GPGPU – A Supercomputer on Your Desktop

GPUs

Graphics Processing Units (GPUs) were originally designed to accelerate graphics tasks like image rendering.

- They became very very popular with videogamers, because they've produced better and better images, and lightning fast.
- And, prices have been extremely good, ranging from three figures at the low end to four figures at the high end.

GPUs are Popular

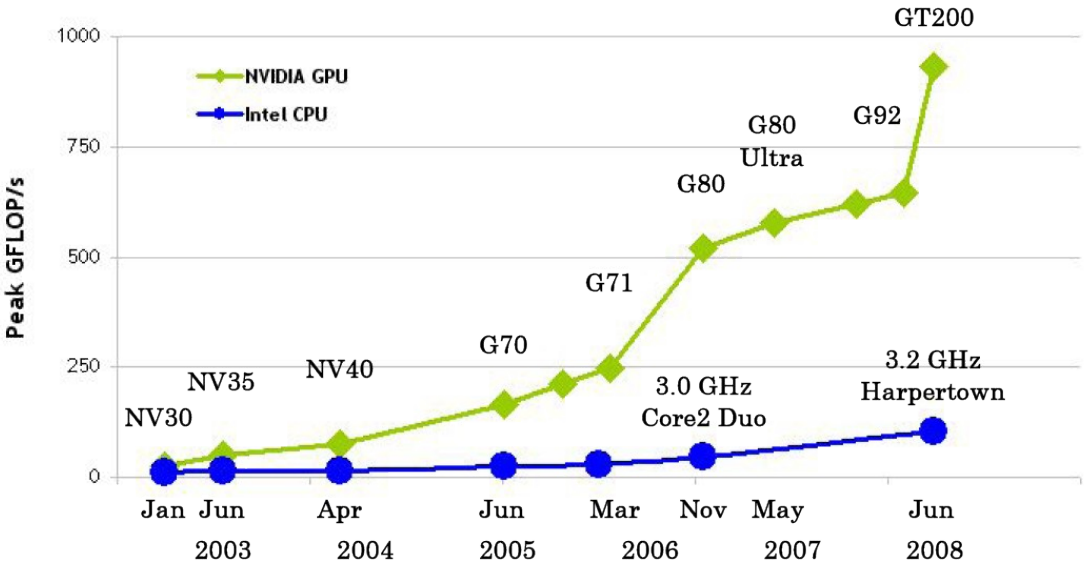
- Chips are expensive to design (hundreds of millions of $$$), expensive to build the factory for (billions of $$$), but cheap to produce.
- In 2006 – 2007, GPUs sold at a rate of about 80 million cards per year, generating about $20 billion per year in revenue.
  - http://www.xbitlabs.com/news/video/display/20080404234228_Shipments_of_Discrete_Graphics_Cards_on_the_Rise_but_Prices_Down_Jon_Peddie_Research.html
- This means that the GPU companies have been able to recoup the huge fix costs.

GPU Do Arithmetic

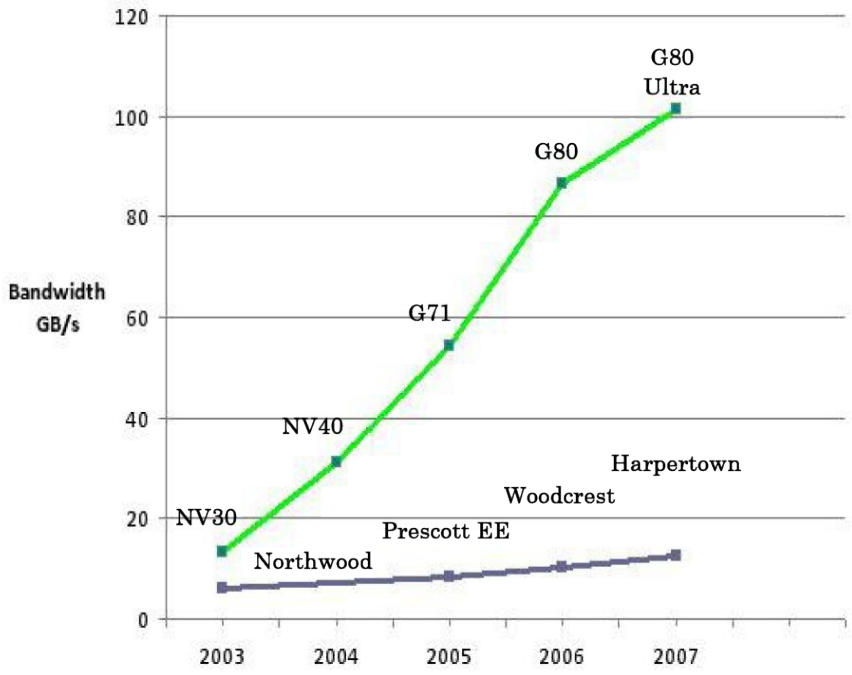
- GPUs mostly do stuff like rendering images.
- This is done through mostly floating point arithmetic – the same stuff people use supercomputing for!

GPU vs. CPU

- A quiet revolution and potential build-up
  - Calculation: 900 GFLOPS vs. 100 GFLOPS
  - Memory Bandwidth: 100 GB/s vs. 12 GB/s

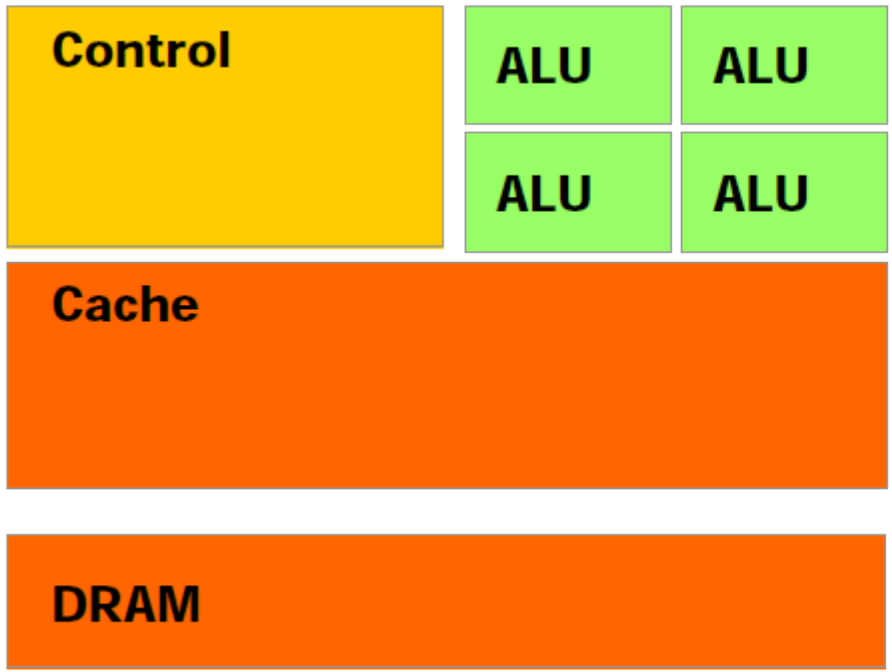


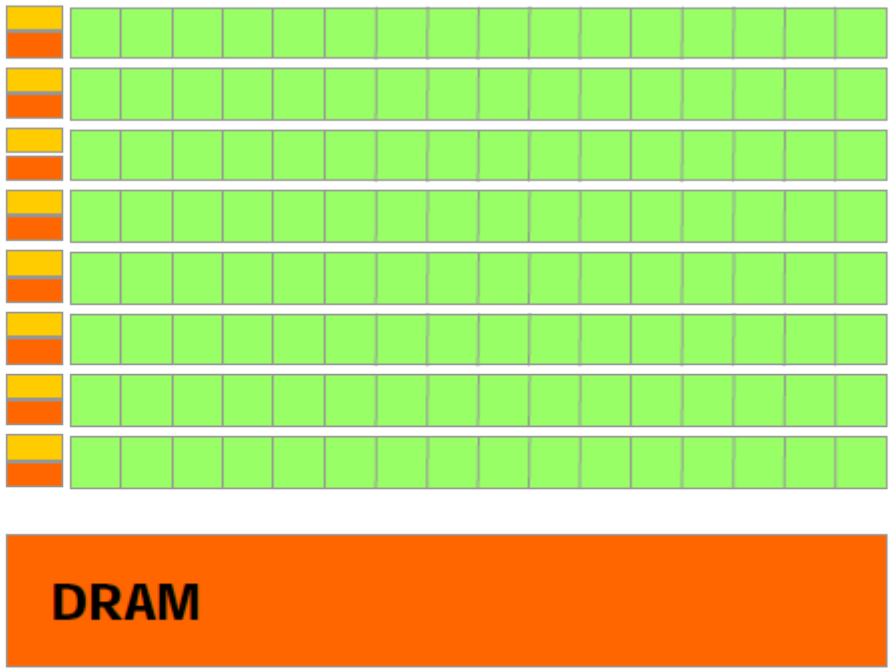
GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	



CPU vs. GPU

- The GPU devotes more transistors to data processing
- GPU is particularly effective for programs with high compute-to-memory-access ratio





Hard to Program?

- In the olden days – that is, until just the last few years – programming GPUs meant either:
  - using a graphics standard like OpenGL (which is mostly meant for rendering), or
  - getting fairly deep into the graphics rendering pipeline.
- To use a GPU to do general purpose number crunching, you had to make your number crunching pretend to be graphics.
- This was hard. So most people didn't bother.

Easy to Program?

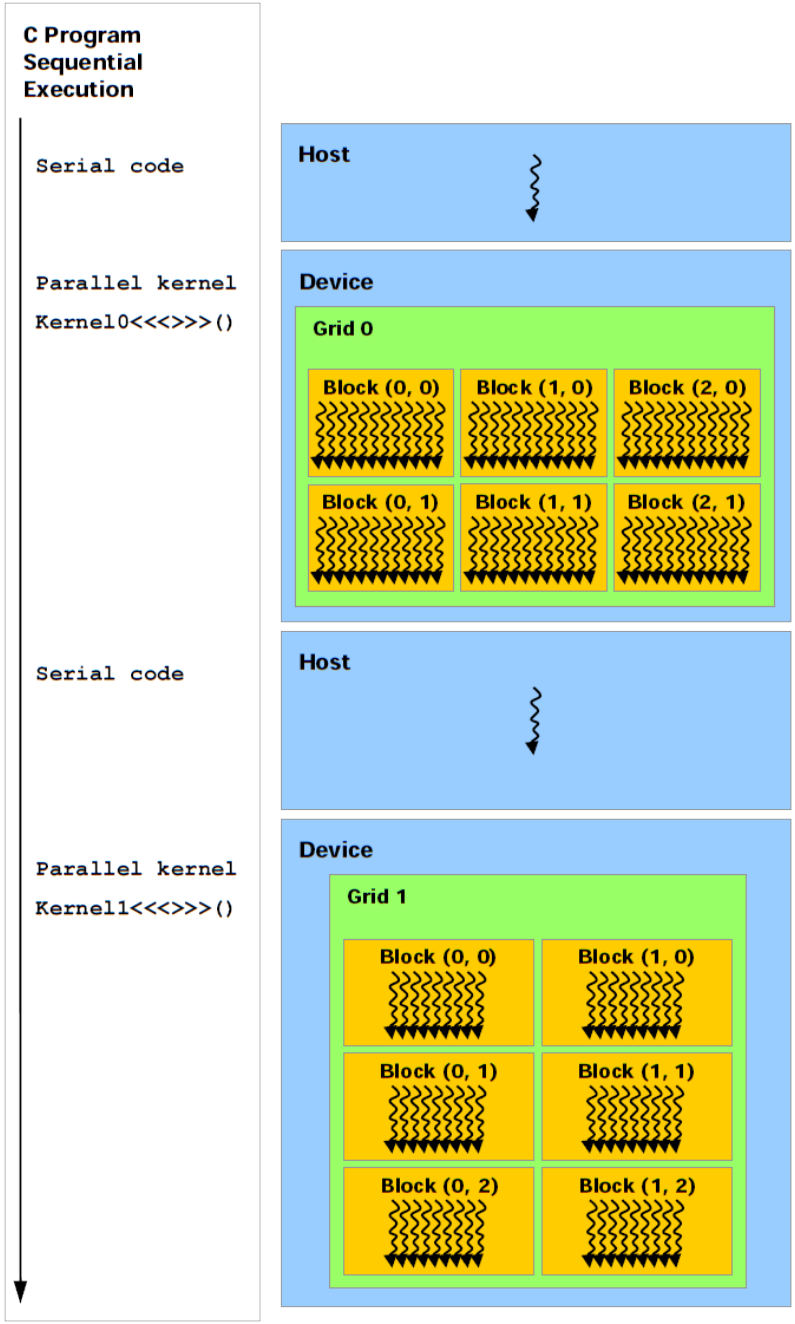
- More recently, GPU manufacturers have worked hard to make GPUs easier to use for general purpose computing.
- This is known as *General Purpose Graphics Processing Units*.

How to Program a GPU

- Part of the program is executed on CPU, part - the kernel - on GPU.
- Proprietary programming language or extensions
  - NVIDIA: CUDA (C/C++)
  - AMD/ATI: StreamSDK/Brook+ (C/C++)
- OpenCL (Open Computing Language): an industry standard for doing number crunching on GPUs.
- Portland Group Fortran and C compilers with accelerator directives.

NVIDIA CUDA

- NVIDIA proprietary
- Formerly known as “Compute Unified Device Architecture”
- Extensions to C to allow better control of GPU capabilities
- Modest extensions but major rewriting of the code
- No Fortran version available



CUDA Example

```
#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
// main routine that executes on the host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/

AMD/ATI Brook+

- AMD/ATI proprietary
- Formerly known as “Close to Metal” (CTM)
- Extensions to C to allow better control of GPU capabilities
- No Fortran version available

Brook+ Example

```
float4 matmult_kernel (int y, int x, int k,
                       float4 M0[], float4 M1[])
{
    float4 total = 0;
    for (int c = 0; c < k / 4; c++)
    {
        total += M0[y][c] * M1[x][c];
    }
    return total;
}

void matmult (float4 A[], float4 B'[], float4 C[])
{
    for (int i = 0; i < n; i++)
    {
        for (j = 0; j < m / 4; j+)
        {
            launch_thread{C[i][j] = matmult_kernel(j, i, k, A, B');}
        }
    }
    sync_threads{}
}
```

http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pd

OpenCL

- Open Computing Language
- Open standard developed by the Khronos Group, which is a consortium of many companies (including NVIDIA, AMD and Intel, but also lots of others)
- Initial version of OpenCL standard released in Dec 2008.
- Many companies will create their own implementations.
  - Apple, Nvidia, AMD/ATI

OpenCL Example (1/2)

```
// create a compute context with GPU device
context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
// create a work-queue
queue = clCreateWorkQueue(context, NULL, NULL, 0);
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(float)*2*num_entries, srcA);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*num_entries, NULL);
// create the compute program
program = clCreateProgramFromSource(context, 1, &fft1D_1024_kernel_src, NULL);
// build the compute program executable
clBuildProgramExecutable(program, false, NULL, NULL);
// create the compute kernel
kernel = clCreateKernel(program, "fft1D_1024");
// create N-D range object with work-item dimensions
global_work_size[0] = num_entries;
local_work_size[0] = 64;
range = clCreateNDRangeContainer(context, 0, 1, global_work_size, local_work_size);
// set the args values
clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 2, NULL, sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, NULL, sizeof(float)*(local_work_size[0]+1)*16, NULL);
// execute kernel
clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

http://en.wikipedia.org/wiki/OpenCL#Example

OpenCL Example (2/2)

```
// This kernel computes FFT of length 1024. The 1024 length FFT is decomposed into
// calls to a radix 16 function, another radix 16 function and then a radix 4 function

__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy) {
  int tid = get_local_id(0);
  int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];
  // starting index of data to/from global memory
  in = in + blockIdx;  out = out + blockIdx;
  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);
  // local shuffle using local memory
  localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);               // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
  localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));
  // four radix-4 function calls
  fftRadix4Pass(data);      // radix-4 function number 1
  fftRadix4Pass(data + 4);  // radix-4 function number 2
  fftRadix4Pass(data + 8);  // radix-4 function number 3
  fftRadix4Pass(data + 12); // radix-4 function number 4
  // coalesced global writes
  globalStores(data, out, 64);
}
```

Portland Group Accelerator Directives

- Proprietary directives in Fortran and C
- Similar to OpenMP in structure
- Currently in beta release
- If the compiler doesn't understand these directives, it ignores them, so the same code can work with an accelerator or without, and with the PGI compilers or other compilers.
- In principle, this will be able to work on a variety of accelerators, but the first instance will be NVIDIA; PGI recently announced a deal with AMD/ATI.
- The directives tell the compiler what parts of the code happen in the accelerator; the rest happens in the regular hardware.

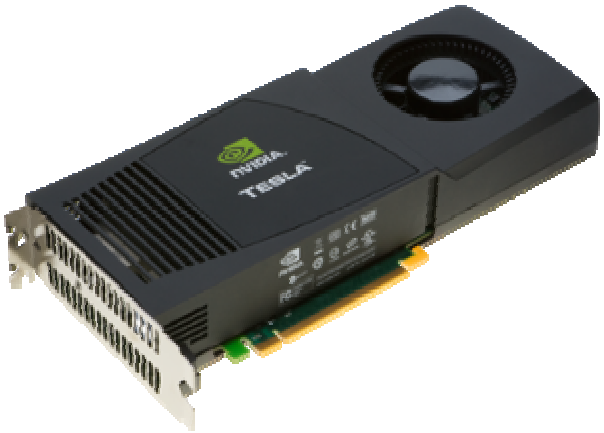
Portland Group Accelerator Directives Example

```
!$acc region
    do k = 1,n1
        do i = 1,n3
            c(i,k) = 0.0
            do j = 1,n2
                c(i,k) = c(i,k) + a(i,j) * b(j,k)
            enddo
        enddo
    enddo
!$acc end region
```

http://www.pgroup.com/resources/accel.htm

NVIDIA Tesla

- NVIDIA now offers a GPU platform named Tesla.
- It consists of their highest end graphics card, minus the video out connector.
- This cuts the cost of the GPU card roughly in half: Quadro FX 5800 is ~$3000, Tesla C1060 is ~$1500 (Q1 2009).

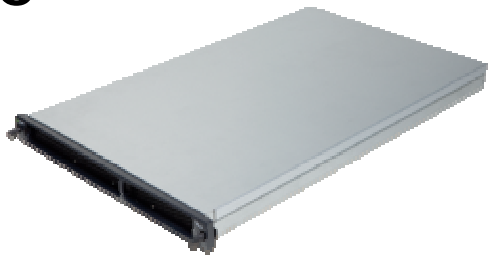


NVIDIA Tesla C1060 Card Specs

- 240 GPU cores
- 1.296 GHz
- Single precision floating point performance: 933 GFLOPs (3 single precision flops per clock per core)
- Double precision floating point performance: 78 GFLOPs (0.25 double precision flops per clock per core)
- Internal RAM: 4 GB
- Internal RAM speed: 102 GB/sec (compared 21-25 GB/sec for regular RAM)
- Has to be plugged into a PCIe slot (at most 8 GB/sec)

NVIDIA Tesla S1070 Server Specs

- 4 C1060 cards inside a 1U server
- Available in both 1.296 GHz and 1.44 GHz
- Single Precision (SP) floating point performance:
  3732 GFLOPs (1.296 GHz) or 4147 GFLOPs (1.44 GHz)
- Double Precision (DP) floating point performance:
  311 GFLOPs (1.296 GHz) or 345 GFLOPs (1.44 GHz)
- Internal RAM: 16 GB total (4 GB per GPU card)
- Internal RAM speed: 408 GB/sec aggregate



Compare x86 vs S1070

Let's compare the best dual socket x86 server today vs S1070.

	Dual socket, Intel 2.66 hex core	**NVIDIA Tesla S1070**
Peak DP FLOPs	128 GFLOPs DP	345 GFLOPs DP (2.7x)
Peak SP FLOPS	256 GFLOPs SP	4147 GFLOPs SP (16.2x)
Peak RAM BW	17 GB/sec	408 GB/sec (24x)
Peak PCIe BW	N/A	16 GB/sec
Needs x86 server to attach to?	No	Yes
Power/Heat	~400 W	~800 W + ~400 W (3x)
Code portable?	Yes	No (CUDA) Yes (PGI, OpenCL)

http://oscer.ou.edu/Workshops/GPGPU/sipe_gpgpu_20090428.ppt

Compare x86 vs S1070

Here are some interesting measures:

	Dual socket, Intel 2.66 hex core	**NVIDIA Tesla S1070**
DP GFLOPs/Watt	~0.3 GFLOPs/Watt	~0.3 GFLOPs/Watt (same)
SP GFLOPS/Watt	0.64 GFLOPs/Watt	~3.5 GFLOPs (~5x)
DP GFLOPs/sq ft	~340 GFLOPs/sq ft	~460 GFLOPs/sq ft (1.3x)
SP GFLOPs/sq ft	~680 GFLOPs/sq ft	~5500 GFLOPs/sq ft (8x)
Racks per PFLOP DP	244 racks/PFLOP DP	181 racks/PFLOP (3/4) DP
Racks per PFLOP SP	122 racks/PFLOP SP	15 racks/PFLOP (1/8) SP

http://oscer.ou.edu/Workshops/GPGPU/sipe_gpgpu_20090428.ppt

What Are the Downsides?

- You have to rewrite your code into CUDA or OpenCL or PGI accelerator directives.
  - CUDA: Proprietary, C/C++ only
  - OpenCL: portable but cumbersome
  - PGI accelerator directives: not clear whether you can have most of the code live inside the GPUs.

Programming for Performance

- The biggest single performance bottleneck on GPU cards today is the PCIe slot:
  - PCIe 2.0 x16: 8 GB/sec
  - 1600 MHz Front Side Bus: 25 GB/sec
  - GDDR3 GPU card RAM: 102 GB/sec per card
- Your goal:
  - At startup, move the data from x86 server RAM into GPU RAM.
  - Do almost all the work inside the GPU.
  - Use the x86 server only for I/O and message passing, to minimize the amount of data moved through the PCIe slot.

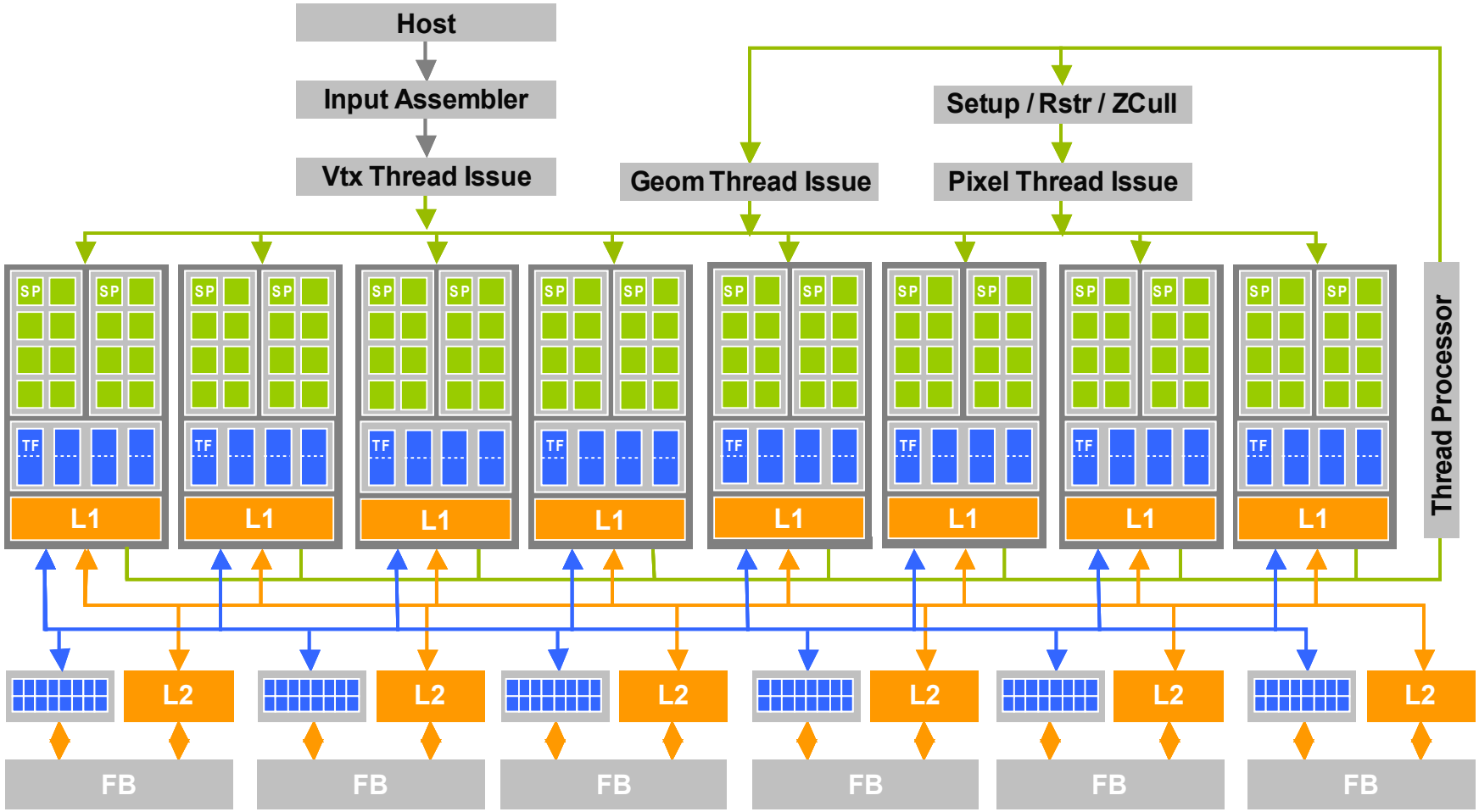
Does CUDA Help?

Example Applications	URL	Speedup
Seismic Database	http://www.headwave.com	66x – 100x
Mobile Phone Antenna Simulation	http://www.accelware.com	45x
Molecular Dynamics	http://www.ks.uiuc.edu/Research/vmd	21x – 100x
Neuron Simulation	http://www.evolvedmachines.com	100x
MRI Processing	http://bic-test.beckman.uiuc.edu	245x – 415x
Atmospheric Cloud Simulation	http://www.cs.clemson.edu/~jesteel/clouds.html	50x

http://www.nvidia.com/object/IO_43499.html

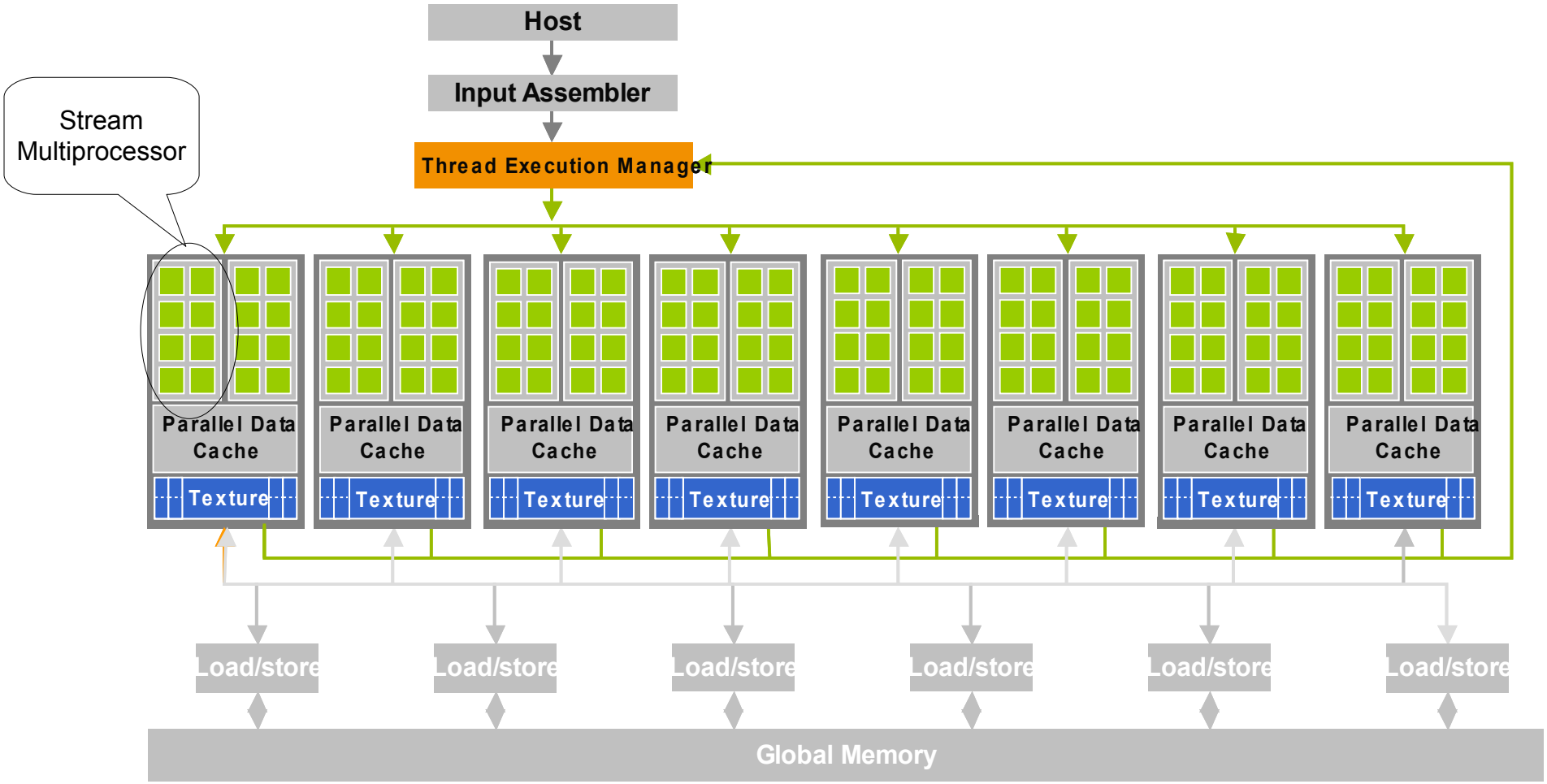
G80 Architecture – Graphics Mode

- Unlike the previous versions, which have had separate vertex, geometry and pixel processors, G80 employs a unified architecture
- GeForce 8800 is composed of 681 million transistors



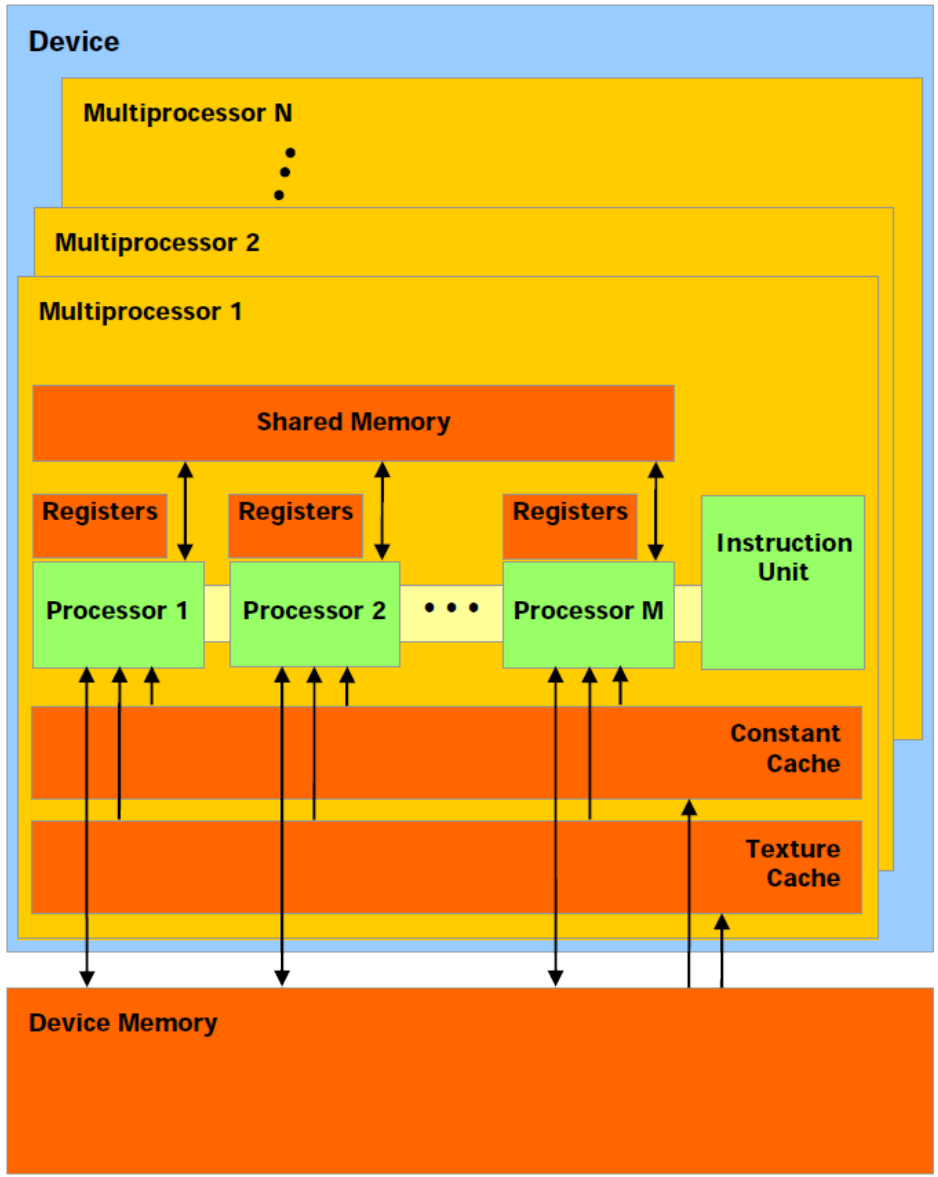
G80 Architecture – Computation Mode

- Processors execute computing threads
- New operating mode/HW interface for computing



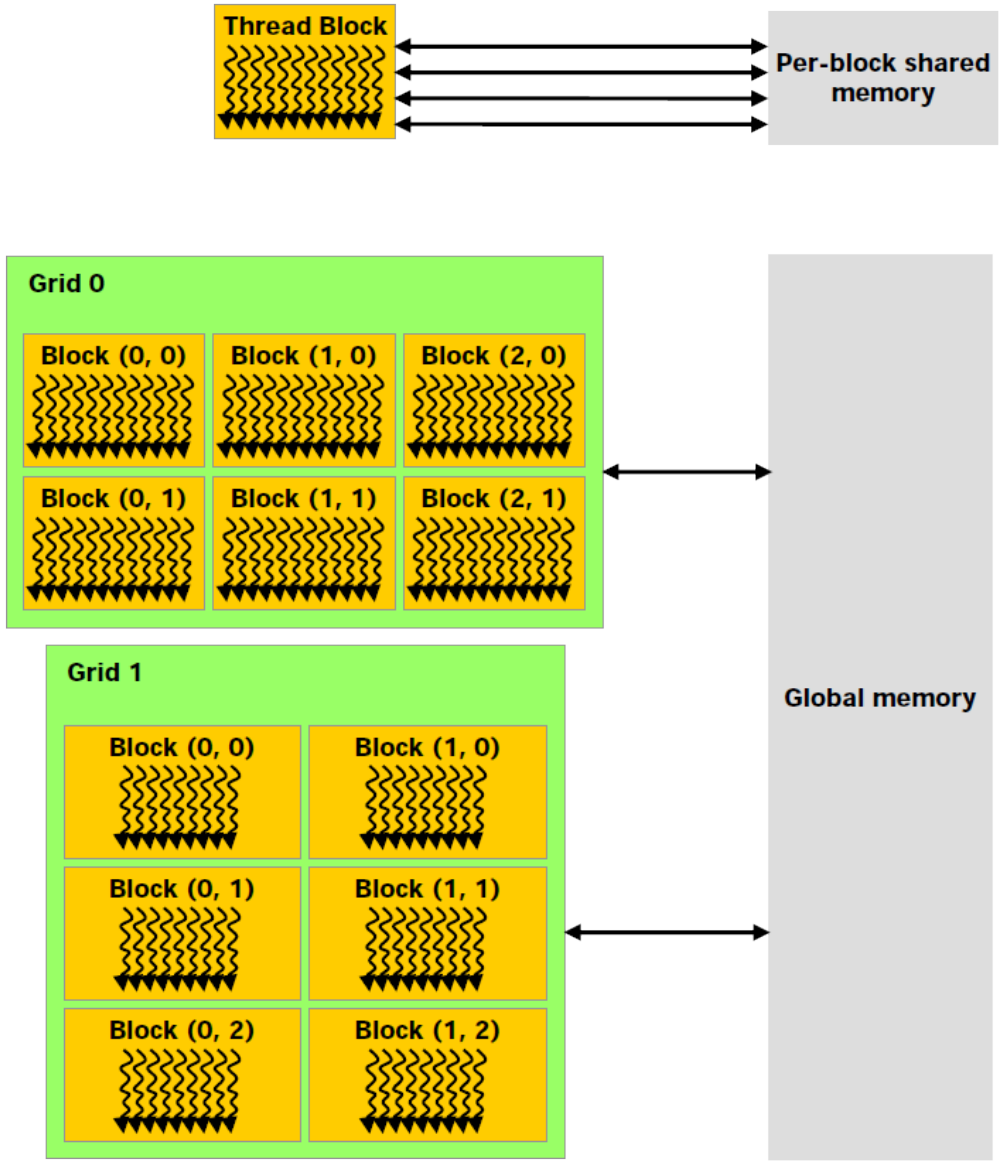
The Multiprocessor

- A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multithreaded instruction unit, and on-chip shared memory.
- The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead.

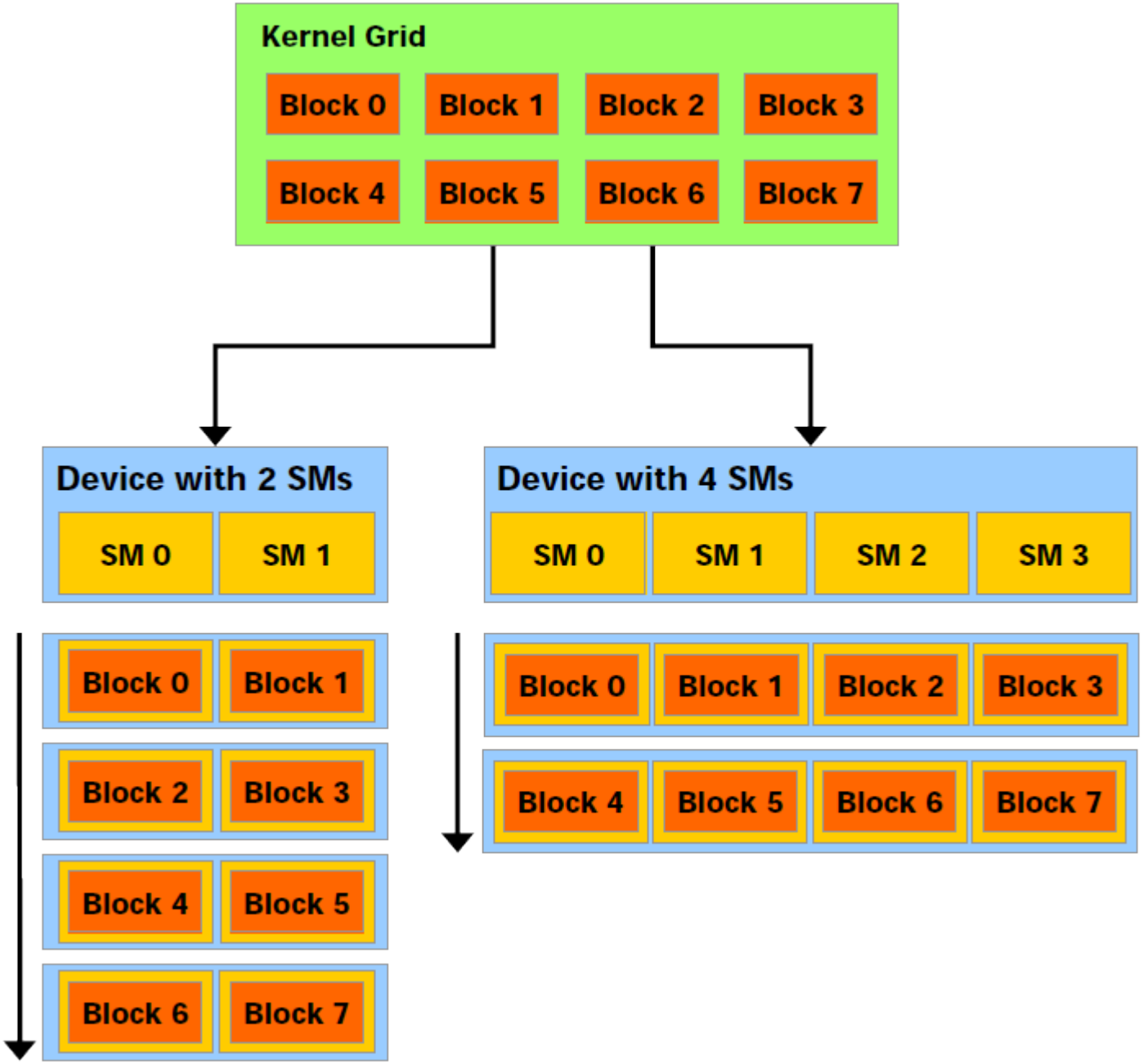


The Warps

- Running threads are divided into blocks, which share the multiprocessor and can exchange data through the local memory
- Multiprocessor employs a new architecture we call SIMT (single-instruction, multiple-thread).
  - The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.
  - The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.



The Mapping Between Blocks and SMs



The Warps

- Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.
- When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit.
- Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp.
  - A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.
  - If threads of a warp diverge via a datadependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.
  - Branch divergence occurs only within a warp; different warps execute independently regardless

Memory

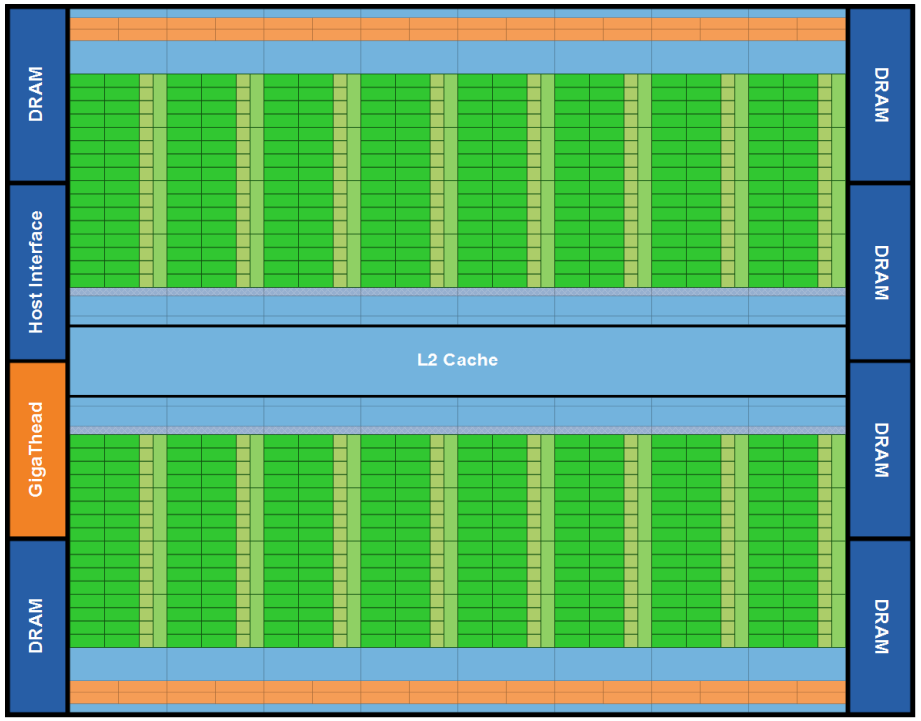
- Each multiprocessor has on-chip memory of the four following types:
  - One set of local 32-bit registers per processor,
  - A parallel data cache or shared memory that is shared by all scalar processor cores and is where the shared memory space resides,
  - A read-only constant cache that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory,
  - A read-only texture cache that is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory; each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering

Compute Capability

- NVIDIA defines the computing capabilities of their devices by a number
- G80 has a Compute Capability of 1.1:
  - The maximum number of threads per block is 512;
  - The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively;
  - The maximum size of each dimension of a grid of thread blocks is 65535;
  - The warp size is 32 threads;
  - The number of registers per multiprocessor is 8192;
  - The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks;
  - The total amount of constant memory is 64 KB;
  - The total amount of local memory per thread is 16 KB;
  - The cache working set for constant memory is 8 KB per multiprocessor;
  - The cache working set for texture memory varies between 6 and 8 KB per multiprocessor;
  - The maximum number of active blocks per multiprocessor is 8;
  - The maximum number of active warps per multiprocessor is 24;
  - The maximum number of active threads per multiprocessor is 768;
  - The limit on kernel size is 2 millions of microcode instructions;

NVIDIA Fermi

- 512 Cores
- 40 nm technology
- First silicon produced at TSMC foundry in September 2009 with 2% yield (7 chips out of 416)



Fermi Architecture

- Each Fermi SM includes
  - 32 cores,
  - 16 load/store units,
  - four special-function units,
  - a 32K-word register file,
  - 64K of configurable RAM,
  - thread control logic.
- Each core has both floating-point and integer execution units.