a mental model of GPU hardware

Acknowledgements and plagiarism alert

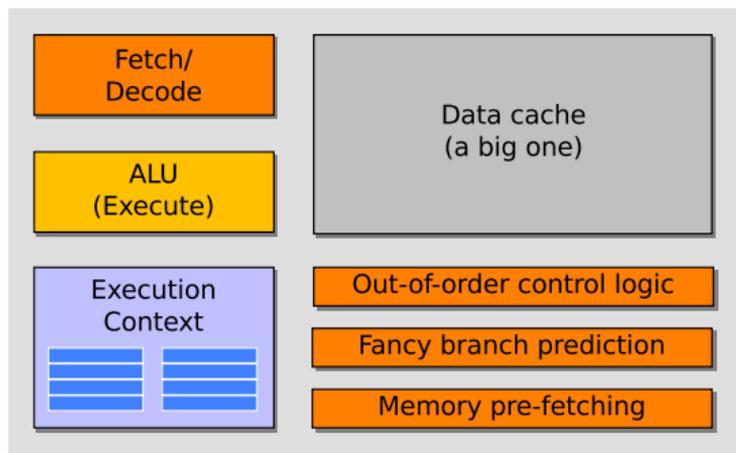
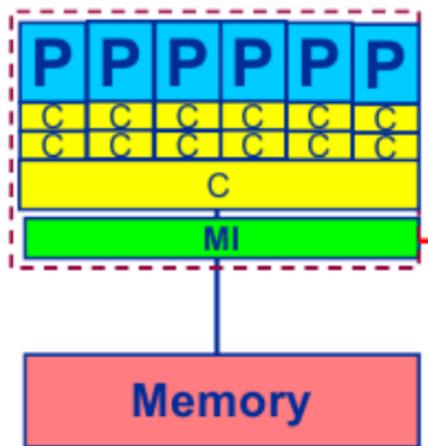
Kayvon Fatahalian (Stanford University)

- Closely modeled on Kayvon's 'classic' talk *From Shader Code to a Teraflop: How GPU Shader Cores Work* delivered at various SIGGRAPH tutorials since 2008
- Expanded and updated to suit our needs where necessary (Kayvon addresses computer graphics folks)

Reading recommendations (albeit outdated)

- <http://graphics.stanford.edu/~kayvonf/>
- K. Fatahalian and M. Houston: *A closer look at GPUs*, Communications of the ACM. Vol. 51, No. 10 (October 2008)
- M. Garland and D. B. Kirk: *Understanding throughput-oriented architectures*, Communications of the ACM. Vol. 53, No. 11 (November 2008)

Starting point: schematic view on CPU cores



- 'Execution context': storage for per-thread private data and state
 - OpenMP: thread ID, local start and end loop bounds etc.
 - CUDA: blockIdx, threadIdx, blockDim, etc.
- Typically realised as registers, very fast
- Two execution contexts? hyperthreading for interleaved execution

Starting point: schematic view on CPU cores

Benefits of all the extra logic?

- CPUs must be reasonably good for all workloads (compromise)
- CPUs minimise latency of a single instruction

GPUs are radically different

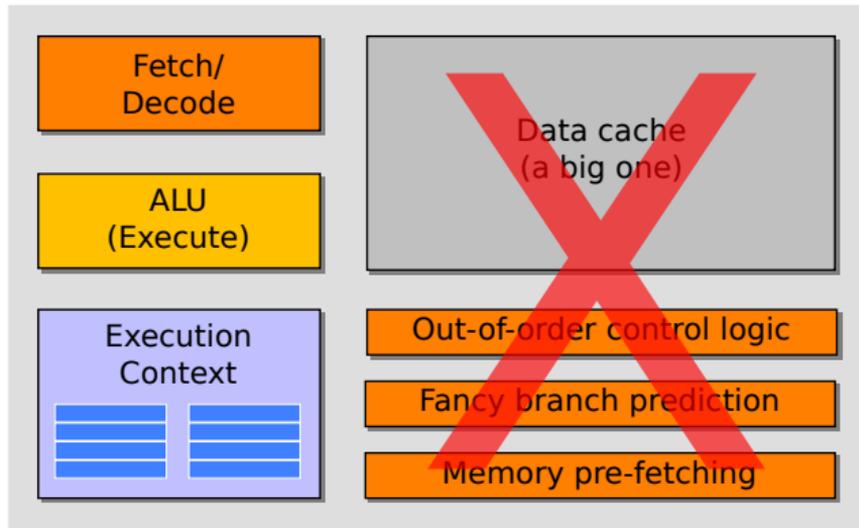
- Optimised for (data-) parallel workloads
- Maximise throughput of many instructions in a given interval

Goal of this part

- Come up with a mental model of GPU hardware
- And a corresponding programming model that forces us to think more about the underlying hardware if we want performance
- My experience: optimising for the GPU typically leads to many ideas how to optimise for the CPU as well, so win-win

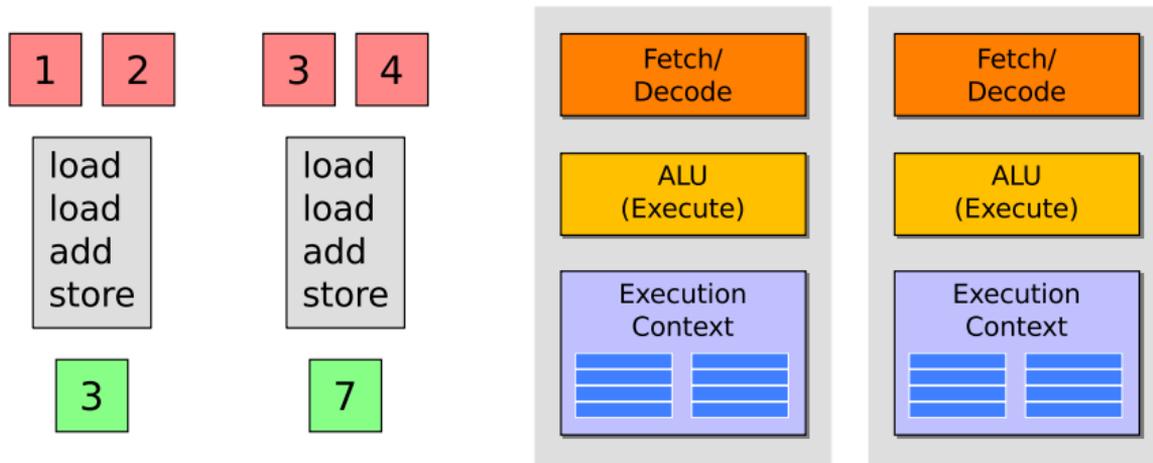
Idea 1: simplification

Idea 1: simplification



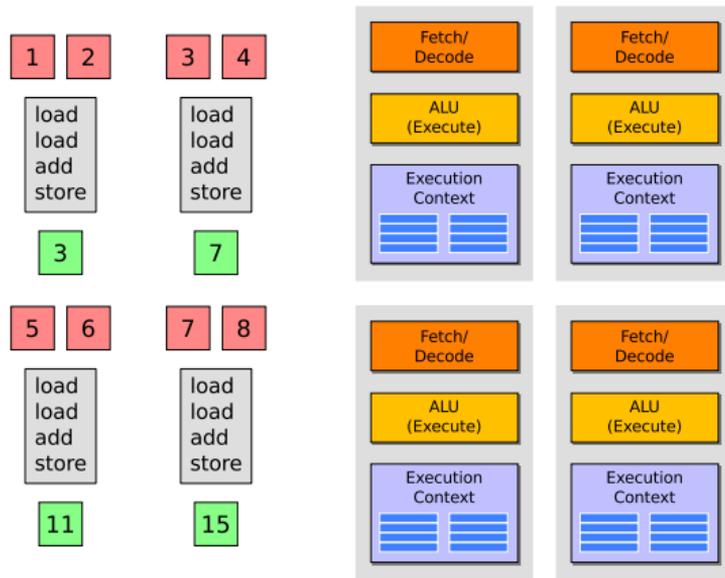
- Remove everything that makes a *single* instruction stream run fast
- Caches and hard-wired control logic, up to 50% of die area in typical CPUs

Idea 1: simplification



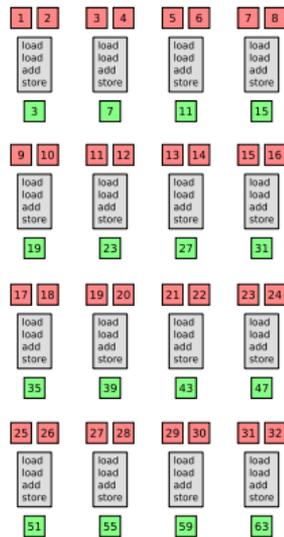
- Invest saved die area and transistors into more clones of the simplified core
- ‘more copies’: More than a conventional multicore CPU could afford
- Two simultaneous instruction streams on two cores: two operations in parallel, twice as fast unless we hit the memory wall

Idea 1: simplification



- Four simultaneous instruction streams on four cores: four operations in parallel

Idea 1: simplification



- Sixteen simultaneous instruction streams on sixteen cores: sixteen operations in parallel

Idea 2: SIMD

Idea 2: SIMD processing

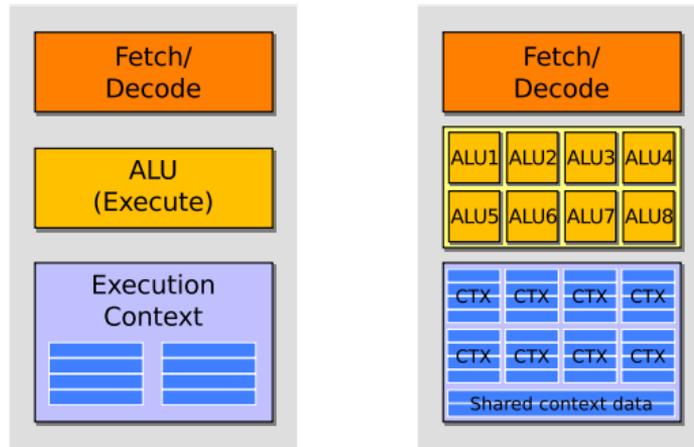
GPU = graphics processing unit (at least, originally)

- Graphics pipeline, 30 000 ft view:
 - geometry processing
 - rasterisation
 - per-pixel lighting and texturing
- Ample parallelism:
 - Geometry processing: same affine transformation applied to many vertices in some surface triangulation
 - Pixel processing: same lighting effect applied to many pixels, same texture mapped to many pixels

This is called *data parallelism*

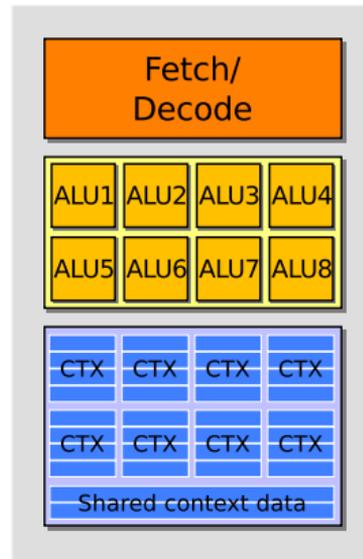
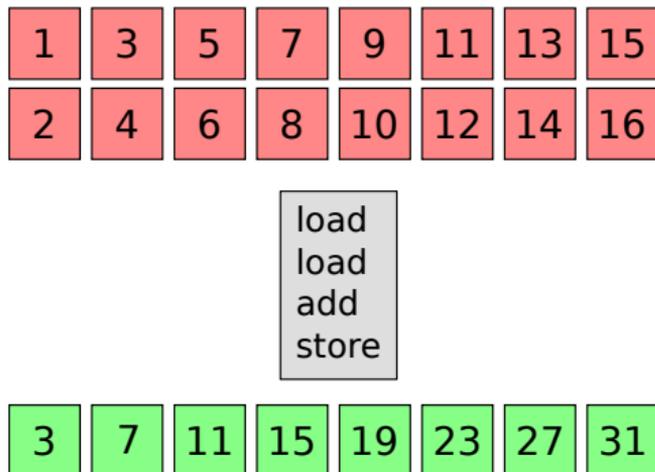
- Many other non-graphics operations exhibit the same ample ‘natural’ parallelism: examples throughout the course, ‘thinking in parallel’

Idea 2: SIMD processing



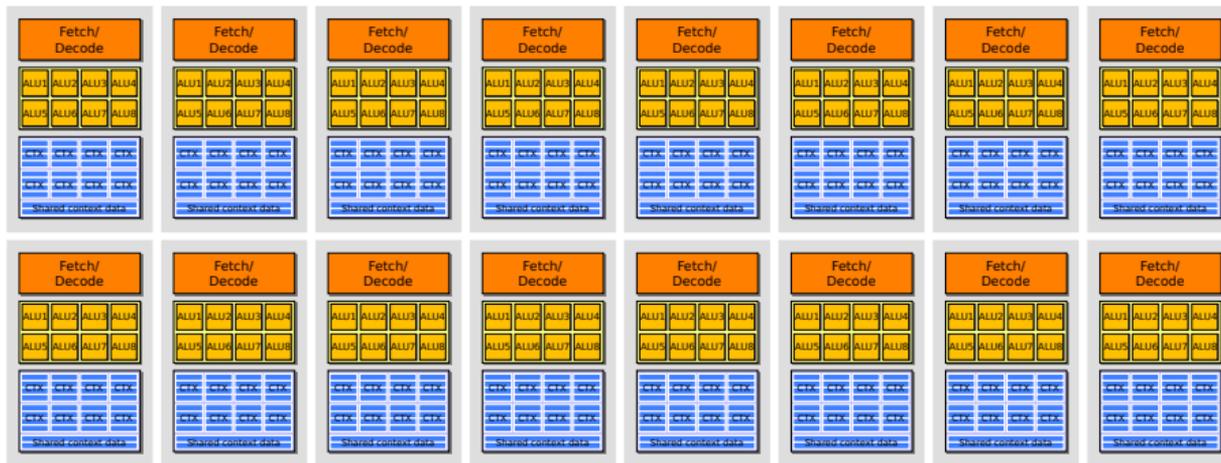
- Why run many *identical* instructions streams simultaneously?
- Amortise cost/complexity of managing an instruction stream across many ALUs to reduce overhead (ALUs are very cheap in HW!)
- Increase ratio of peak useful flops vs. total transistors → SIMD

Idea 2: SIMD processing



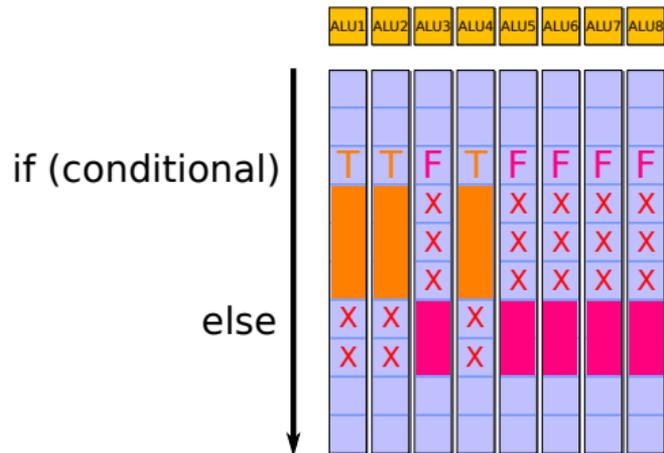
- One 8-wide instruction per core: 8 operations in parallel
- SIMD processing favours SIMD (=contiguous) memory access for performance

Idea 2: SIMD processing



- Sixteen simultaneous 8-wide instruction streams on 16 cores: 128 operations in parallel
- Increasing SIMD width pays off rapidly, obviously

Important SIMD drawback



- Branch (warp) divergence: threads need to take different code paths
- Impossible in the SIMD paradigm! Solution: masking of ALUs or result registers and serialisation of if and else branch
- Consequence: $1/\text{numALUs}$ of peak performance in the worst case

SIMD processing \neq SIMD instructions

Option 1: explicit vector instructions

- x,86, PPC, Xeon Phi etc.
- SSE, AVX, AVX-2, AVX-512, AltiVec
- Typical SIMD width: 128–512 bit
 - 4–16 parallel ops for single-precision floating point
 - 2–8 parallel ops for double-precision floating point

Using vector instructions

- Use built-in intrinsics, e.g. `__m128` data type on x86
 - Drawback: implies coding for a specific microarchitecture
- Let the compiler vectorise code
 - Drawback: doesn't (always) work for non-trivial access patterns

SIMD processing \neq SIMD instructions

Option 2: scalar instructions, implicit hardware vectorisation

- More programmer-friendly at the expense of more *implicit* knowledge
- HW determines instruction stream sharing across ALUs
- Amount of sharing deliberately hidden from software
- NVIDIA GPUs: SIMT *warps* of 32 threads each (SIMT = ‘single instruction multiple threads’ in NV-speak)
- AMD GPUs: *wavefronts* of 64 threads each

NVIDIA GPUs

- A *warp* is the SIMD group granularity level on CUDA GPUs
- Corresponding device property: `warpSize`
- 1 warp = 32 threads on all CUDA GPU generations so far
- Important difference: SIMD width always 32 values, and not some number of bits (independent of data type)

Idea 3: latency hiding

Idea 3: latency hiding

Stalls: delays due to dependencies in the instruction stream

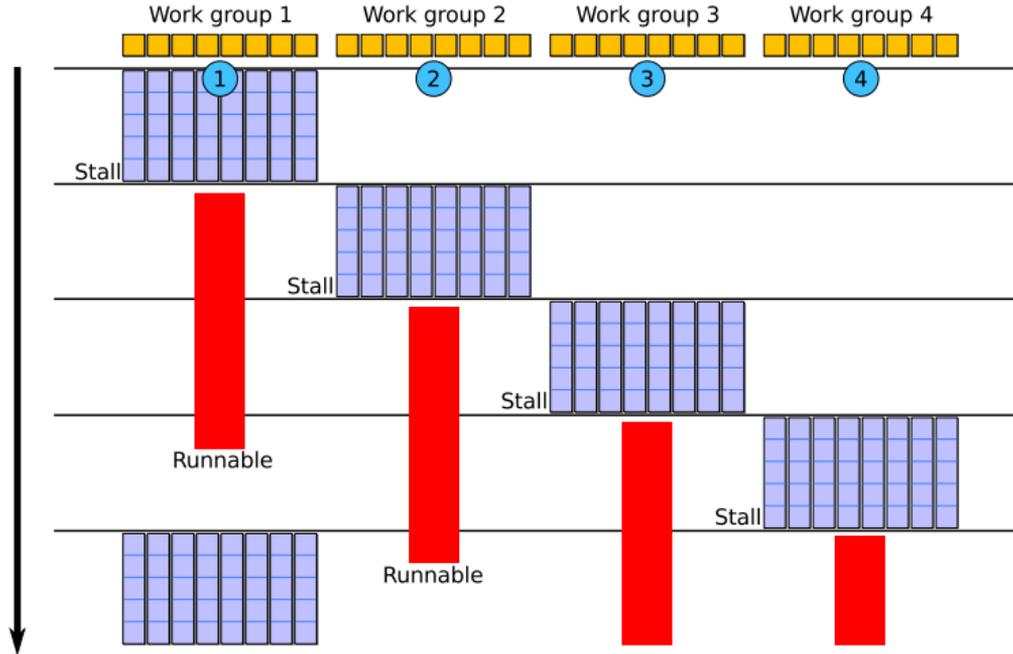
- Example: ADD operands depend on completed LOAD
- Latency: accessing data from memory easily takes 1 000+ cycles
- Simplification: fancy caches and logic that helps to hide stalls have been removed in idea # 1. Bad idea?

But: GPUs assume LOTS of independent SIMD groups

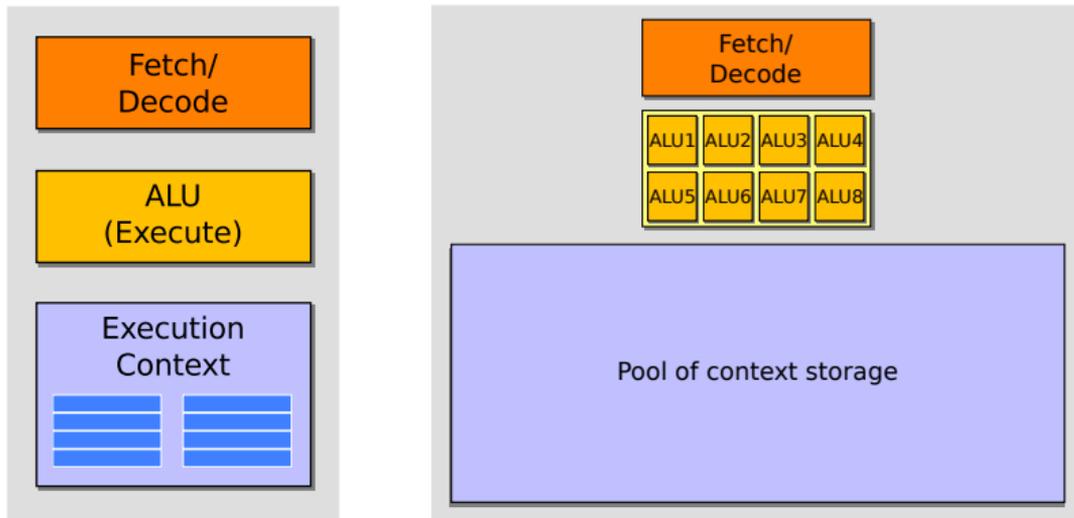
Idea 3: interleave many SIMD groups (warps) on a single core

- Switch to instruction stream of another (non-stalled = ready) SIMD group in case currently active group stalls
- Price to pay: need to store more contexts (private data+state)
- GPUs manage this in hardware, overhead-free
- Ideally, latency is fully hidden, throughput is maximised

Idea 3: latency hiding

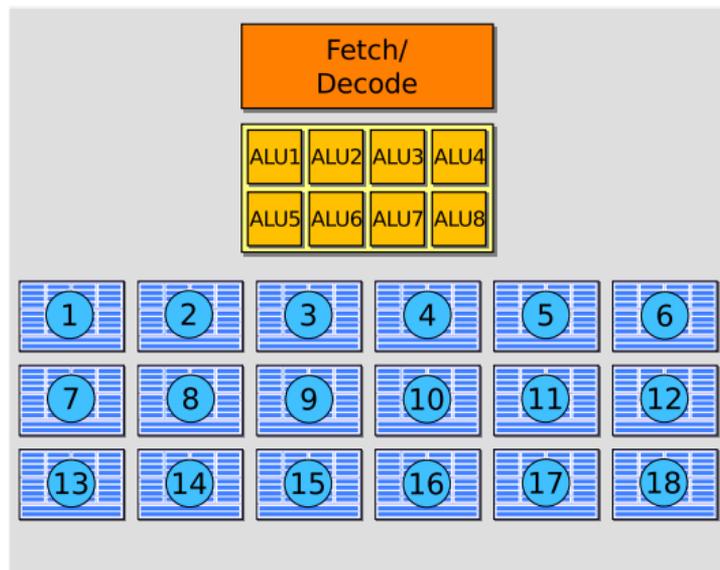


Idea 3: latency hiding



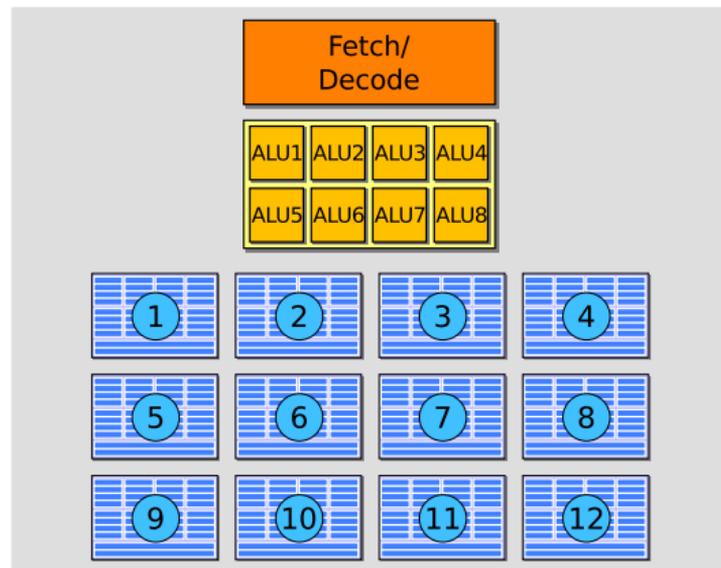
- Register file and shared context storage, this is a limited resource!
- Interleaving no longer possible if per-thread contexts cannot be stored

Idea 3: latency hiding



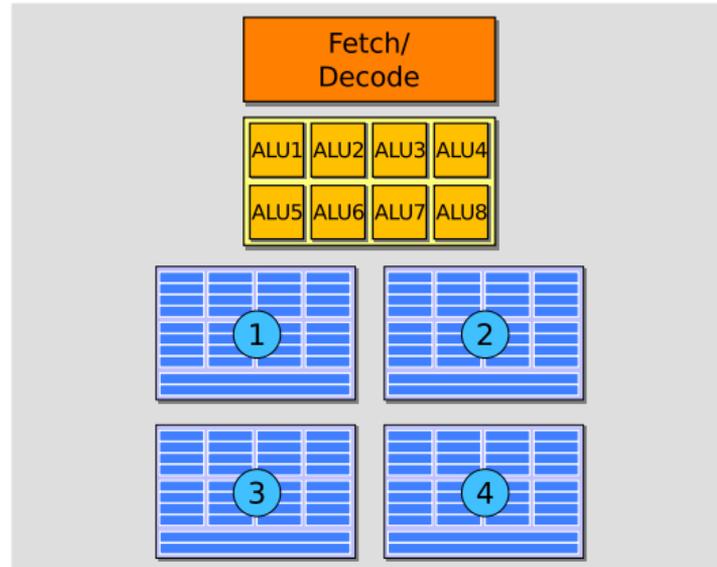
- 18 small contexts: high latency hiding possibility

Idea 3: latency hiding



- 12 medium contexts

Idea 3: latency hiding



- 4 large contexts: low latency hiding possibility

Summary

Three key ideas to maximise 'compute density'

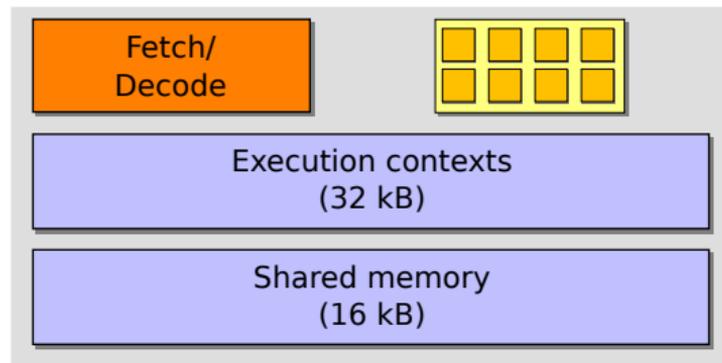
- Use many simplified cores to run in parallel
- Pack cores full of ALUs
 - By sharing instruction stream across SIMD groups
 - On GPUs by scalar instructions shared by a warp of true hardware threads
- Avoid latency stalls by interleaving execution of many SIMD groups
 - By throwing in some circuits for overhead-free essentially zero-time context switches in hardware

Historical evolution: idea 3.5

- Re-introduce some things that got cut away, like caches, but with potentially different purpose
- Add new special purpose units, like double precision units, tensor cores, raytracing cores, ...

Examples: real GPU designs

NVIDIA Tesla (first generation)



- First CUDA-capable GPU
- Figure shows a single core of this design (NVIDIA speak: streaming multiprocessor) with eight ALUs (NVIDIA: CUDA cores)
- Not shown: SFU units for transcendentals, LD/ST units
- 32 threads (one warp) share an instruction stream (4x instruction replay)

NVIDIA Tesla (first generation)

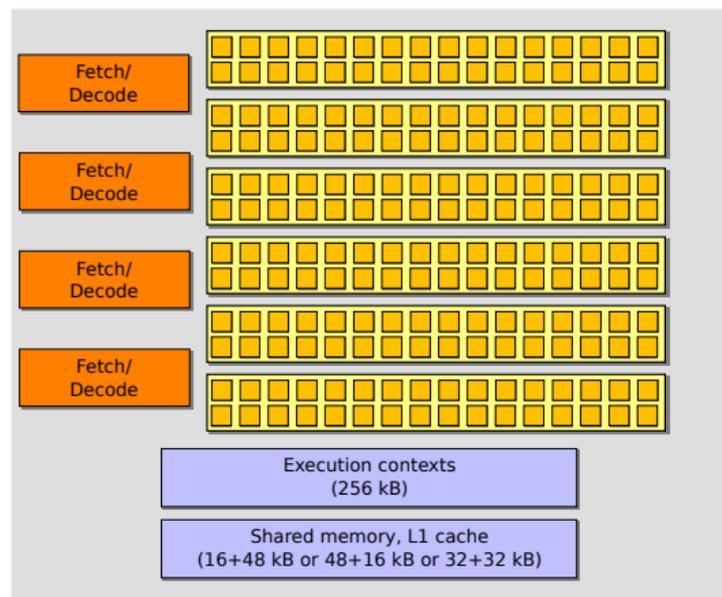
GeForce 8800 GTX (high-end model of this generation)

- November 2006
- 16 cores per chip: support for 12288 simultaneous threads
- 345.6 GFLOP/s peak SP, 0 GFLOP/s peak DP
- 86.4 GB/s peak memory bandwidth, 768 MB memory
- 155 W power consumption

GeForce 8600 GT (entry-level)

- April 2007
- 4 cores per chip; support for 3072 simultaneous threads
- 114.2 GFLOP/s peak SP, 0 GFLOP/s peak DP
- 22.4 GB/s peak memory bandwidth, 512 MB memory
- 43 W power consumption

NVIDIA Kepler (fourth generation)



- Figure shows a single core (NVIDIA: SMX) of this design
- Not shown: 32 SFUs, 32 LD/ST units, undisclosed # DP units

NVIDIA Kepler

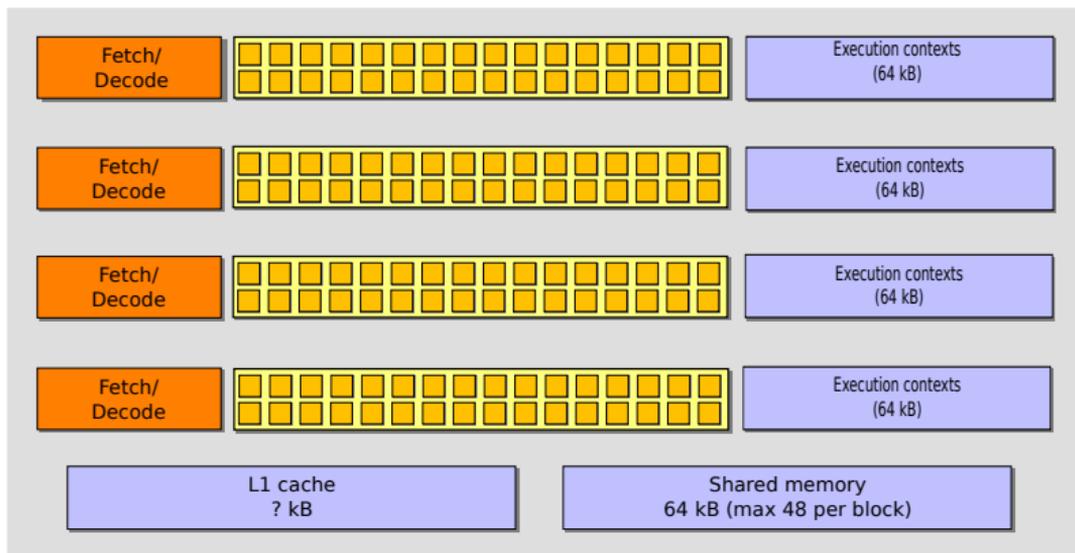
GeForce GTX 680

- March 2012
- 8 cores per chip: support for 16384 simultaneous threads
- 512 kB shared L2 cache (note: scales with #cores)
- 16, 32 or 48 kB L1 cache per core
- 3090 GFLOP/s peak SP, 129 GFLOP/s peak DP (realised by eight DP-ALUs per core, so 1/24 DP perf compared to SP)
- 192.4 GB/s peak memory bandwidth, 2 GB memory
- 195 W power consumption

Tesla K20, K20c, K40

- Professional versions
- Substantially improved over the GeForce (DP 1/2 SP)

NVIDIA Maxwell



- Figure shows a single core (NVIDIA: SMM) of this design
- Not shown: 8 SFUs, 8 LD/ST units, ? DP units per F/D unit

NVIDIA Maxwell

GeForce GTX 750 Ti

- March 2014, first Maxwell, cc 5.0
- 8 cores per chip: support for 16384 simultaneous threads
- 2048 kB shared L2 cache
- L1 cache per core undisclosed
- Full 64 kB shared memory per core
- First generation chip to demonstrate energy efficiency
- Mid-range performance

NVIDIA Maxwell

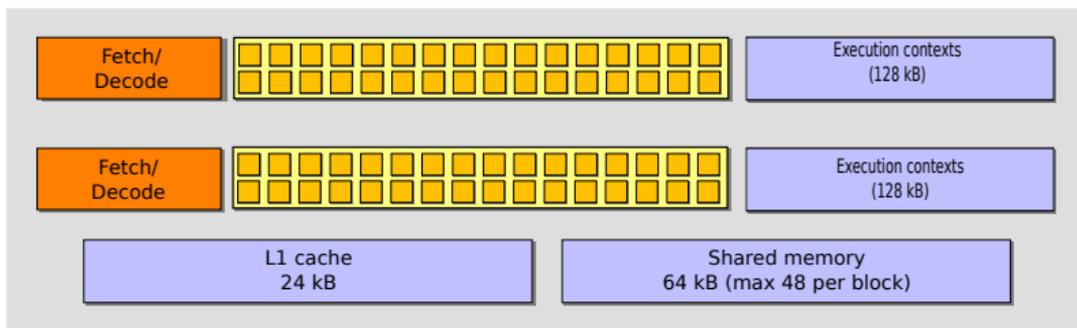
GeForce GTX 960

- January 2015
- 5 cores per chip: support for 10240 simultaneous threads
- 1048 kB shared L2 cache
- L1 cache per core undisclosed
- 96 kB shared memory per core (64 per block)
- DP @ 1/32 SP
- 112 GB/s peak bandwidth
- 2.3 TFLOP/s peak SP

Tesla K40, K80

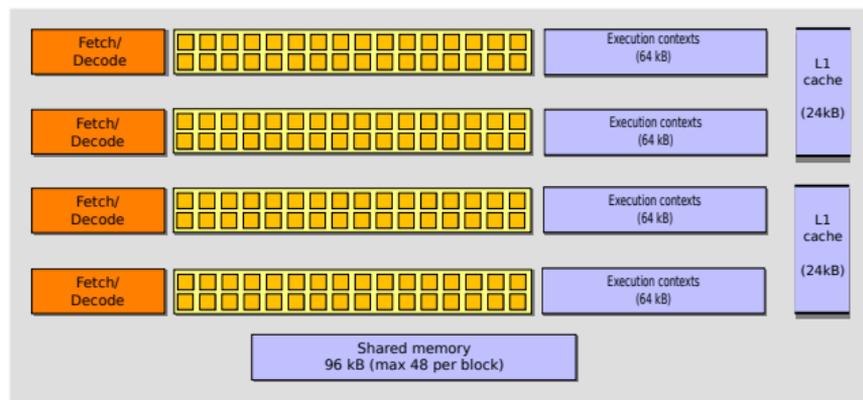
- Professional versions (1 or 2 GPUs)
- Substantially improved over the GeForce (DP 1/2 SP, ECC)

NVIDIA Pascal



- Figure shows a single core (NVIDIA: SM) of the first GP100 design
- Not shown: 8 SFUs, 8 LD/ST units, 16 DP units per F/D unit
- 2:1 ratio of single- to double-precision throughput

NVIDIA Pascal



- Figure shows a single core (NVIDIA: SM) of the second GP104 design
- Not shown: 8 SFUs, 8 LD/ST units, 1 DP units per F/D unit

NVIDIA Pascal

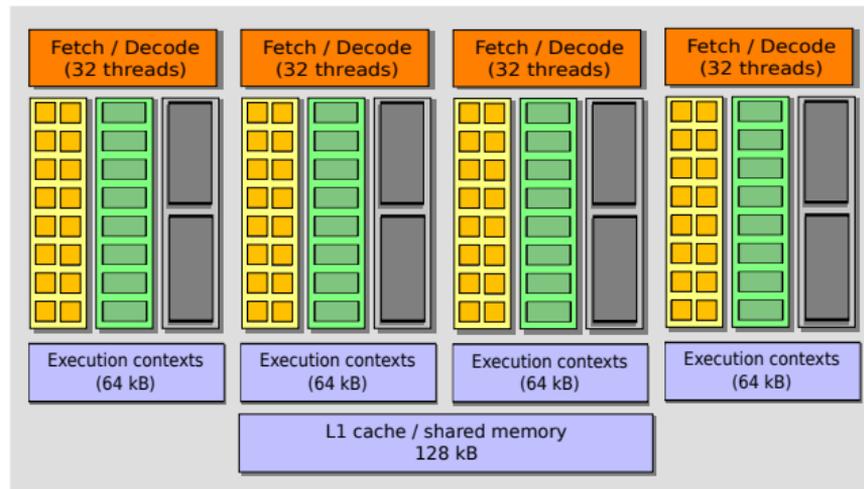
GeForce GTX 1060 (GP104)

- July 2016
- 10 SM (Streaming Multiprocessor) units: support for 40960 simultaneous threads
- 1536 kB shared L2 cache, 48 kB L1 cache
- 96 kB shared memory per core
- DP @ 1/32 SP
- 192 GB/s peak bandwidth
- 4 TFLOP/s peak SP

Tesla P100 (GP100)

- Professional version
- Substantially improved over the GeForce (DP 1/2 SP, ECC)

NVIDIA Volta



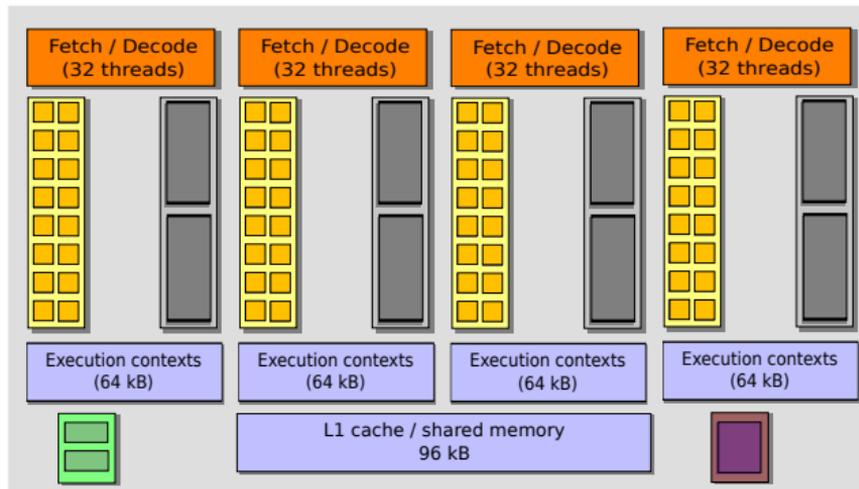
- Figure shows a single core of the GV100 design
- Green: DP units; gray: tensor cores
- Not shown: 4 SFUs, 32 LD/ST units

NVIDIA Volta

V100 GPU accelerator

- June 2017
- 84 SM (Streaming Multiprocessor) units
- 6 MB shared L2 cache
- DP @ 1/2 SP
- 900 GB/s peak bandwidth (HBM2)
- 8.1 TFLOP/s peak SP

NVIDIA Turing



- Figure shows a single core of the TU104 design
- Not shown: 4 SFUs, 16 LD/ST units
- New: raytracing core

NVIDIA Turing

GeForce RTX 2080

- Sep. 2018
- 48 SM (Streaming Multiprocessor) units
- 4 MB shared L2 cache
- DP @ 1/32 SP
- 448 GB/s peak bandwidth
- 8.9 TFLOP/s peak SP

More information

AnandTech

- Excellent and accessible description of the hardware
- Highly recommended
-
- <http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review>
- <http://www.anandtech.com/show/7764/the-nvidia-geforce-gtx-750-ti-and-gtx-750-review-maxwell>
- <http://www.anandtech.com/show/8923/nvidia-launches-geforce-gtx-960>
- <http://www.anandtech.com/show/10326/the-nvidia-geforce-gtx-1080-preview>
- <http://www.anandtech.com/show/11367/nvidia-volta-unveiled-gv100-gpu-and-tesla-v100-accelerator-announced>
- <https://www.anandtech.com/show/13282/nvidia-turing-architecture-deep-dive/4>