Programmable Graphics Hardware (GPU)

A Primer

Klaus Mueller

Stony Brook University

Computer Science Department

Parallel Computing Explained

video

Parallel Computing Explained

Any questions?

Parallelism

What you just saw was an embarrassingly parallel task

- no or very little communication among parallel tasks

Most computational problems are not like that

Parallelism

On the other hand, some processes are not parallel at all

- are they embarrassingly sequential?

Need to find and gauge where the parallelism is

Types of Parallelism

Task based parallelism

- unrelated processes are executed in parallel
- slowest process determines the speed
- also known as *coarse grained parallelism*
- MIMD model = Multiple Instructions Multiple Data

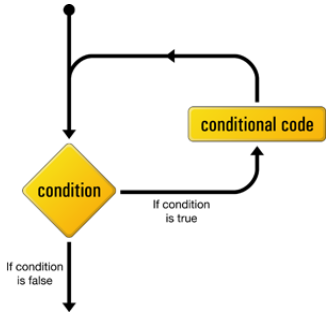
Data based parallelism

- decompose a specific task into *threads*
- each thread executes the same statement at the same time
- also known as *fine grained parallelism*
- SIMD model = Single Instructions Multiple Data

Patterns of Parallelism

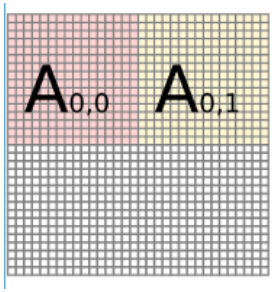
Loops

- *for* and *while* statements
- Fork and Join

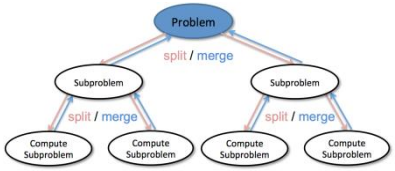


Tiling and grids

- break the domain into sub-problems that map well to the hardware
- 2D tiles/grid for images, 3D tiles/grid for volumes

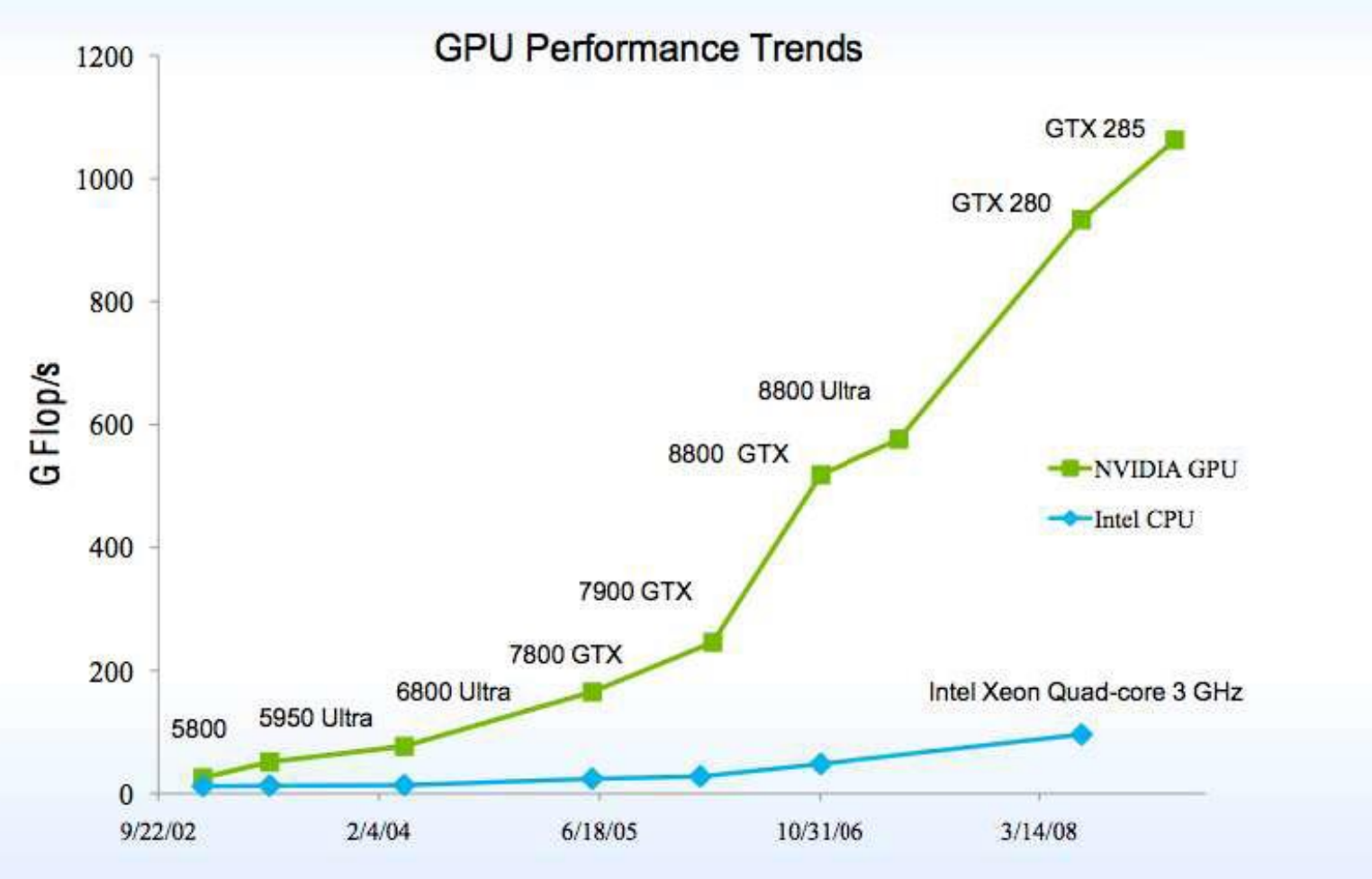


Divide and Conquer



- recursion: can present problems for parallelism when too deep
- better use an iterative approach that solves a level in parallel

Speedup Curves



Amdahl's Law

Governs theoretical speedup

$$S = \frac{1}{(1-P) + \frac{P}{S_{parallel}}} = \frac{1}{(1-P) + \frac{P}{N}}$$

P: parallelizable portion of the program

S: speedup

N: number of parallel processors

Amdahl's Law

Governs theoretical speedup

$$S = \frac{1}{(1-P) + \frac{P}{S_{parallel}}} = \frac{1}{(1-P) + \frac{P}{N}}$$

P: parallelizable portion of the program
S: speedup
N: number of parallel processors

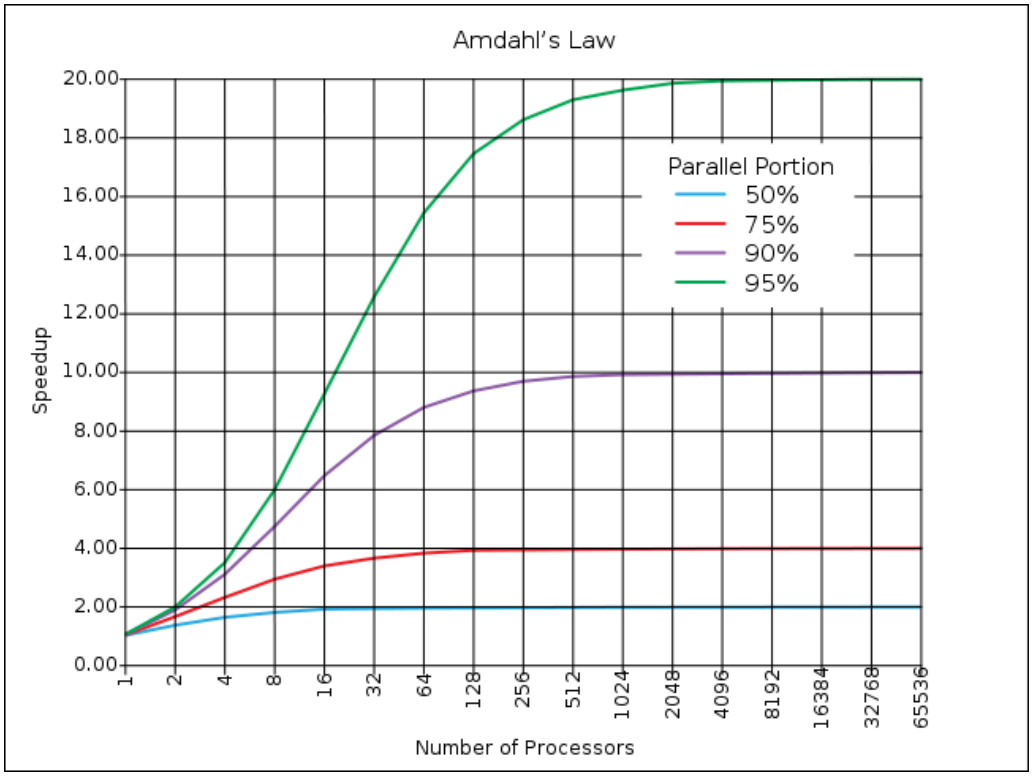
P determines theoretically achievable speedup

- example (assuming infinite N): P=90% → S=10
  P=99% → S=100

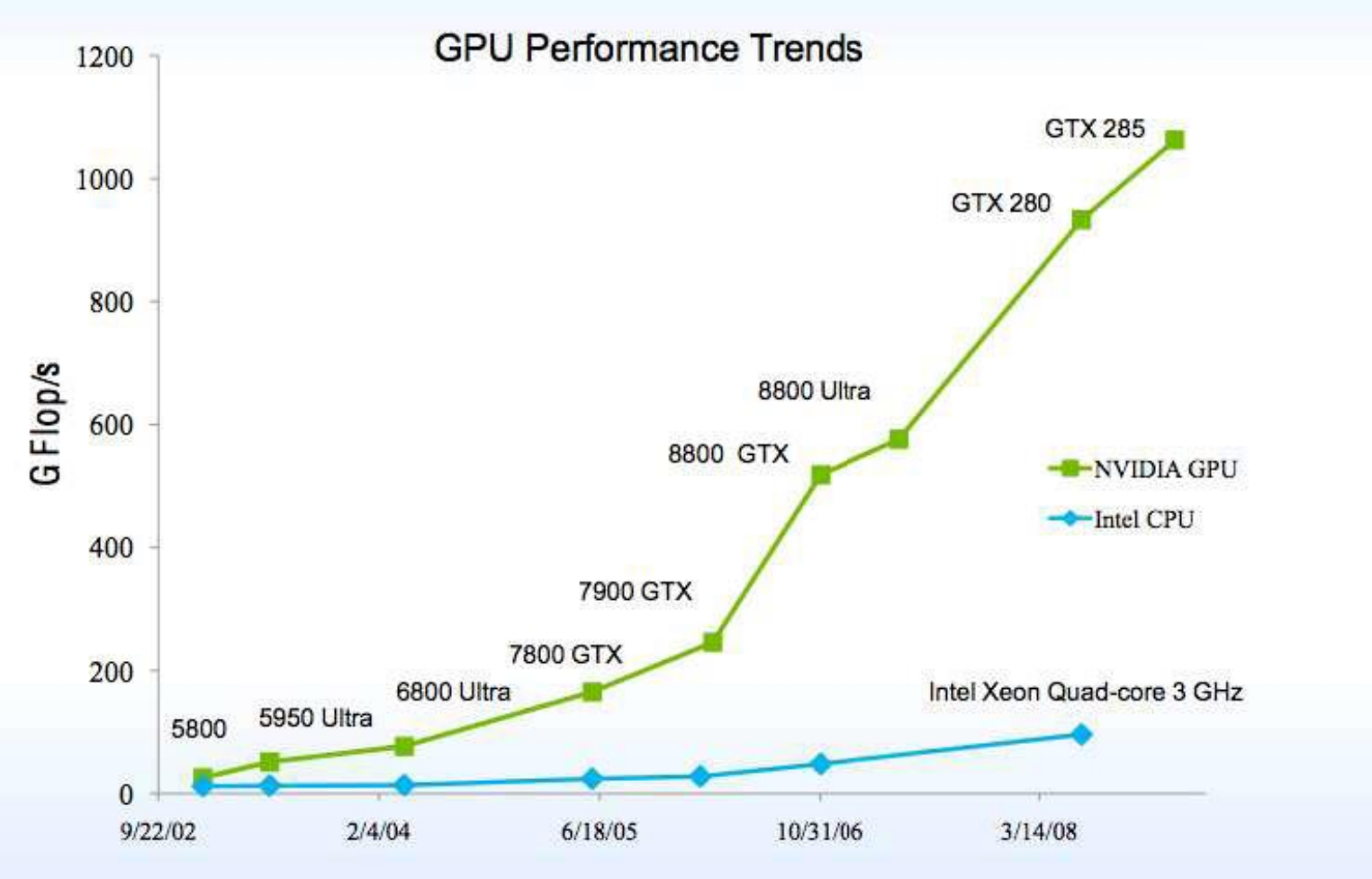
Amdahl's Law

How many processors to use

- when P is small → a small number of processors will do
- when P is large (embarrassingly parallel) → high N is useful



Speedup Curves

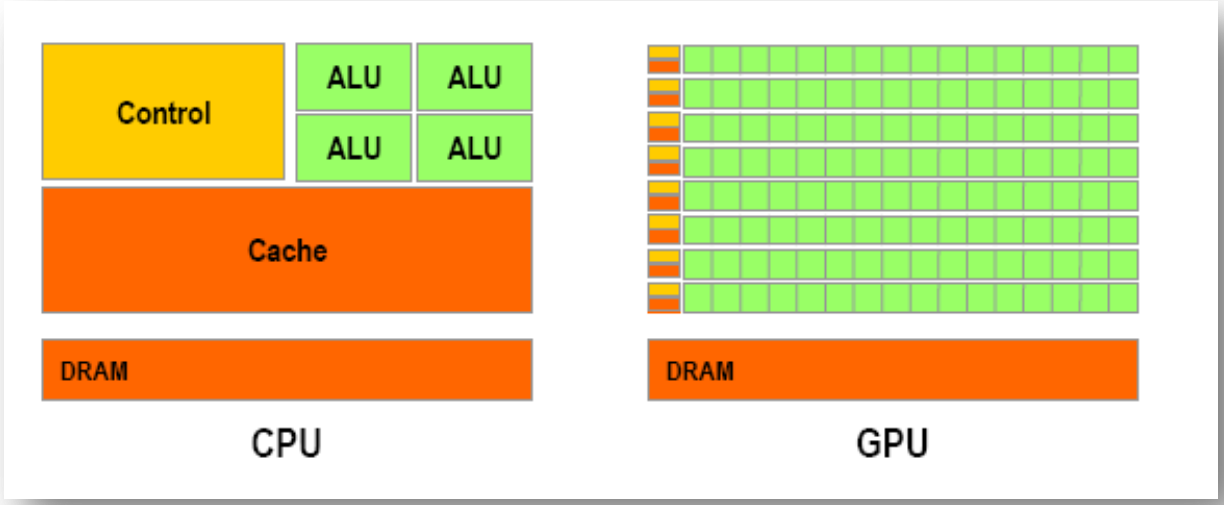


Beyond Theory….

GPUs are more than parallel computing

There are certain features that provide a turbo boost

- special ASIC circuits for frequent operations
- latency hiding by rapid thread switching
- special memory organization for 2D data
- schedulers
- managers
- APIs, drivers
- caches
- dedication to computing



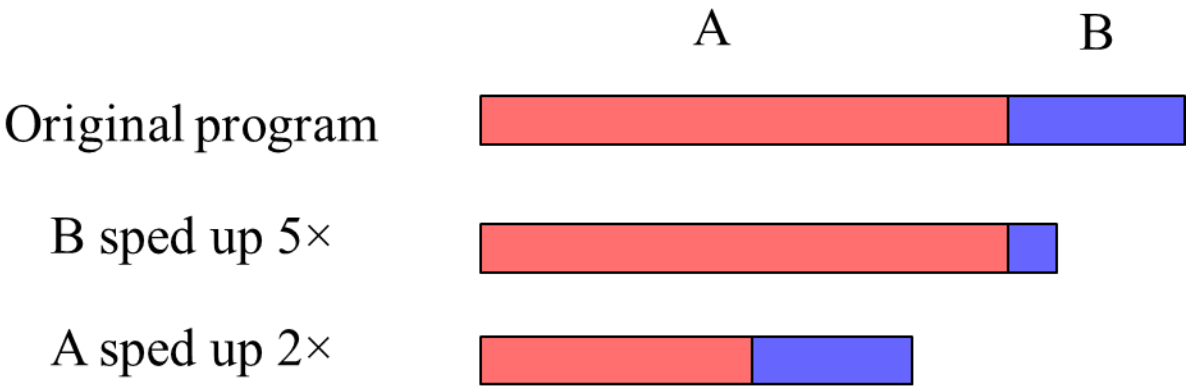
Focus Efforts on Most Beneficial

Optimize program portion with most ‘bang for the buck’

- look at each program component
- don’t be ambitious in the wrong place

Example:

- program with 2 independent parts: A, B (execution time shown)



- sometimes one gains more with less

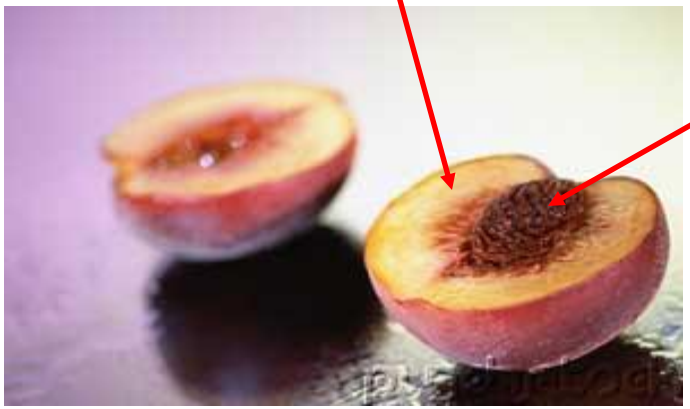
Programming Strategy

Use GPU to complement CPU execution

- recognize parallel program segments and only parallelize these
- leave the sequential (serial) portions on the CPU

parallel portions (enjoy)

sequential portions (do not bite)

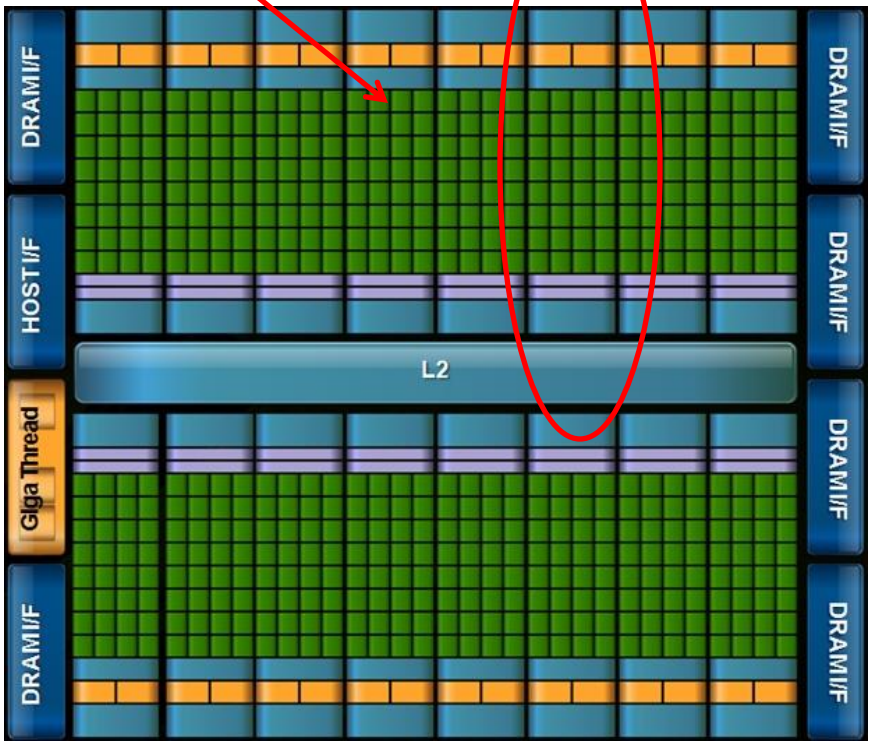


PPP (Peach of Parallel Programming – Kirk/Hwu)

The Hardware …. NVIDIA Fermi

CUDA Core

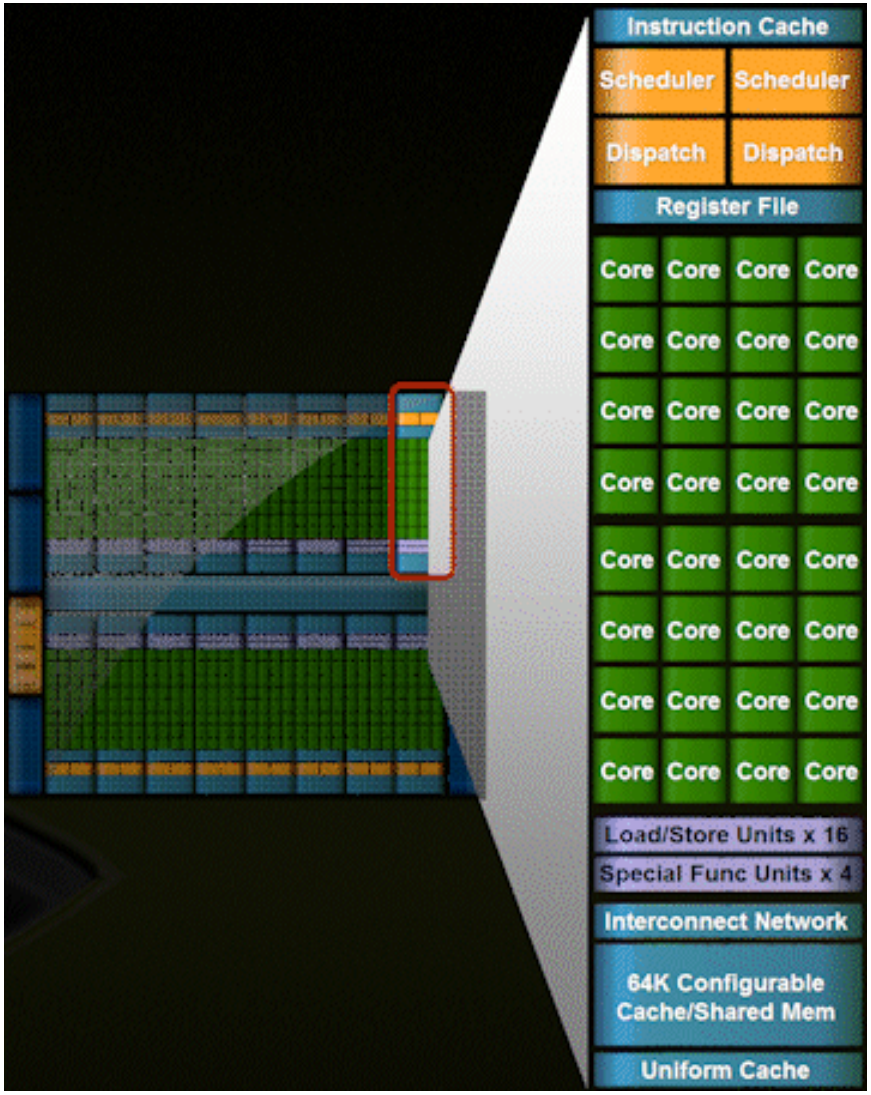
SM (Streaming Multiprocessor) → has 32 Streaming Processors (SP) = CUDA core



On chip:

- SMs: up 16
- CUDA cores: 32/SM → up to 512/chip

The Hardware …. NVIDIA Fermi



full cross-bar interface

32 CUDA Cores

4 special function units (sin, cosine, reciprocal, and square root)

Host and Device

Host → CPU

- controls program flow
- manages threads
- loads GPU programs (kernels)
- has host memory

Device → GPU

- loads data
- performs computations
- has device memory

Heterogeneous programming model

Parallelism Exposed as Threads

Thread management:

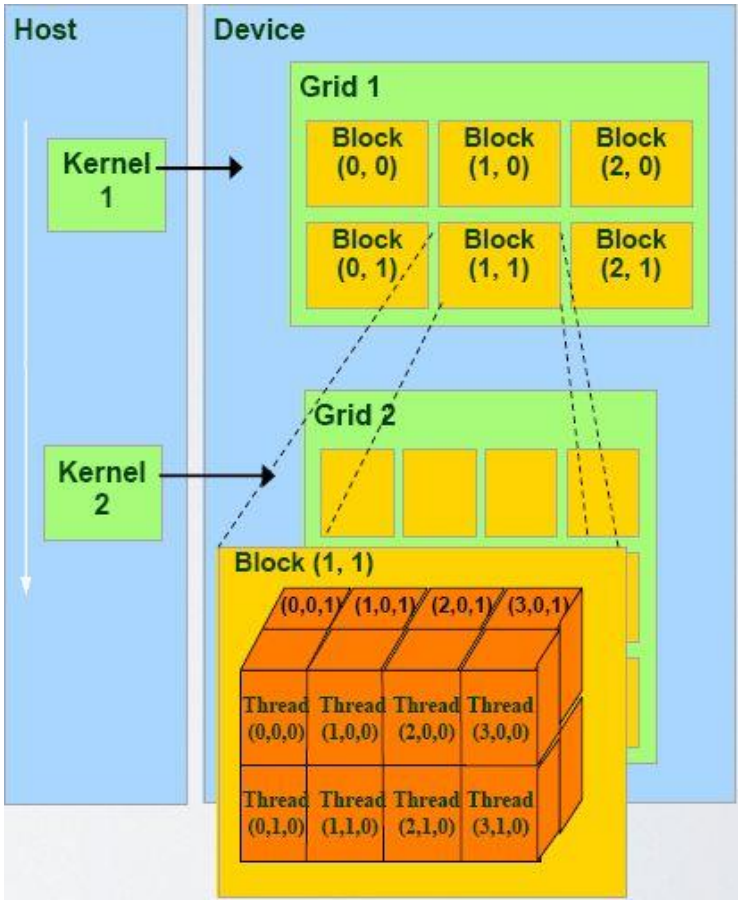
- all threads run the same code
- a thread runs on one core

The threads divide into *blocks*

- each block has a unique ID → *block ID*
- each thread has a unique ID within a block → *thread ID*
- block ID and thread ID can be used to compute a *global ID*

The blocks form a *grid*

Block/grid size can be set in program



An Important Player: Memory

CUDA threads may access data from multiple memory spaces:

Thread-level

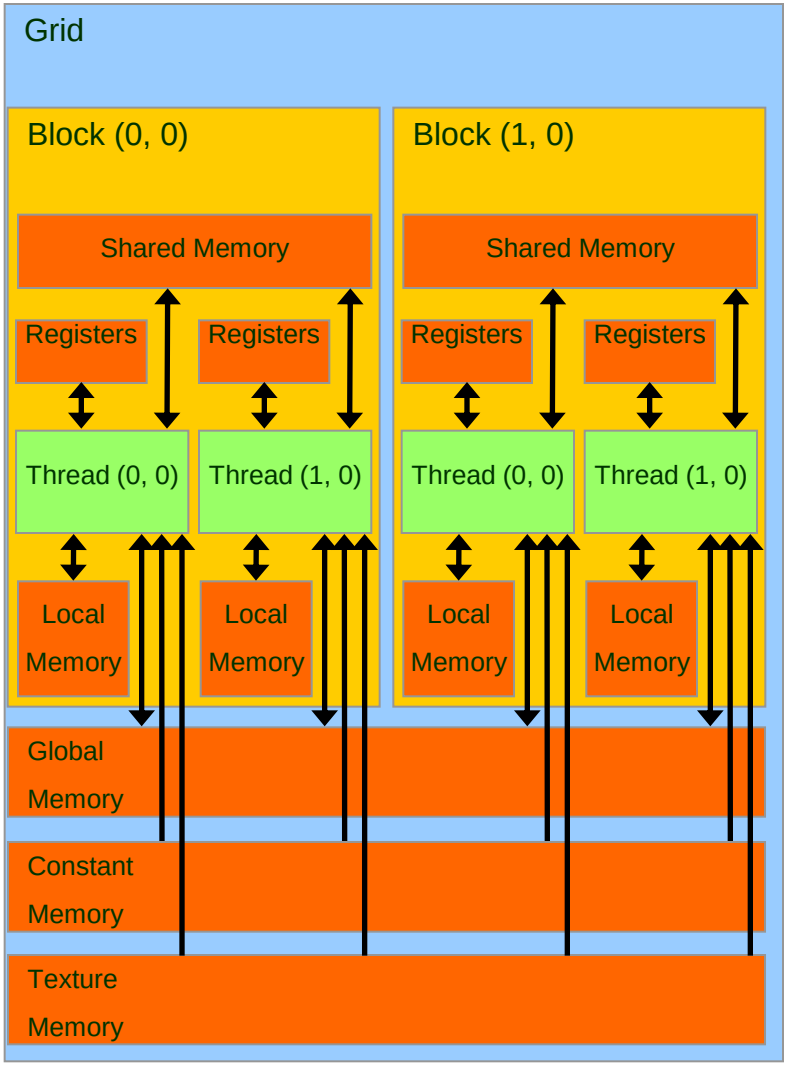
- registers (fast)
- local memory to handle register spills (slow)

Block-level

- shared memory

Grid-level

- global memory (slowest)
- constant memory (read-only)
- texture memory (cached, read-only)
- surface memory (writable texture)



Latency Hiding -- Revisited

Latency hiding is a form of hardware multi-threading

Major source of the speedup of GPUs

- a new set of threads (called *warps* = 32 threads) is switched to within one clock cycle whenever one of the threads in the currently active set stalls

But….hardware multi-threading requires memory

- contexts of all these threads must be maintained in memory
- this typically limits the amount of threads that can be simultaneously maintained for latency hiding
- so there is a tradeoff

Avoid Latency – Exploit Locality

Temporal locality

- data that was accessed before will be likely accessed again
- use cache to reduce access latencies

Spatial locality

- data close to the data accessed last will likely be accessed soon
- fetch entire cache lines when accessing one element

Exploit locality by

- storing data in shared memory
- configure hardware caches (L2, CUDA vs. self-managed shared memory)
- e.g., split 64 KB/block into 48 KB CUDA cache and 16 KB self-managed (Fermi and higher)

Next – Small Example

Programmed in CUDA

CUDA = Compute Unified Device Architecture

- C-like language
- language and API created by NVIDIA
- libraries available (cuBLAS, cuFFT, Thrust, …)

Vector Add – CPU

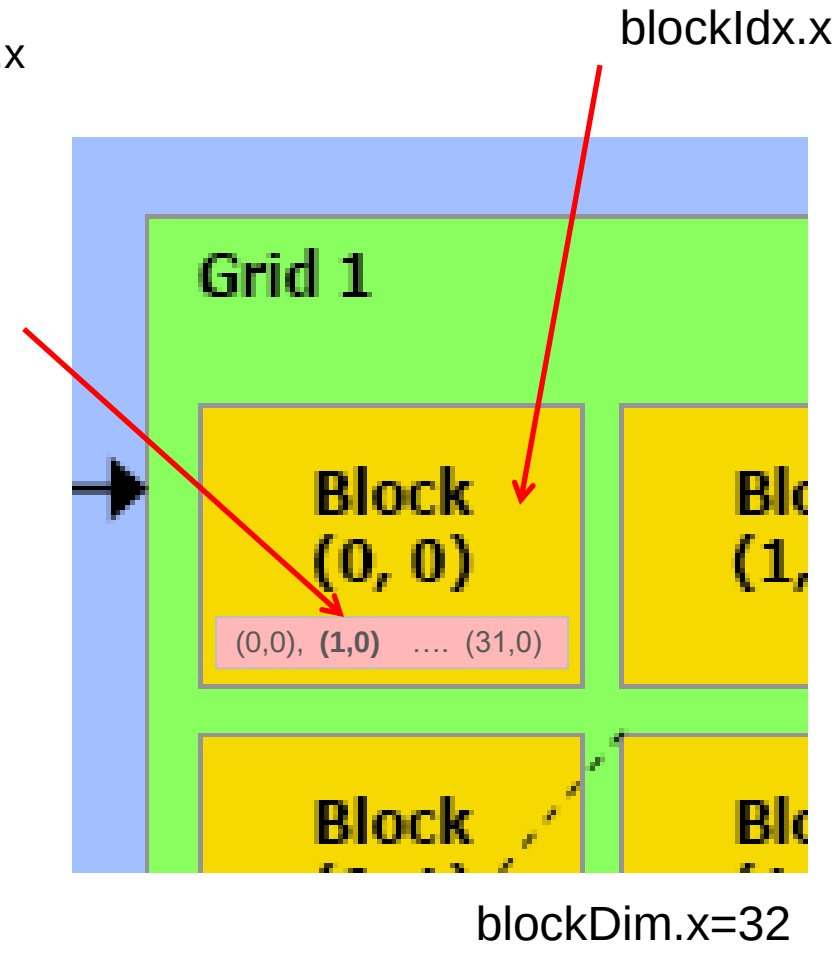
```
void vectorAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i]; }


int main() {
    int N = 4096;
        // allocate and initialize memory
    float *A = (float *) malloc(sizeof(float)*N);
    float *B = (float *) malloc(sizeof(float)*N);
    float *C = (float *) malloc(sizeof(float)*N);
    init(A); init(B);

    vectorAdd(A, B, C, N);          // run kernel
    free(A); free(B); free(C);}     // free memory
```

Vector Add – GPU (kernel program)

```
__global__ void gpuVecAdd(float *A, float *B, float *C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x
    C[tid] = A[tid] + B[tid]; }
```



tid = blockId.x * blockDim.x + threadIdx.x

Vector Add – GPU (host program)

```
int main() {
    int N = 4096;            // allocate and initialize memory on the CPU
    float *A = (float *) malloc(sizeof(float)*N);
            float *B = (float *) malloc(sizeof(float)*N); *C = (float*)malloc(sizeof(float)*N)
    init(A); init(B);
            // allocate and initialize memory on the GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, sizeof(float)*N);
    cudaMalloc(&d_B, sizeof(float)*N);     cudaMalloc(&d_C, sizeof(float)*N);
    cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
    cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
            // configure threads
    dim3 dimBlock(32,1);
    dim3 dimGrid(N/32,1);
            // run kernel on GPU
    gpuVecAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
            // copy result back to CPU
    cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);
            // free memory on CPU and GPU
    cudaFree(d_A);   cudaFree(d_B);    cudaFree(d_C);   free(A);   free(B);   free(C); }
```

Common Optimizations

Loop unrolling

- reduces arithmetic and creates better vectorization

Loop fusion

- but check for dependencies

Thread fusion

- increases workload for threads

Kernel fusion

- encourages data reuse

Collaborative load into shared memory

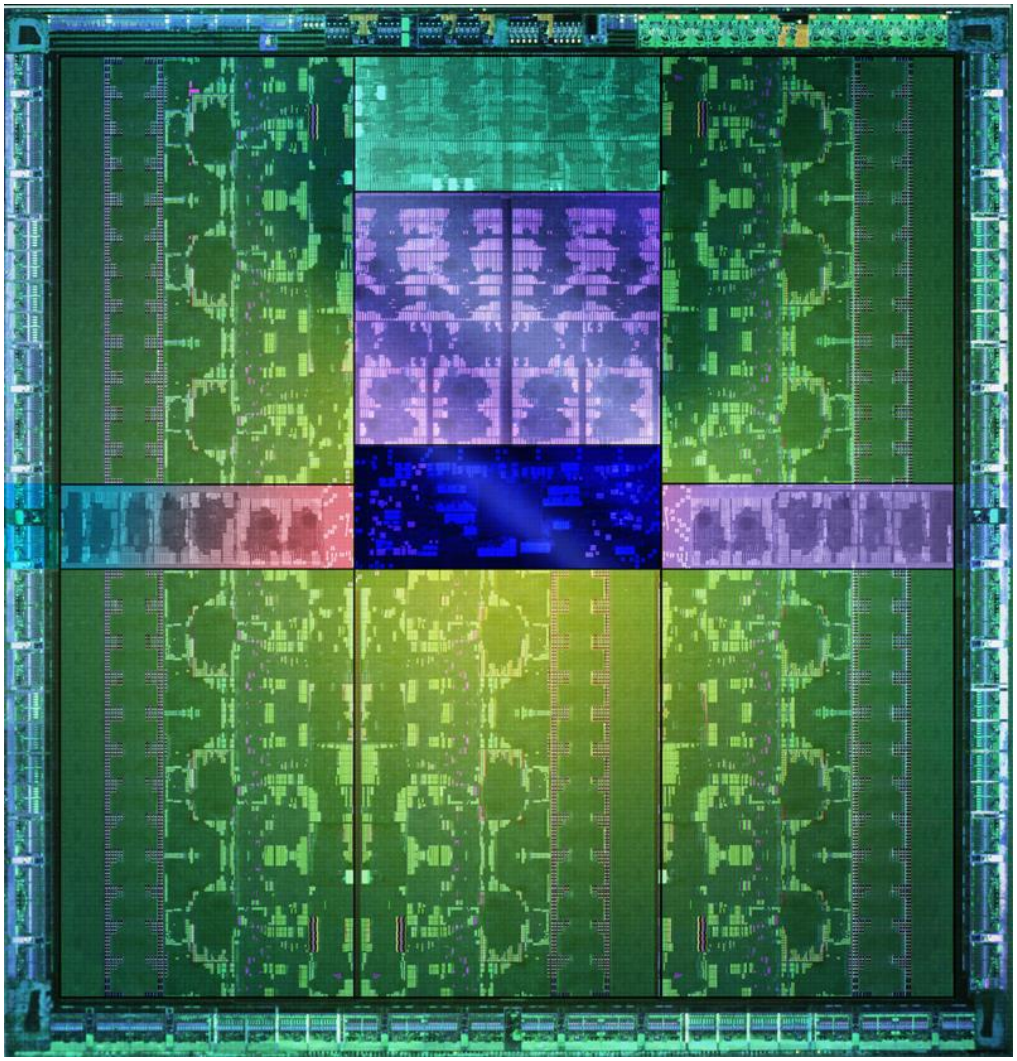
- when memory indexing is irregular

Larger blocks

- more threads can better hide memory latency
- but more threads require more registers → trade-off

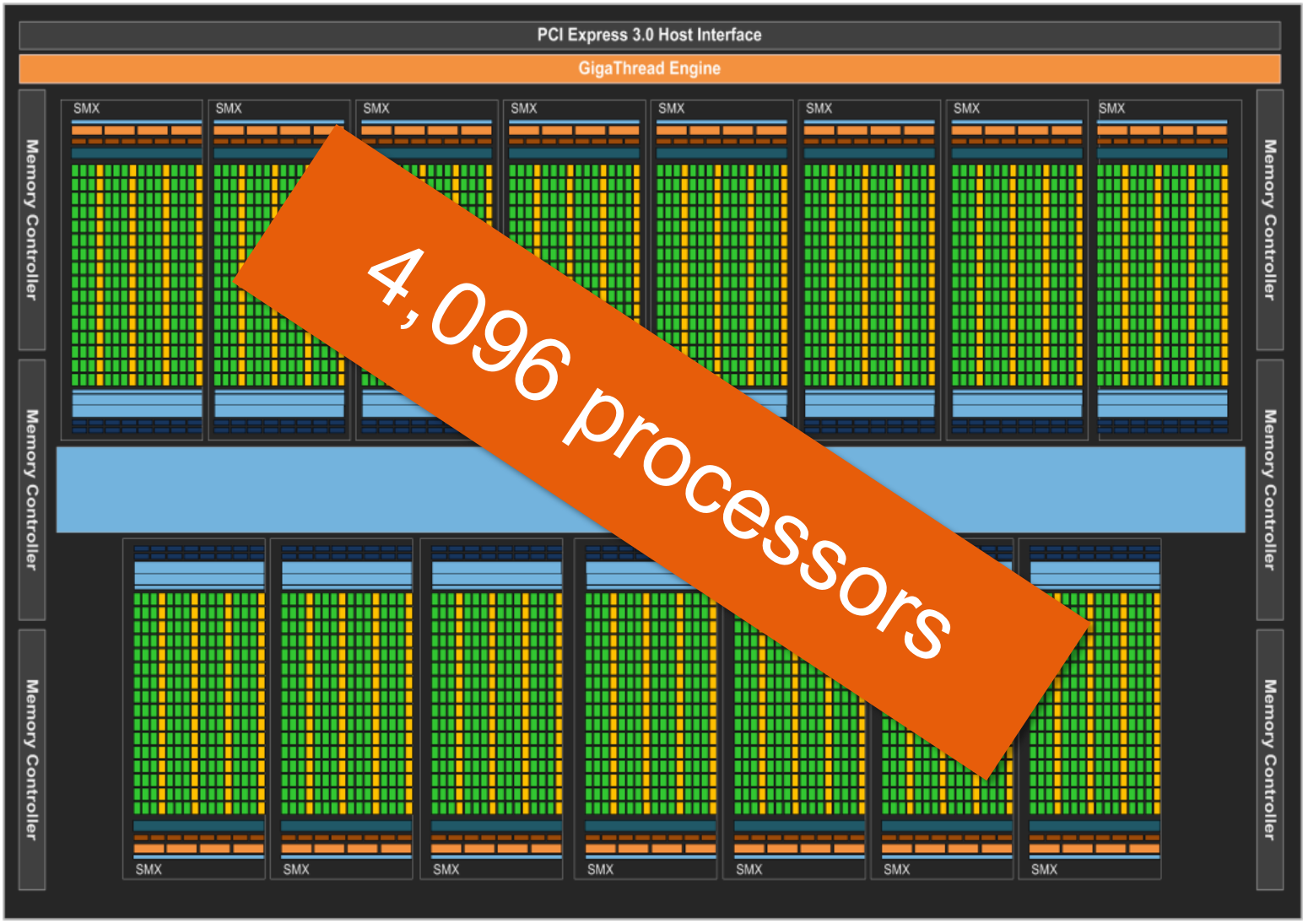
but beware of hardware intelligence
some might already be done for you

NVIDIA Kepler Architecture



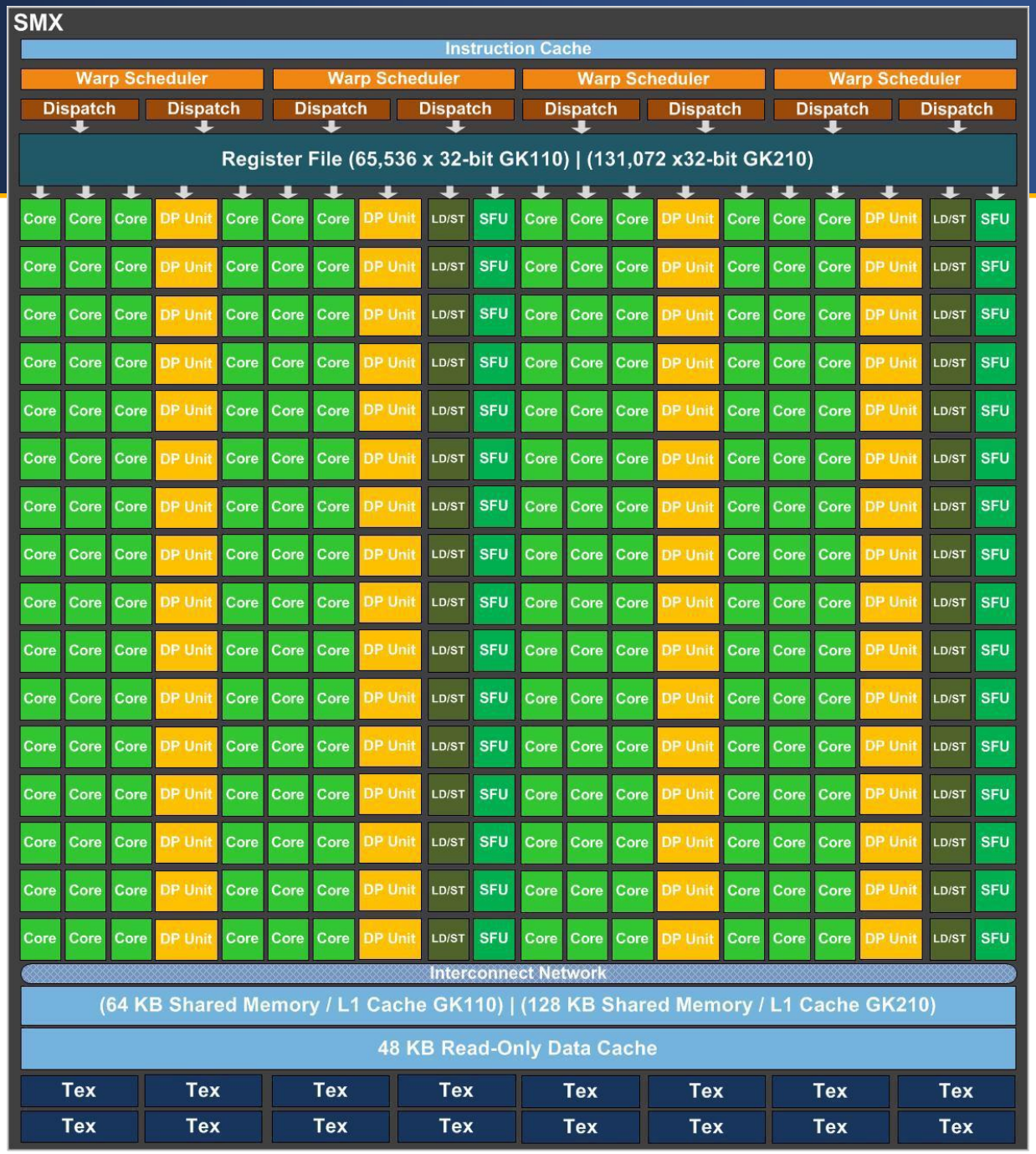
Kepler GK110 Die Photo

16 Streaming Multiprocessors (SMX)



One SMX

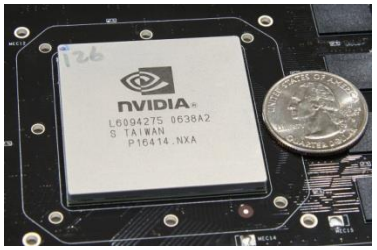
- 192 single-precision CUDA cores
- 64 double-precision units
- 32 special function units (SFU)
- 32 load/store units (LD/ST)



High Performance Computing on the Desktop

PC graphics boards featuring GPUs:

- NVIDIA GeForce, ATI Radeon
- available at every computer store for less than $500
- set up your PC in less than an hour and play



the latest board:

NVIDIA GeForce GTX 980

“Just” Computing

Compute-only (no graphics): NVIDIA Tesla K and M series

K 80

True GPGPU

(General Purpose Computing using GPU Technology)

24 GB memory per card, 560 processors

$4,000

Bundle 8 cards into a server: 5,280 processors, 192 GB memory

Relevance to Visualization

Volume rendering is a compute-intensive tasks

- it has also a high degree of parallelism
- for example, we can cast the rays in parallel

There are also many compute intensive tasks in data science

- the challenge is to find the parallelism
- for example, each iteration of k-means can be run in parallel
- but the iterations themselves are sequential.