General Purpose GPU computing – using CUDA

Axel Arnold

Institute for Computational Physics
Universität Stuttgart

October 21, 2010

Why?

CPU $\approx$ 500$, $\approx$ 100 W

- 4 cores executing 4 FP instruction @ 3 GHz
- 3 DDR channels of width 8 byte, clock 668 Mhz

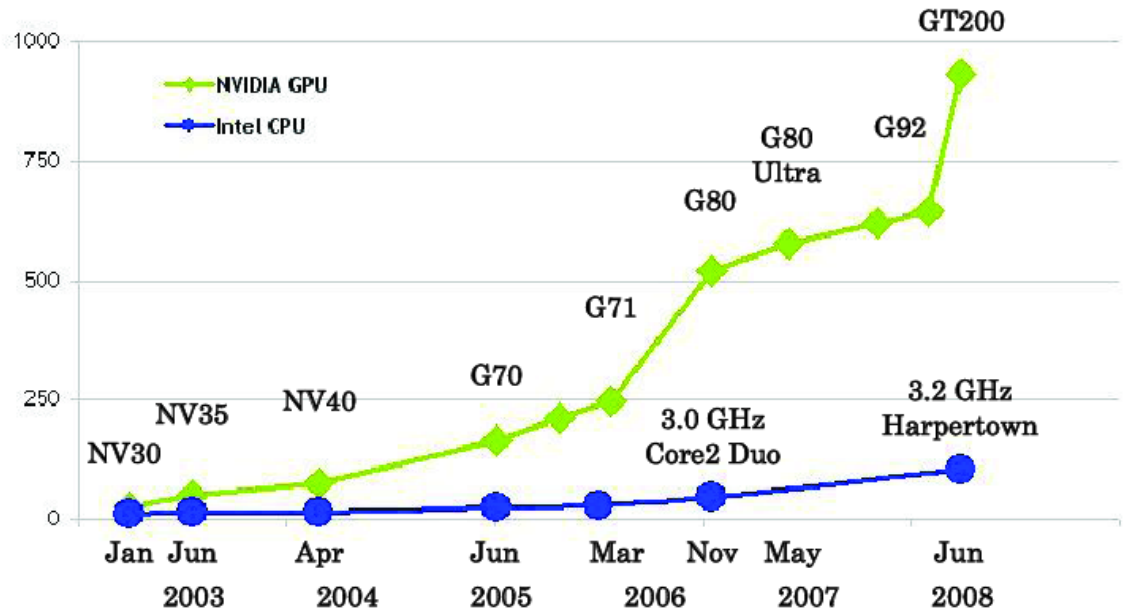
$\approx$ 50 Gigaflops and $\approx$ 32 GB/s data throughput

GPU $\approx$ 500$, $\approx$ 200 W

- 15 multiprocessors executing 128 FP instructions @ 1.4 GHz
- DDR memory bus of total width 48 bytes, clock 1.8 GHz

$\approx$ 1.3 Gigaflops and $\approx$ 170 GB/s data throughput

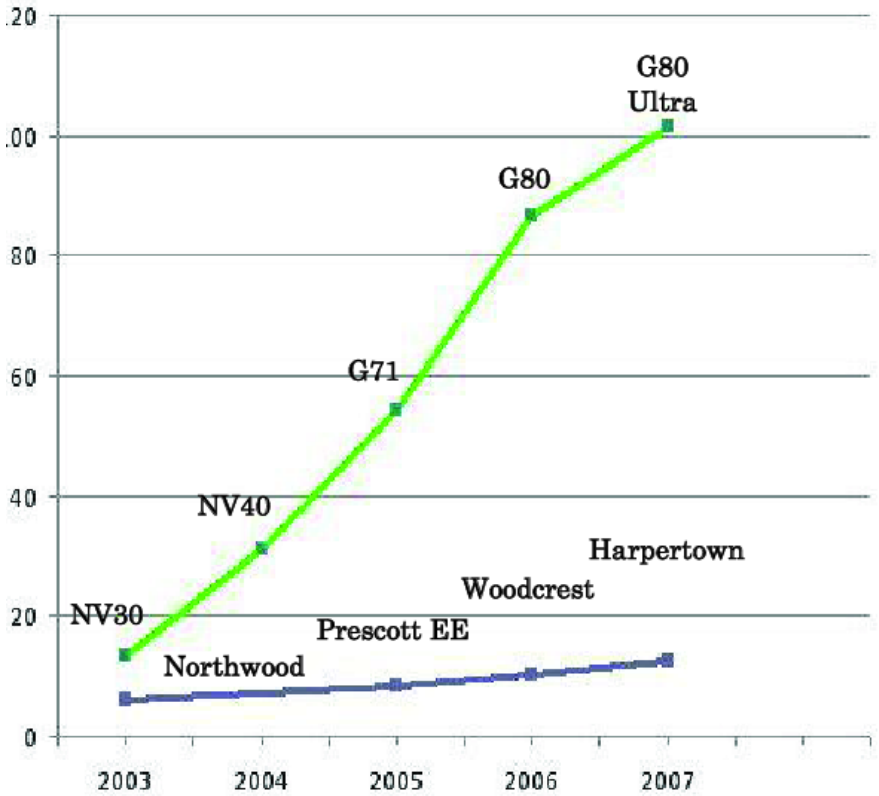
And in the future?



NVIDIA, 2010

CPU flops double every 18 months,
GPU flops every 9 months

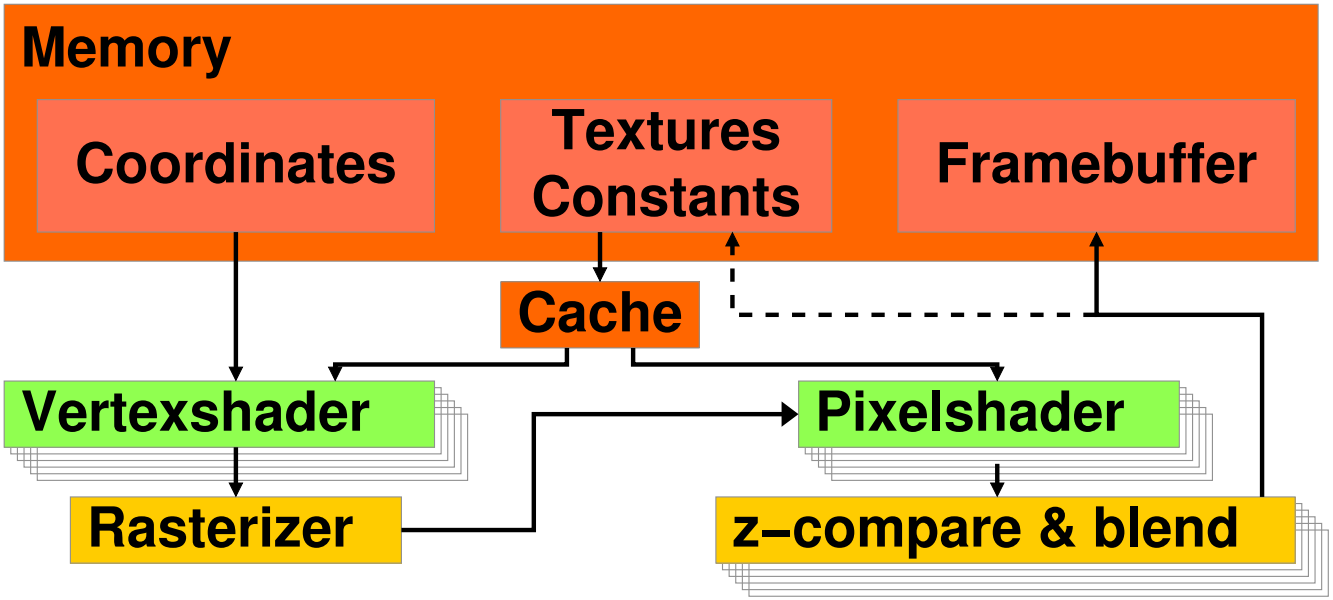
And in the future?



NVIDIA, 2010

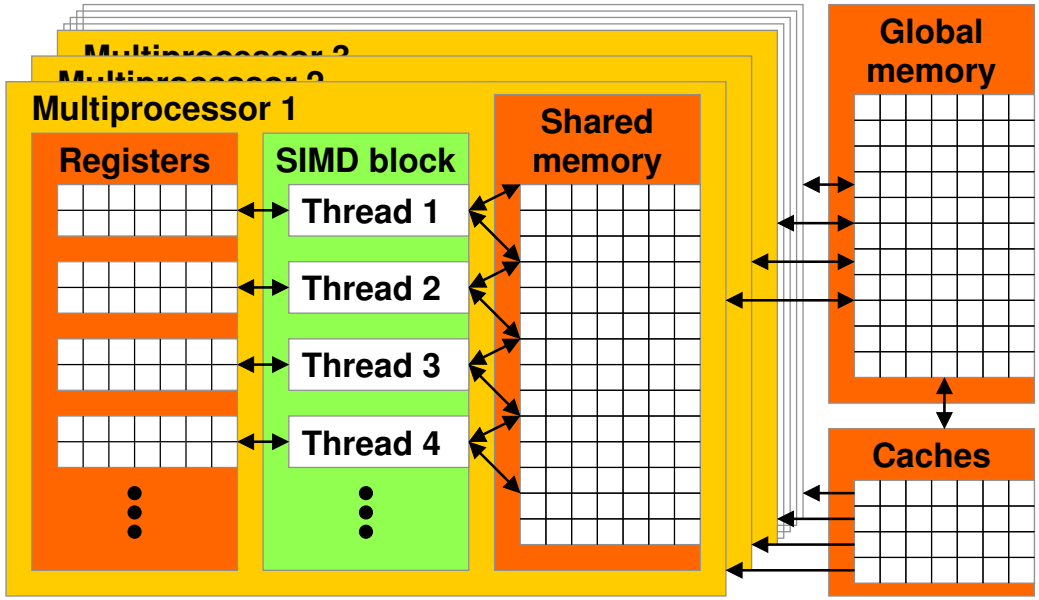
memory bandwidth factor constant

OpenGL



	GLSL	output
vertex shader	✓	coordinate transformations (placing of objects, camera position)
rasterizer		visible pixels of object
pixel shader	✓	color from texture & lighting
z-compare and blend		framebuffer pixels and z-depths (if visible)

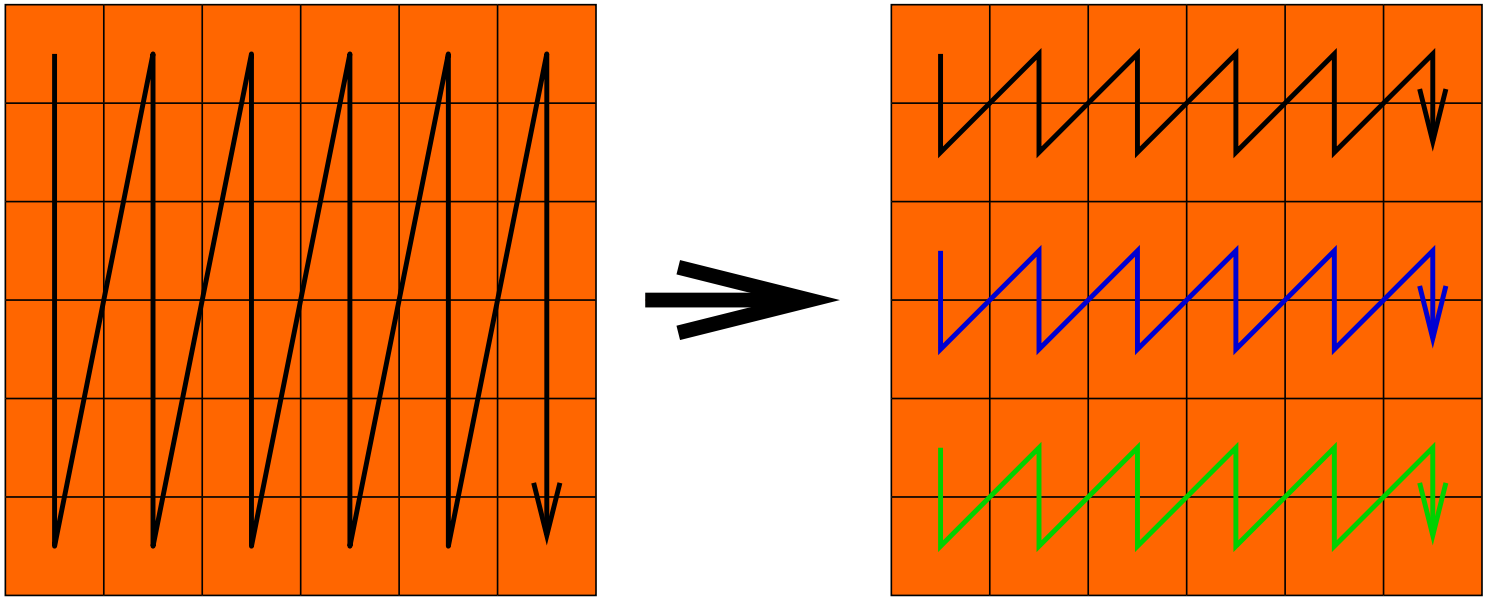
NVIDIA GeForce 8800 inside



- vertex and pixel shaders run on unified programmable hardware
- up to 16 multiprocessors, each containing 8 FP units
- *one* instruction per 4 half cycles $\longrightarrow$ executes the *same instruction* on 32 data elements at once

512 instructions at once $\Longrightarrow$ parallel programming

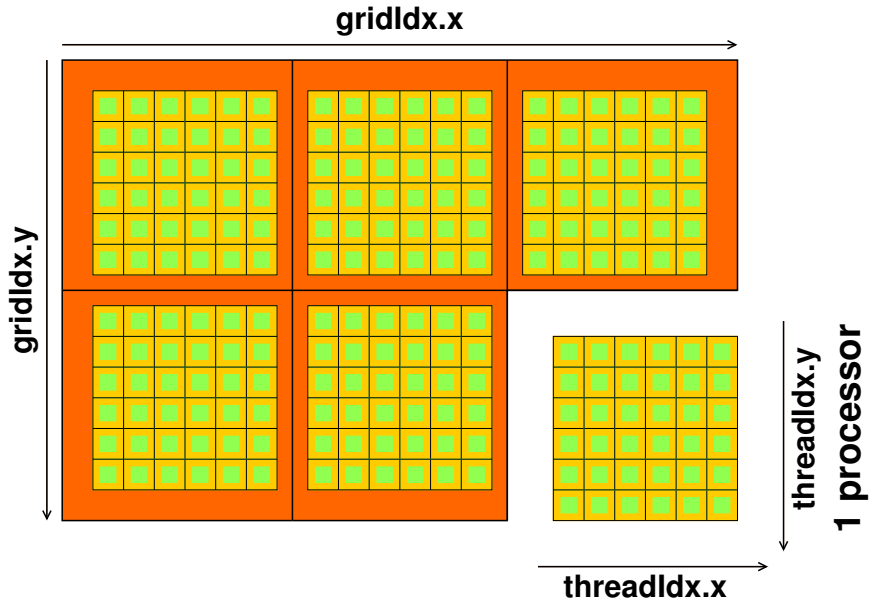
Multi–threading



- divide the work into smaller units
- run one program many times at once
- each instance has its own registers and stack
- everything else is shared
- read/write memory concurrently

→ synchronization necessary

GPU execution model

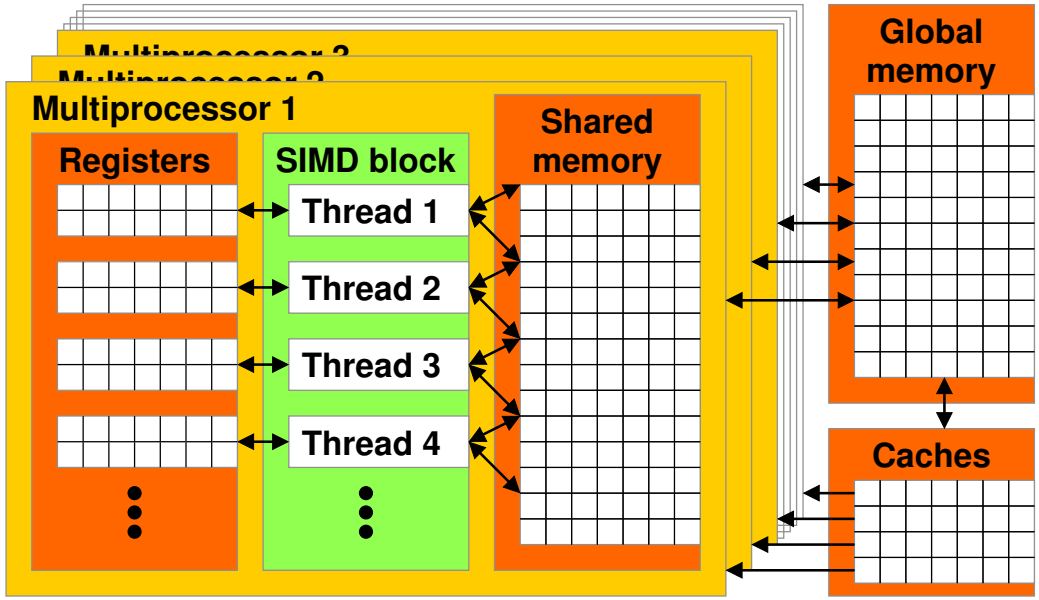


- organization on the GPU in three classes:

thread	a single instance of a program
block	a set of up to 1024 threads on the same MP
grid	a set of up to 65535^3 blocks on the same GPU

- typically more threads than FP units
- ... and more blocks than multiprocessors

Memory



- 128kB register space per MP (2.8 TB/s)
  private per thread, limits number of threads
- up to 6GB unsynchronized global memory (170 GB/s)
- 16/48kB shared memory on each MP (2.8 GB/s)
- 48/16kB L1 cache
- 16kB 1D and 2D read only texture cache

GPU vs. CPU architecture

CPU

- MIMD
- # threads $\leq$ # cores
- $\leq$ 8 threads per CPU at once
- cores on one motherboard: OpenMP parallelization
- between boards: MPI parallelization
- several layers of caches hide memory latency

GPU

- 32-way SIMD
- # threads $\gg$ # cores
- $>$ 1000 threads at once
- cores in a multiprocessor: shared memory
- between multiprocessors: global memory
- $\geq$ 192 threads per block hide memory latency
- $\geq$ 16 blocks to load all MPs

NVIDIA Compute Unified Device Architecture

- C/C++-like language for kernels (GPU programs)
- C/C++ for host CPU programming
- one file can contain host and GPU code
- similar to OpenMP on conventional hardware
  - automatic computation domain decomposition
  - thread number can be chosen independently of hardware
- CUDA-emulator
  - for systems without CUDA-enabled GPU
  - debugger
  - `printf`-debugging

Example code — $c = A \times b$, GPU part

```
__global__
void Atimesb_g(float *A, float *b, float *c,
               int A_stride, int b_blocks) {
  __shared__ float lb[BLOCK_SIZE];
  float my_elem = 0;
  for (unsigned int r = 0; r < b_blocks; ++r) {
    unsigned int offset = BLOCK_SIZE*r;
    // load b for reuse by all threads
    lb[threadIdx.x] = b[threadIdx.x + offset];
    __syncthreads();
    // calculate elements using broadcast
    for (int e = 0; e < BLOCK_SIZE; ++e)
      my_elem += A[A_stride*(e + offset) +
        threadIdx.x + blockDim.x*blockIdx.x]*lb[e];
    // directly write back
    c[threadIdx.x + blockDim.x*blockIdx.x] = my_elem;
}
```

Example code — CPU part

```
void Atimesb(int argc, char **argv,
             float *A, float *b, int sb, float *c, int sc) {
  CUT_DEVICE_INIT(argc, argv);

  float *g_b, *g_c;
  cudaMalloc((void**) &g_b, sb*sizeof(float));
  cudaMalloc((void**) &g_c, sc*sizeof(float));
  cudaMemcpy(g_b, b, sb*sizeof(float), cudaMemcpyHostToDevice);

  dim3 grid(BLOCK_SIZE, 1, 1);
  dim3 threads(sc/BLOCK_SIZE, 1, 1);
  Atimesb_g<<<grid, threads, 0>>>(g_A, g_b, g_c,
    s_c, s_b/BLOCK_SIZE);

  cudaMemcpy(c, g_c, sc*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(g_b); cudaFree(g_c);
}
```

In detail: shared memory

- exchange data between all threads in a block
- organized in 16 independent banks
- adjacent 32-bit words belong to adjacent banks:

bank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47

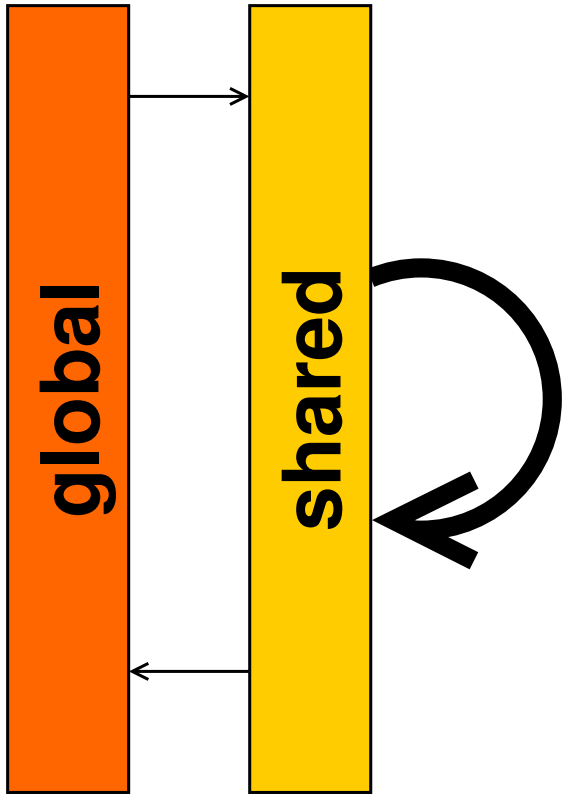
- all threads can read/write simultaneously
- reading/writing to the same bank is a *conflict*

n-*way conflict*: $n + 1$ threads access the same bank

- n-way conflict costs n extra cycles $\Longrightarrow$ avoid!
- exception: one broadcast word per cycle
- threads can synchronize shared memory accesses

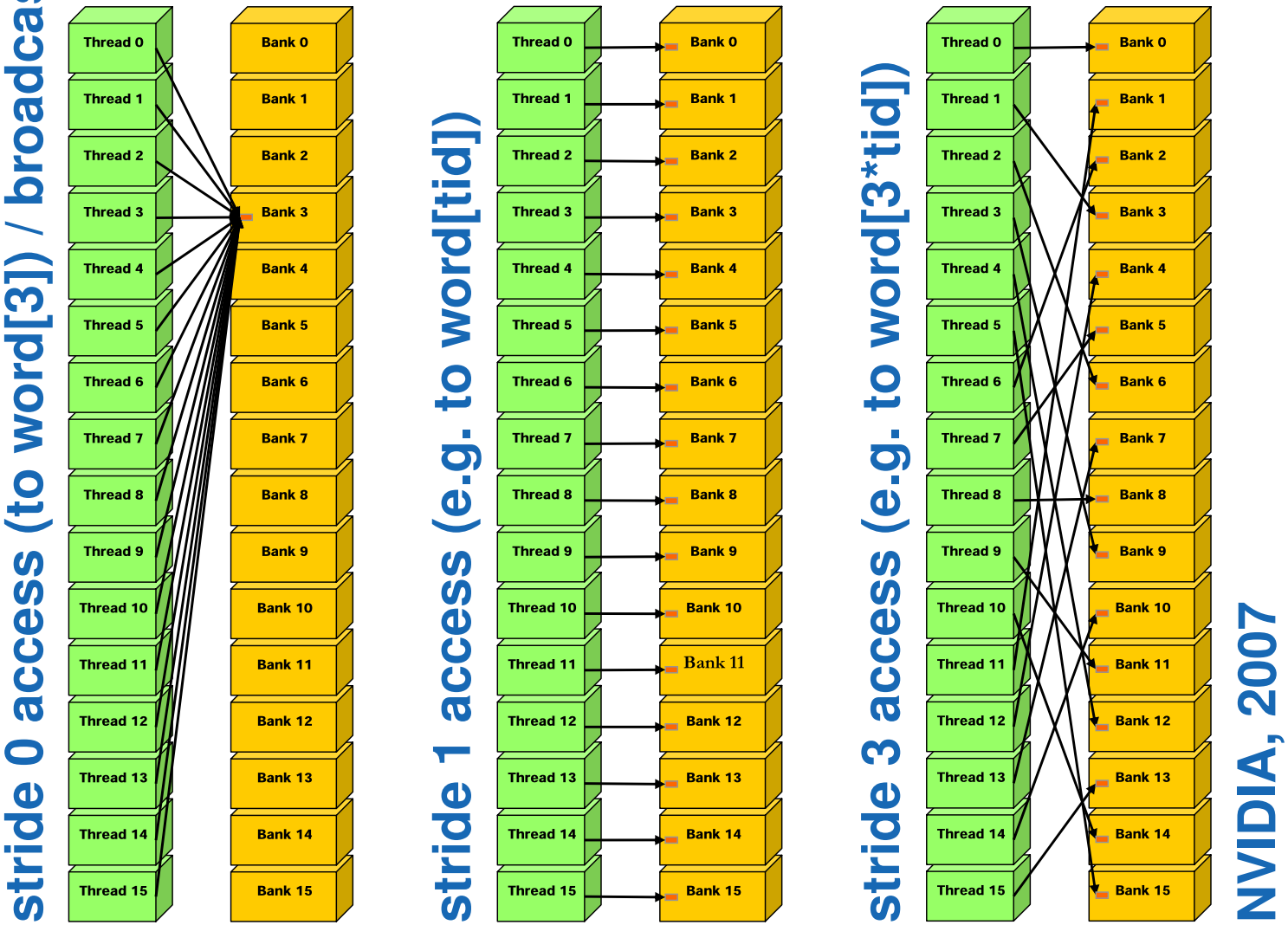
Shared memory II

1. load data from global to shared memory
2. synchronize (all finish load)
3. calculate in shared memory
4. synchronize (all threads finish computing)
5. write back data to global memory



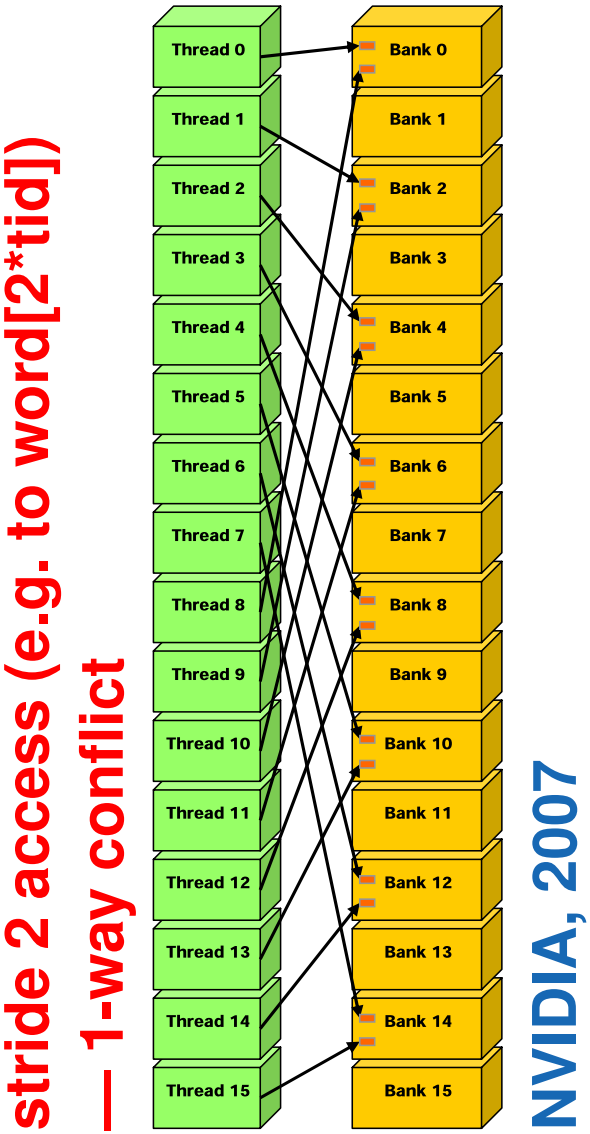
- use shared memory to communicate between threads
- try to minimize global memory accesses by using shared memory
- ... and to reuse shared memory as often as possible

(Avoiding) bank conflicts



NVIDIA, 2007

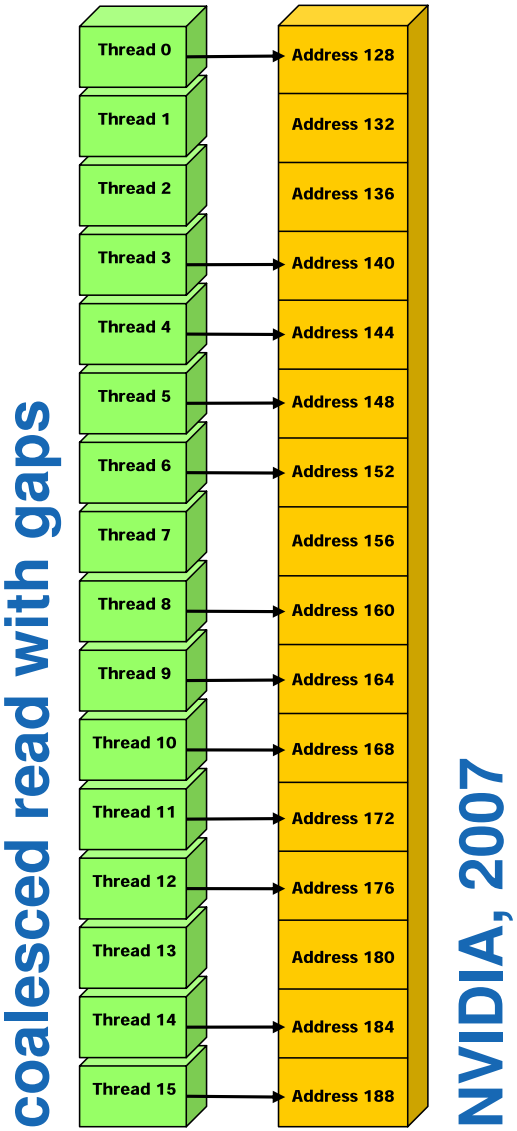
(Avoiding) bank conflicts II



- conflicts arise if two threads access words i, j simultaneously, where $i - j \mod 16 = 0$
- accessing words $i \times s$ (strided access) is conflict-free, if GCD$(s, 16) = 1$, i.e. s is odd
- conflict-free is any permutation of 16 adjacent elements

1/2-way conflicts do not justify additional computation

Global memory — coalescing



- global memory has stricter performance requirements
- if not obeyed, 10GB/s instead of 100GB/s!

Coalescence

- access to 4, 8 or 16 byte values
- all values for a half-warp (16 threads) within a 128 byte segment
- not all threads need to read their value
- prior to G90:
  - base address must be 128-aligned
  - thread *i* accesses value at base+*i*

Further benefits/drawbacks of CUDA

GPU texture fetching

- accurate nearest point sampling
- linear/bilinear interpolation

→ fast and accurate table lookups

CUDA buffers in OpenGL and vice versa

- fast visualization

download of data from the GPU

- up/downstream about 4 GB/s