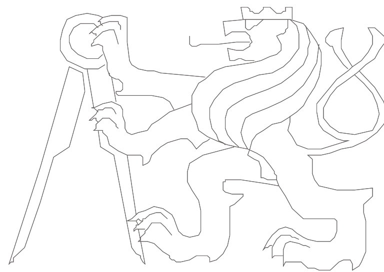
Advanced Computer Architectures

GPU (Graphics processing unit),
GPGPU (General-purpose **computing** on GPU;
General-purpose GPU)
and GPU **Computing**

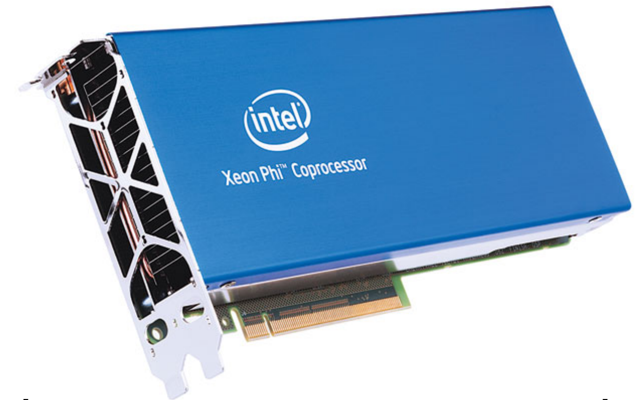


Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

Motivation

- **Tianhe-1A** - Chinese Academy of Sciences' Institute of Process Engineering (CAS-IPE)
- Molecular simulations of 110 milliards atoms (1.87 / 2.507 PetaFLOPS)
- Rmax: 2.56 Pflops, Rpeak: 4,7 PFLOPS
- **7 168** Nvidia Tesla M2050 (448 Thread processors, 512 GFLOPS FMA)
- **14 336** Xeon X5670 (6 cores / 12 threads)
- „If the Tianhe-1A were built only with CPUs, it would need more than 50,000 CPUs and consume more than 12MW. As it is, the Tianhe-1A consumes 4.04MW.“
  http://www.zdnet.co.uk/news/emerging-tech/2010/10/29/china-builds-worlds-fastest-supercomputer-40090697/
  which leads to 633 GFlop/kWatt (K Computer - 830 GFlop/kWatt)
- It uses own interconnection network: Arch, 160 Gbps
- There are already three supercomputers utilizing graphics cards in China, Tianhe-1 (AMD Radeon HD 4870 X2), Nebulae (nVidia Tesla C2050) and Tianhe-1A
- http://i.top500.org/system/176929
- http://en.wikipedia.org/wiki/Tianhe-I

Motivation

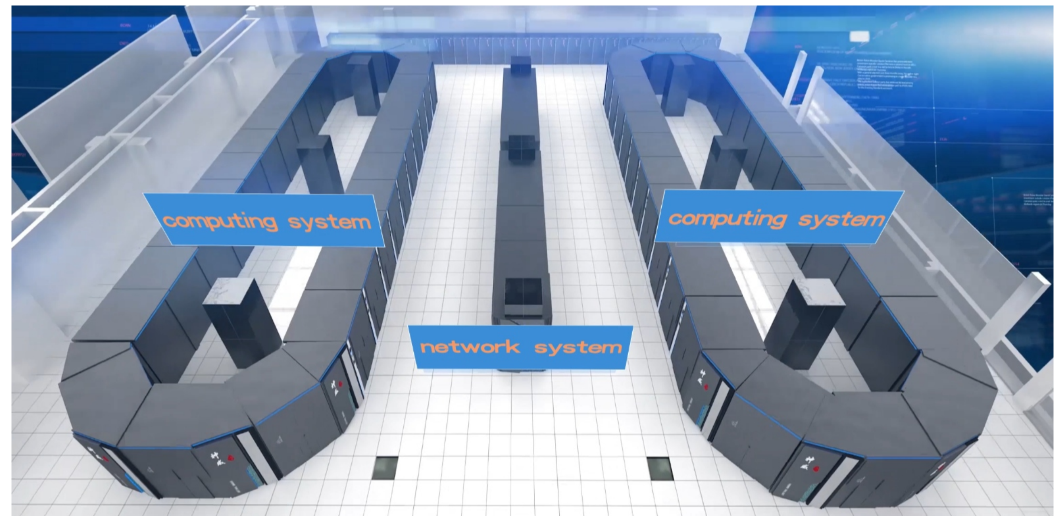


Tianhe-2

- 33.86 PFLOPS
- Power consumption 17 MW
- Kylin Linux
- 16,000 nodes, each built from 2 Intel Ivy Bridge Xeon processors and 3 Intel Xeon Phi coprocessors (61 cores) = 32000 CPU and 48000 coprocessors, 3 120 000 cores in total
- Fortran, C, C++, Java, OpenMP, and MPI 3.0 based on MPICH
- A broadcast operation via MPI was running at 6.36 GB/s and the latency measured with 1K of data within 12,000 nodes is about 9 us
- directive-based intra-node programming model by OpenMC (in progress) – instead of OpenMP, CUDA, OpenACC, or OpenCL

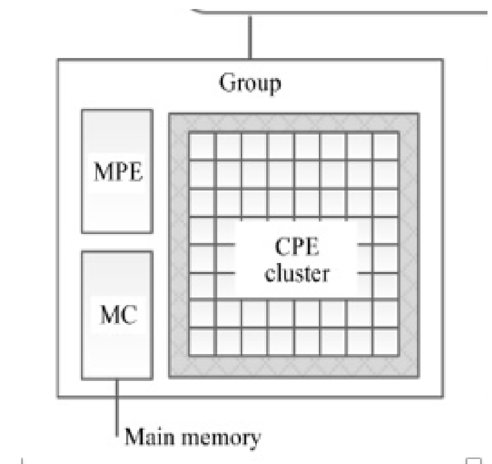
Sunway TaihuLight

- 93 PFLOPS (LINPACK benchmark), peak 125 PFLOPS
- Interconnection 14 GB/s, Bisection 70 GB/s
- Memory 1.31 PB, Storage 20 PB
- 40,960 SW26010 (Chinese) – total 10,649,600 cores
- SW26010 256 processing cores + 4 management
- 64 KB of scratchpad memory for data (and 16 KB for instructions)
- Sunway RaiseOS 2.0.5 (Linux based)
- OpenACC (for open accelerators) programming standard
- Power Consumption 15 MW (LINPACK)



Sunway TaihuLight

- Core group
  - Management Processing Element (MPE)
  - 64 Computing Processing Elements (CPEs)
- 4 core groups on SW26010 chip
- 8 floating point operations per cycle per CPE core (64 bit), 16 MPE



Source: Report on the Sunway TaihuLight System,
Jack Dongarra, University of Tennessee, Oak Ridge National Laboratory

More accurate results

Multiply-Add (MAD):

A × B = Product (truncate extra digits)

\+

C = Result

Fused Multiply-Add (FMA)

A × B = Product (retain all digits)

\+

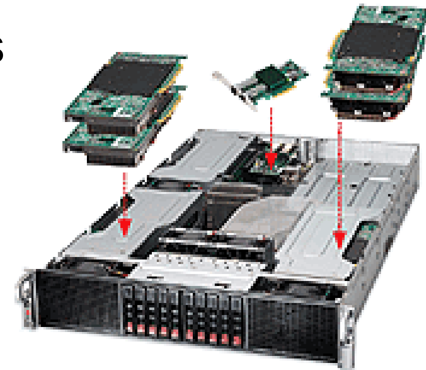
C = Result

- **API** (Application Programming Interface): OpenGL, DirectX – the GPU can be considered in such applications as coprocessor of main CPU

Motivation

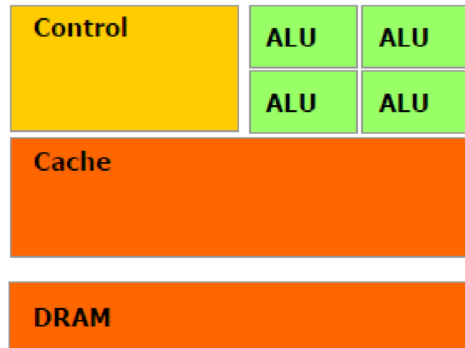
- **Estonia Donates Project:** Our GPGPU supercomputer is GPU-based massively parallel machine, employing more than thousand parallel streaming processors. Using GPU-s is very new technology, very price- and cost-effective compared to old CPU solutions. **Performance (currently):**
  6240 streaming processors + 14 CPU cores
  23,2 arithmetic TFLOPS (yes, 23 200 GFLOPS)
  http://estoniadonates.wordpress.com/our-supercomputer
- **Supermicro: 2026GT-TRF-FM475**
  - 2x Quad/Dual-Core Intel® Xeon® processor 5600/5500 series
  - Intel® 5520 chipset with QPI up to 6.4 GT/s + PLX8648
  - Up to 96GB of Reg. ECC DDR3 DIMM SDRAM
  - **FM475: 4x NVIDIA Tesla M2075 Fermi GPU Cards**
  - **FM409: 4x NVIDIA Tesla M2090 Fermi GPU Cards**

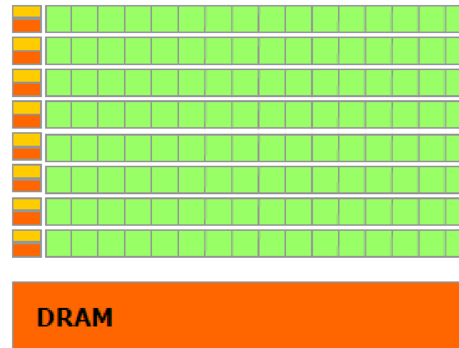
  http://www.supermicro.com/GPU/GPU.cfm#GPU_SuperBlade

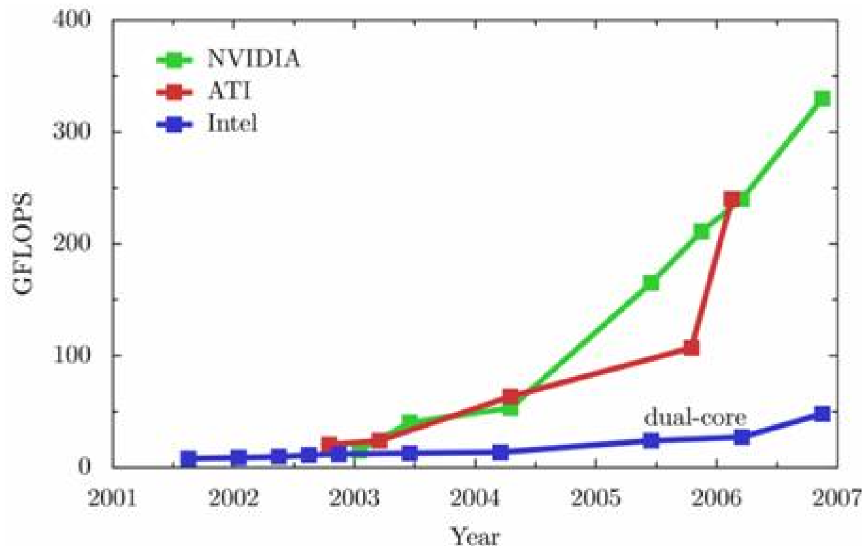


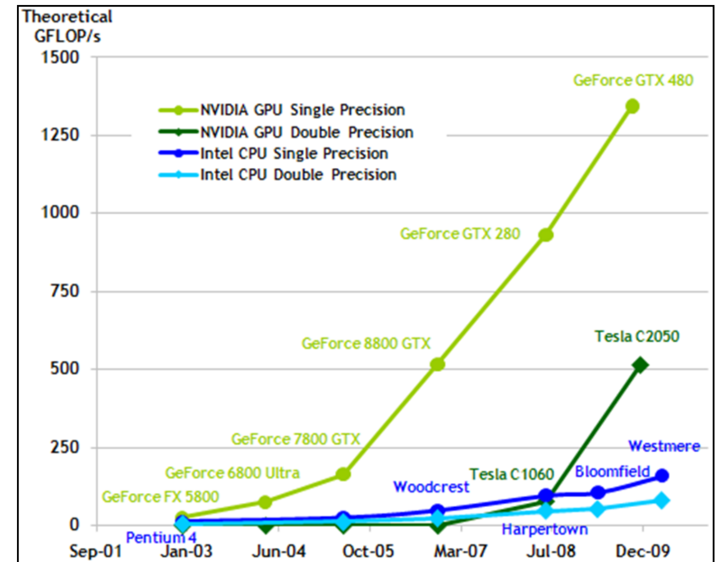
- **„FASTRA: the world's most powerful desktop supercomputer“**
  We have now developed a PC design that incorporates 13 GPUs, resulting in a massive 12TFLOPS of computing power.
  http://fastra2.ua.ac.be/

CPU vs. GPU









Nvidia: „GPU computing is possible because today's GPU does much more than render graphics: It sizzles with a teraflop of floating point performance and crunches application tasks designed for anything from finance to medicine.“ Source: www.nvidia.com

Performance metrics – do you remember?

Let $\{R_i\}$ be execution speeds of different programs

$i = 1, 2, ..., m$ measured in MIPS (MFLOPS), or IPS (FLOPS)

- The arithmetic mean performance: $R_a = \sum_{i=1}^{m} \frac{R_i}{m} = \frac{1}{m} \sum_{i=1}^{m} R_i$

  R_a is equally weighted *(1/m)* in all programs and is proportional to the sum of the IPC, but not the sum of execution times (inversely proportional). Arithmetic mean (average) not generally usable:

$$R_a = \frac{1}{2}(R_1 + R_2) = \frac{1}{2}\left(\frac{IC_1}{T_1} + \frac{IC_2}{T_2}\right) = \frac{1}{2}\left(\frac{IC_1}{IC_1.CPI_1 T_{CLK}} + \frac{IC_2}{IC_2.CPI_2 T_{CLK}}\right) =$$

$$= \frac{1}{T_{CLK}}\left(\frac{IPC_1 + IPC_2}{2}\right) = \frac{1}{T_{CLK}}\left(\frac{IC_1}{2C_1} + \frac{IC_2}{2C_2}\right) \quad but \quad IPC_{1,2} = \frac{IC_1 + IC_2}{C_1 + C_2}$$

  Only iff $C_1 = C_2$ (total number of cycles of both programs is equal) then R_a is usable

- In practice: The arithmetic mean of execution speed of two (or more) different programs is not related to overall execution speed! Not usable!

Performance metrics – do you remember?

- The geometric mean:

$$R_g = \prod_{i=1}^{m} R_i^{\frac{1}{m}}$$

It does not summarize real performance. It has no inverse relation to overall execution time of all programs. Usable only for comparison with normalized results to reference compute.

- The harmonic mean:

$$R_h = \frac{m}{\sum_{i=1}^{m} \frac{1}{R_i}}$$

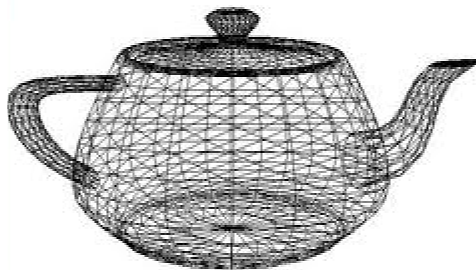
$$R_h = \frac{2}{\frac{1}{R_1}+\frac{1}{R_2}} = \ldots = \frac{1}{T_{CLK}}\left(\frac{2}{CPI_1+CPI_2}\right) = \frac{1}{T_{CLK}}\frac{2\,IC_1\,IC_2}{C_1 IC_2+C_2 IC_1}$$

Only iff $IC_1 = IC_2$ (both programs are of the same size) then R_h is usable

- There exist even weighted versions of these performance metrics.

3D graphics pipeline

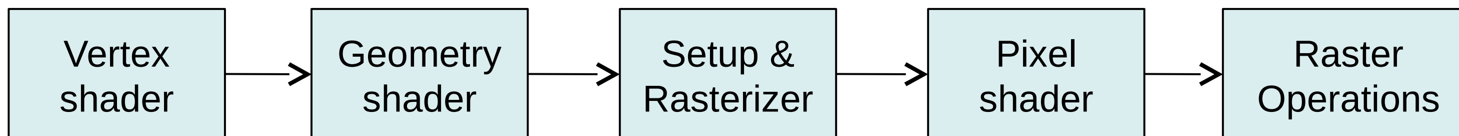
- it is a way of processing graphic data to achieve an image (the input is representation of a 3D scene, output is 2D image)
- Next phases of geometric/graphics data processing:
  - transformations (scaling, rotations, translation,..) – matrix multiplication
  - lighting (only vertexes) – dot products of vectors
  - projection transformations (into camera 3D coordinates) – matrix multiplication,
  - clipping, rasterization and texture mapping (pixels from this stage)



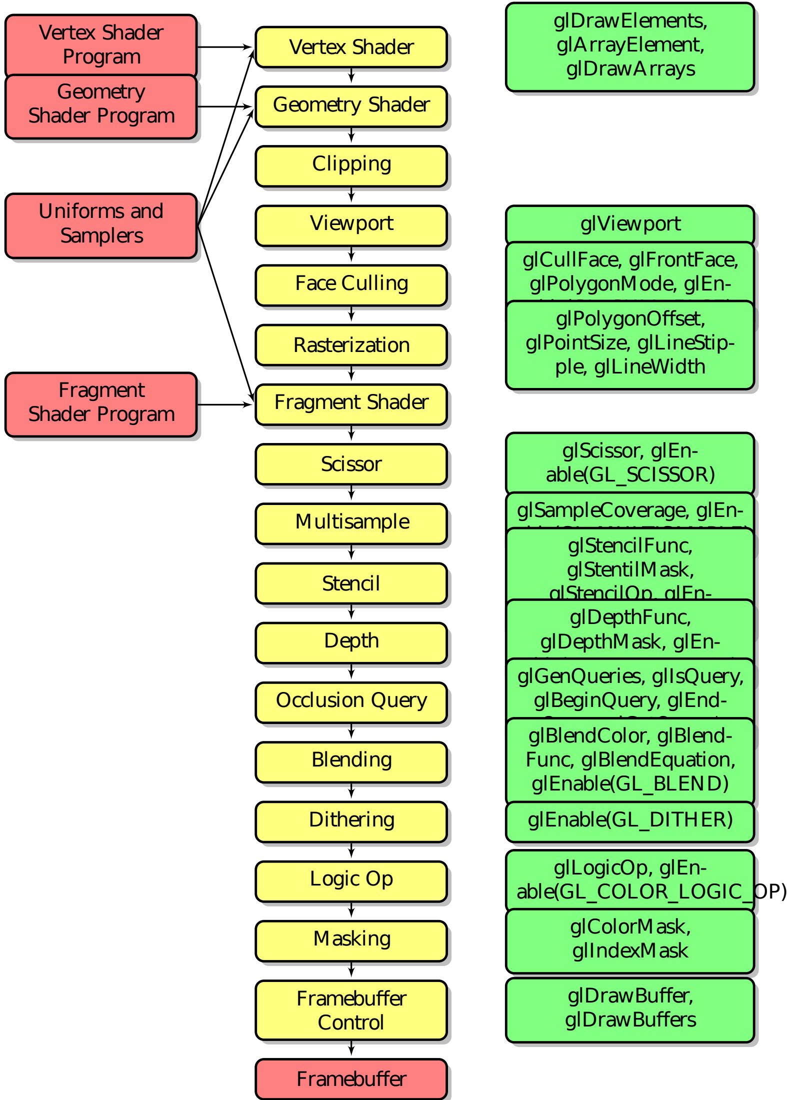
- What is important for us -> HW support required – GPU development

GPU

- What was the original solution?
  - narrowly specialized single-purpose HW according to the principle of 3D graphic pipeline:
  - vertex shader (3D model manipulation, vertexes lightening),
  - geometry shader (adds/removes vertexes,..)
  - pixel shader (more precise: fragment shader) – (input is rasterization output; defines color of „pixel“ (fragment) - textures..)
  - ROP unit (create pixel from pixel fragments, optimizes image for view)   ROP – Raster OPerator, (ATI: Element Render Back-End)



OpenGL Pipeline

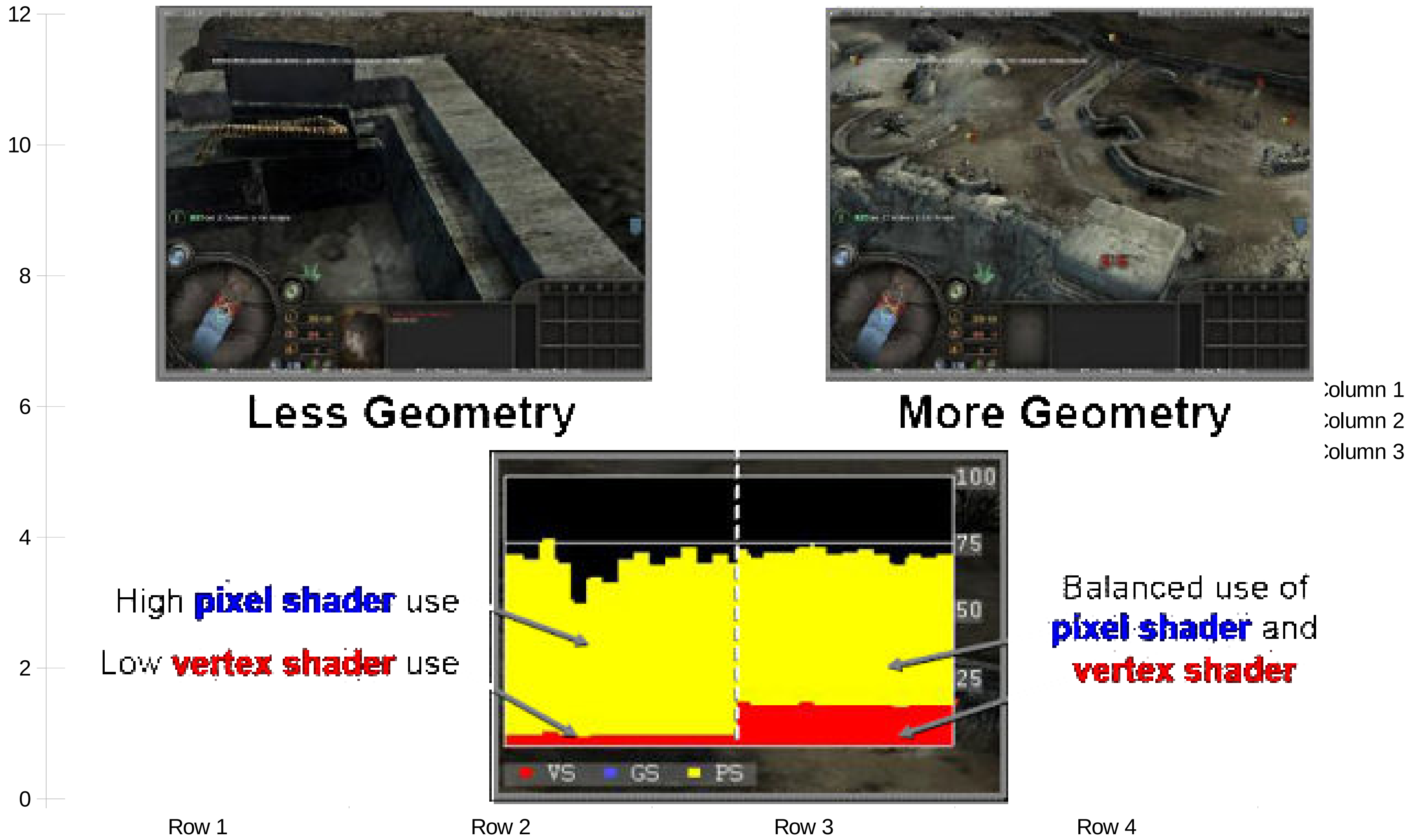


GPU

- Today concept (and future directions)?
  - HW function in each phase much more flexible, programmable (not only the computation operation „program“, but even support of control-flow primitives)
  - 16, 24, 32, 64 floating point precision supported today
  - unified shaders (each can be used for all functions..) - (ATI Xenos, GeForce 8800) – advantage?
    - low detail scene (vertex shader vs. pixel shader)
    - highly detailed scene (vertex shader vs. pixel shader)

- What does it mean for us / what are the benefits?

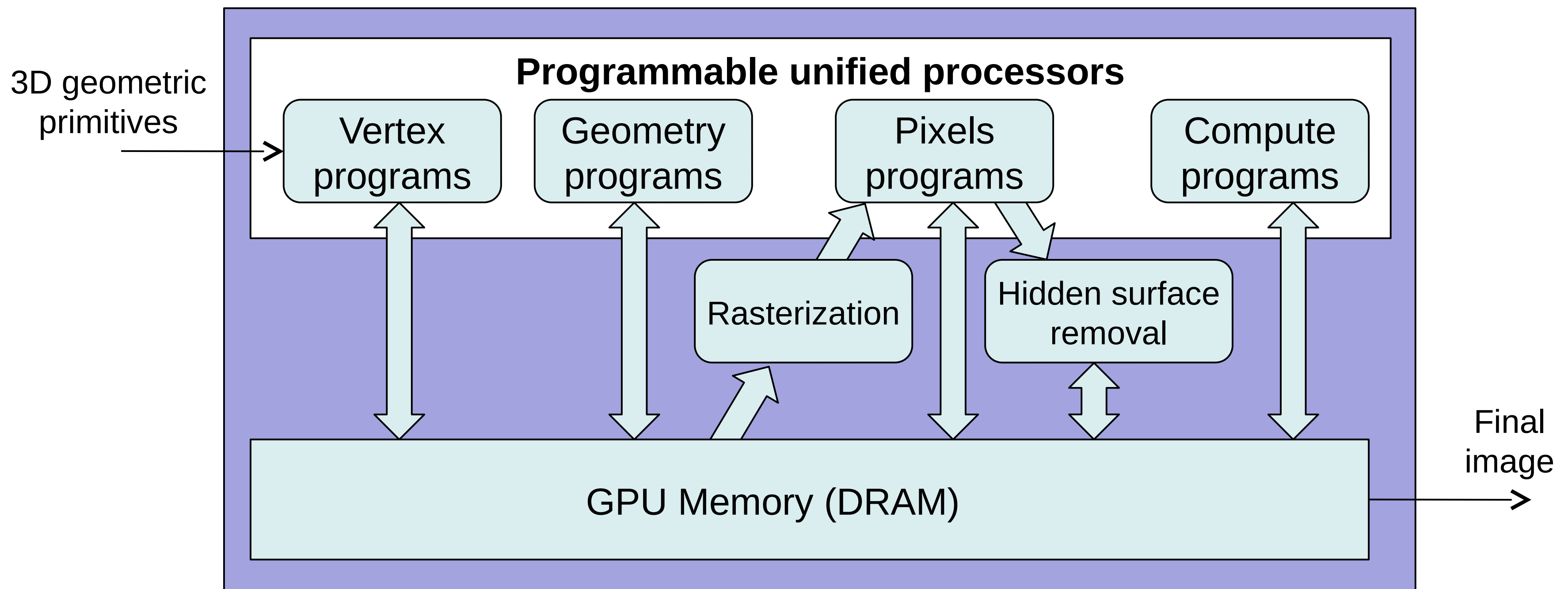
GPU



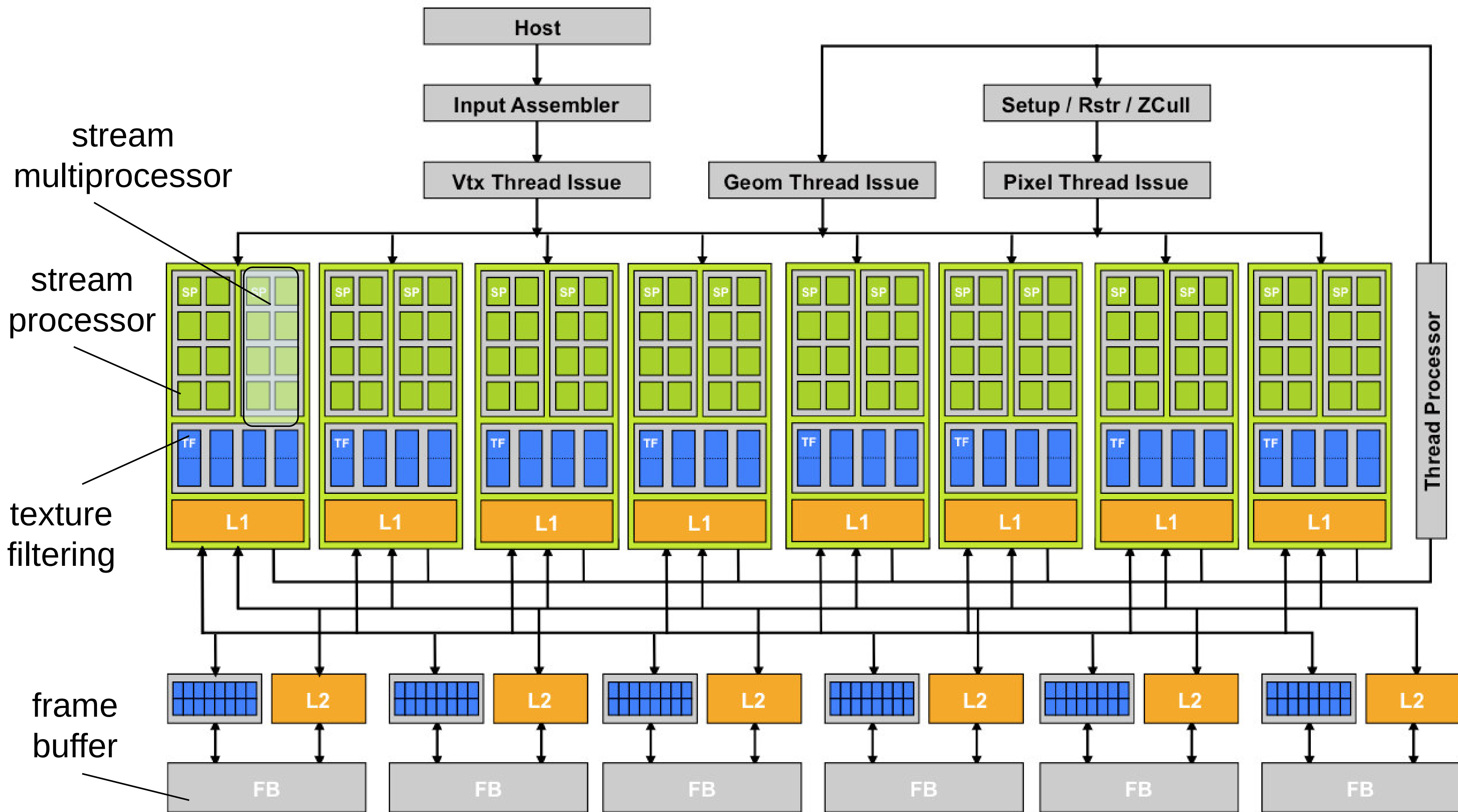
http://techreport.com/articles.x/11211/3

GPU

- The Shader Unification Principle – Architecture provides one large set of data paralleled floating point processors generic enough to replace the functions of individual shaders

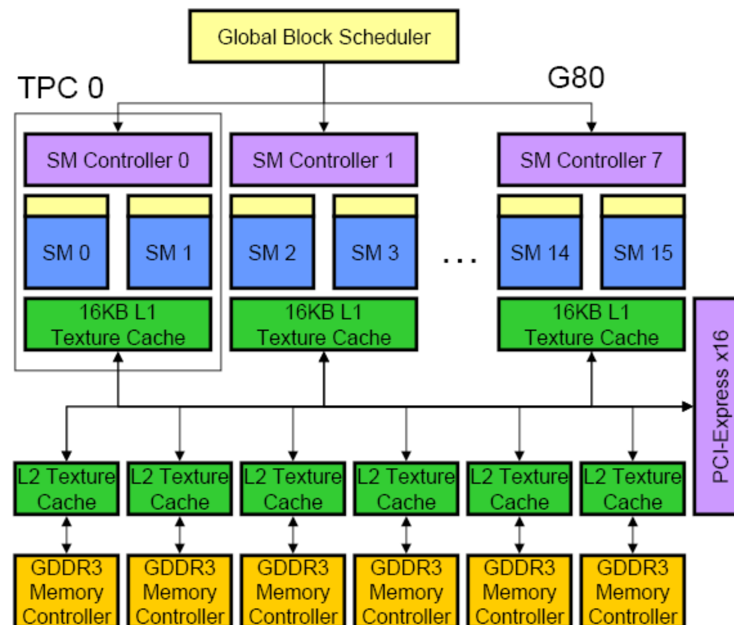


GPU - GeForce 8800

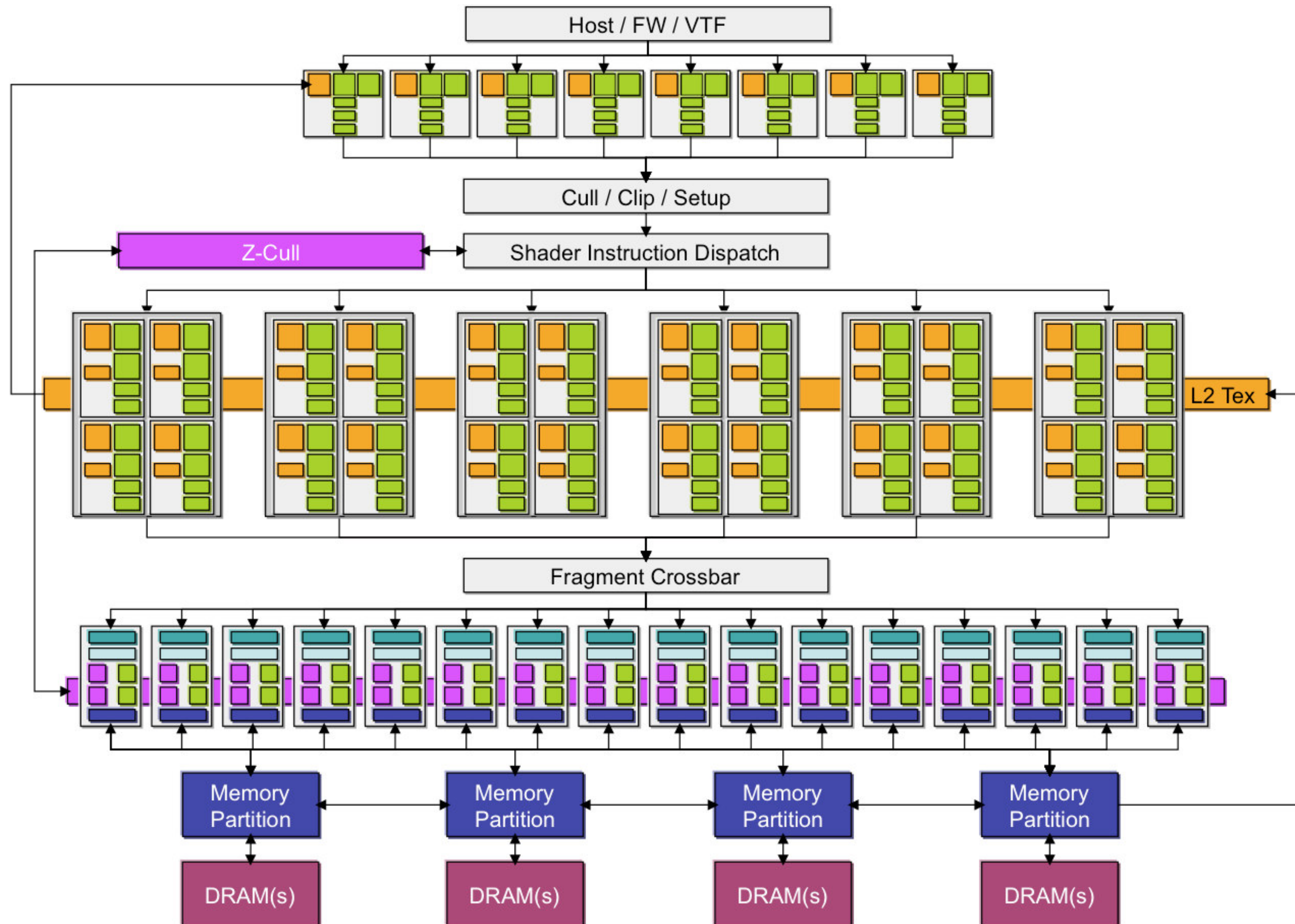


GeForce 8800 – hardware limits

- 512 threads in one block
- 8 blocks on one SM (Streaming Multiprocessor)
- 768 threads on one SM > 768x16=12 288 threads in total!
- 128 threads simultaneously running
- 16 384 bytes shared cache for one SM
- two threads from different blocks cannot cooperate together

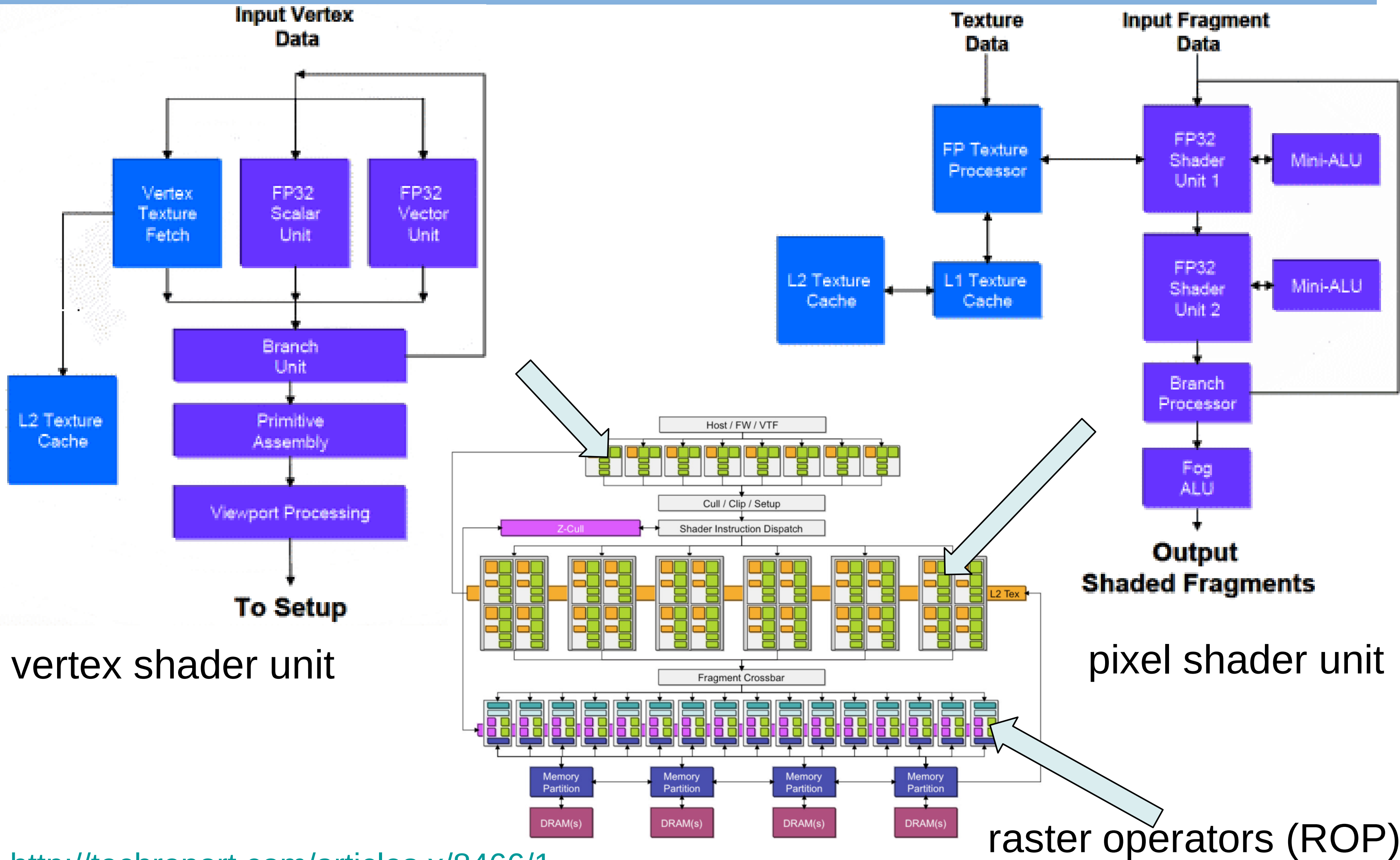


GPU - GeForce 7800 – for comparison



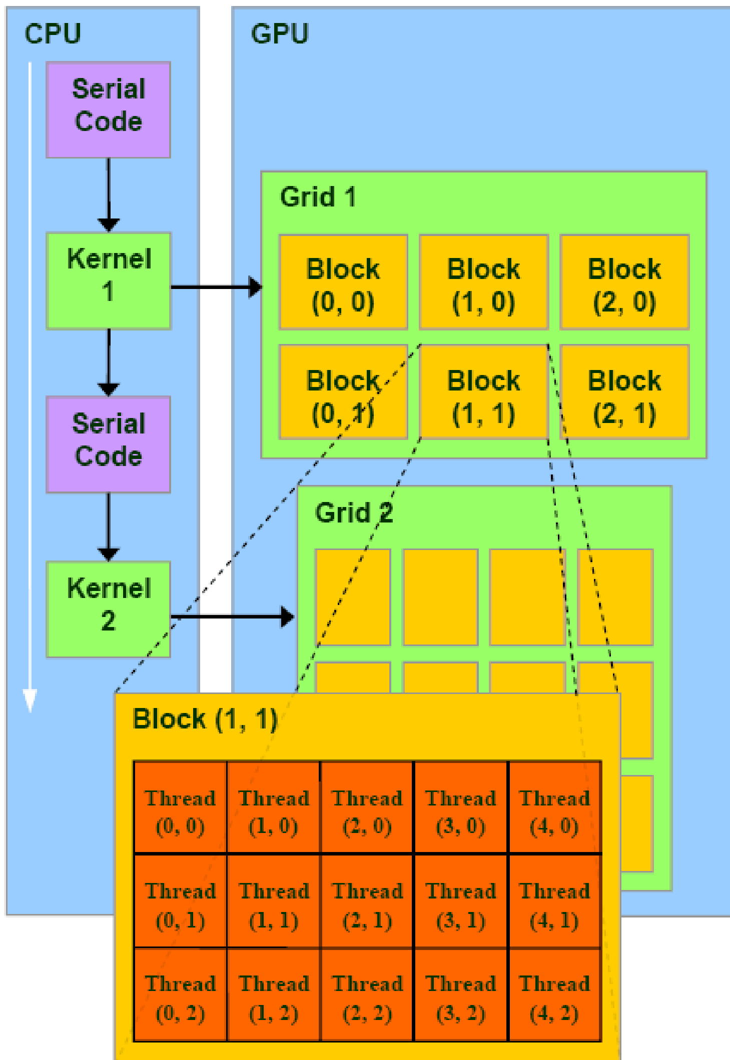
http://techreport.com/articles.x/8466/1

GPU - GeForce 7800



http://techreport.com/articles.x/8466/1

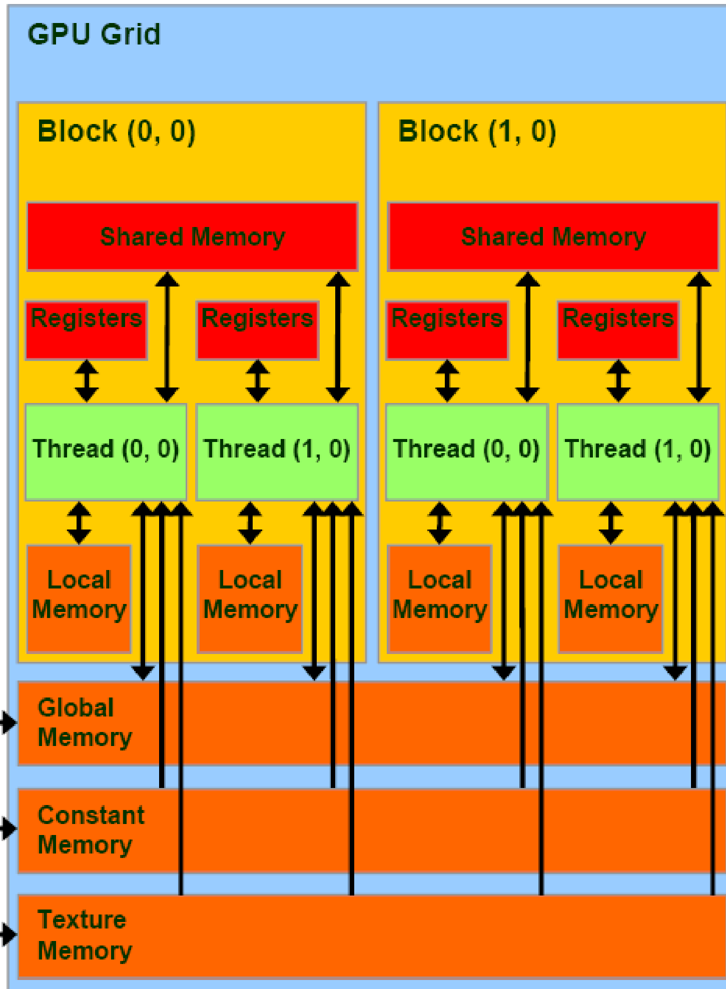
CUDA (*Compute Unified Device Architecture*)



- Kernel – part of application running on GPU
- Kernel – is executed on Grid
- Grid – lattice of thread blocks
- Thread block – group of threads starting at same address and communicating through shared memory and synchronization barriers (<=512)
- One block maps to one SM (Streaming Multiprocessor), SIMD (Single Instruction, Multiple Data)
- One thread in same block to one execution unit (Streaming Processor core - SP core)

http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=2

CUDA (*Compute Unified Device Architecture*)



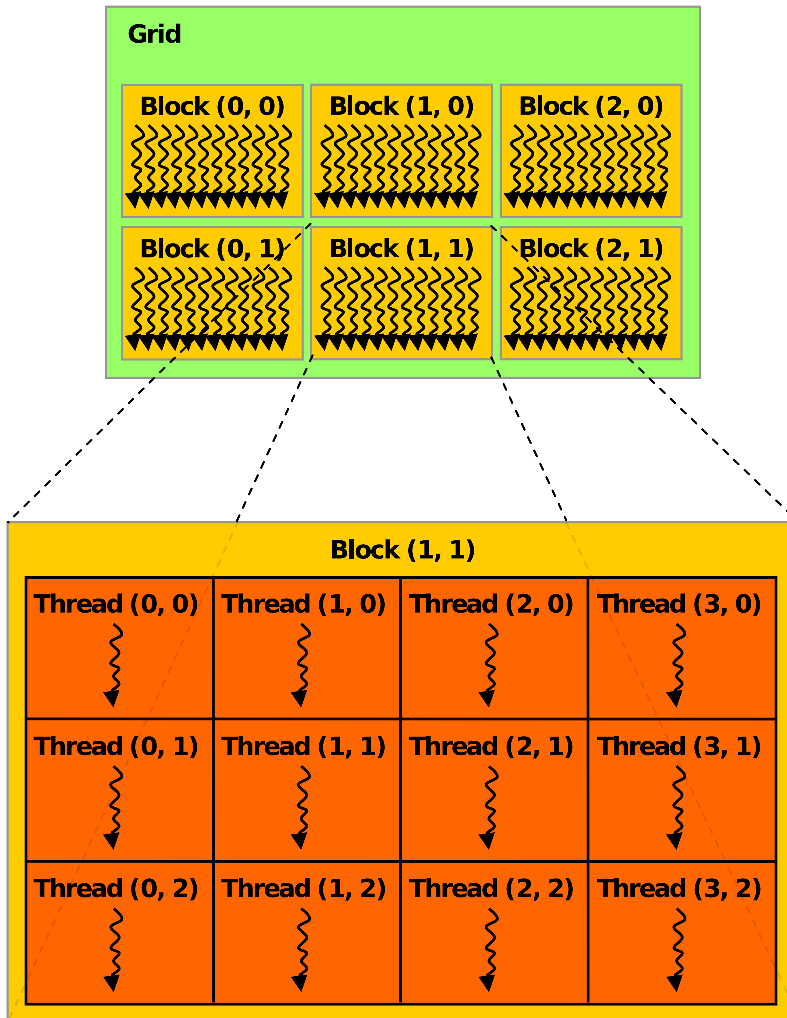
CUDA memory model:

- Registers and shared memory – on chip
- Local memory – frame buffer
- Constant Mem and Texture Mem – frame buffer, only for reading, cached on chip, coherence?
- Global memory

red = fast (on chip)
orange = slow (DRAM)

http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=3

CUDA – Threads, Blocks, Grid

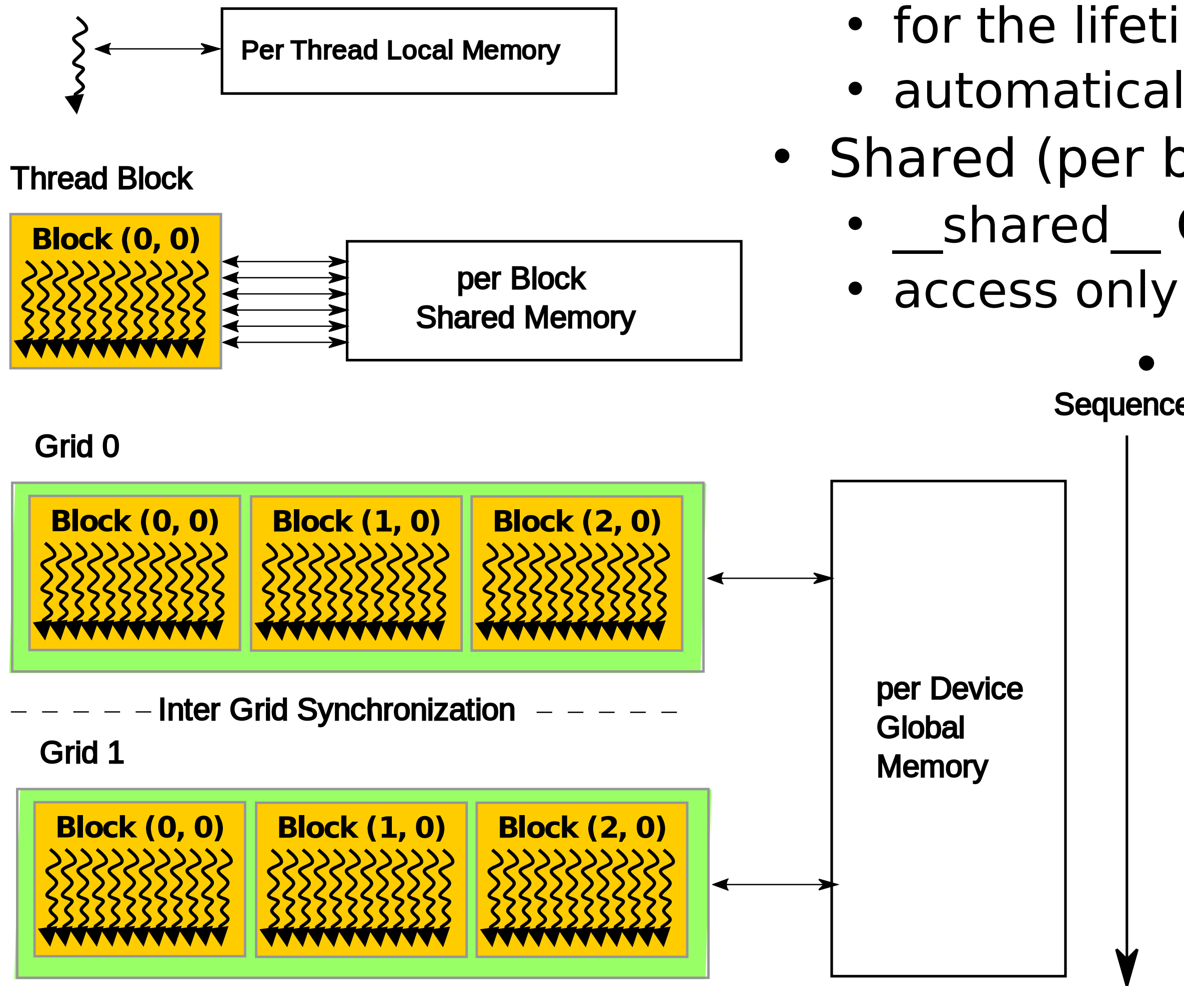


- SIMD (Single Instruction, Multiple Data) architecture
- Thread execution managed in **warp**s
- Warp size 32 threads
  4 on each of 8 SPs
- SIMT broadcasted synchronously to all SPs
- some inactive, branches
- warp diverge, converge
- branch synchronization stack

SIMT Terminology, Nvidia, CUDA, CPU

Nvidia CUDA	OpenCL	Hennessy & Patterson
Thread	Work-item	Sequence of SIMD Lane operations
Warp	Wavefront	Thread of SIMD Instructions
Block	Workgroup	Body of vectorized loop
Grid	NDRange	Vectorized loop

CUDA – Memory Local, Shared, Global



- Local (thread private) memory
  - for the lifetime of the thread
  - automatically by the compiler
- Shared (per block) memory
  - __shared__ CUDA keyword
  - access only threads within the block
- Global memory
  - __device__ keyword
  - accessible to all threads and host (CPU).
  - allocated and deallocated by the host
  - Used to initialize the data that the GPU will work on

Source: https://nyu-cds.github.io/python-gpu/02-cuda/

Memory model – Types

- local, Constant, and Texture – off-chip, cached
- each SM has a L1 cache for global memory
- all SMs share a L2 cache
- constant memory – read only, shorter latency, higher through.
- texture memory is read only.

Type	Read/write	Speed
Global memory	read and write	slow, but cached
Texture memory	read only	cache optimized for 2D/3D access pattern
Constant memory	read only	where constants and kernel arguments are stored
Shared memory	read/write	fast
Local memory	read/write	used when it does not fit in to registers part of global memory slow but cached
Registers	read/write	fast

CUDA – Declaration of Variables Placement

Declaration	Memory	Scope	Lifetime
int v	register	thread	thread
int vArray[10]	local	thread	thread
__shared__ int sharedV	shared	block	block
__device__ int globalV	global	grid	application
__constant__ int constantV	constant	grid	application

- Speed (Fast to slow):
- Register file
- Shared Memory
- Constant Memory
- Texture Memory
- Local Memory (thread local global memory) and Global Memory

CUDA C

- CUDA C is based on C language with extensions but even some limitations
- Kernel – specified by __global__
- each thread executing kernel is assigned by unique ID

```
// Definition of vector addition function:
void VecAdd(int n, float* A, float* B, float* C)
{
   for(int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

int main()
{
   ...
   // Sum of vectors of length N:
   VecAdd(N, A, B, C);
}
```

CUDA C

- CUDA C is based on C language with extensions but even some limitations
- Kernel – specified by __global__
- each thread executing kernel is assigned by unique ID

```
  // Kernel definition
__global__ void VecAdd(int n, float* A, float* B, float* C)
{
   int i = threadIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}

int main()
{
   ...
   // Execute N threads onGPU – Sum of vectors of length N:
   VecAdd<<<1, N>>>(N, A, B, C);
}
```

<<<number of blocks, number of threads>>>

CUDA C

- To support native support for vector, 2D and 3D matrices, variable *threadIdx* is implemented as 3-components vector
- For 2D block of dimensions (Dx, Dy), thread on position (x,y) has its ID (x + y Dx)
- For 3D block of dimensions (Dx, Dy, Dz), thread on position (x,y,z) has its id ID (x + y Dx + z Dx Dy)

```
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
```

CUDA C

```
int main()
{
   ...
   // Execute kernel as block of dimensions N * N * 1 threads
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

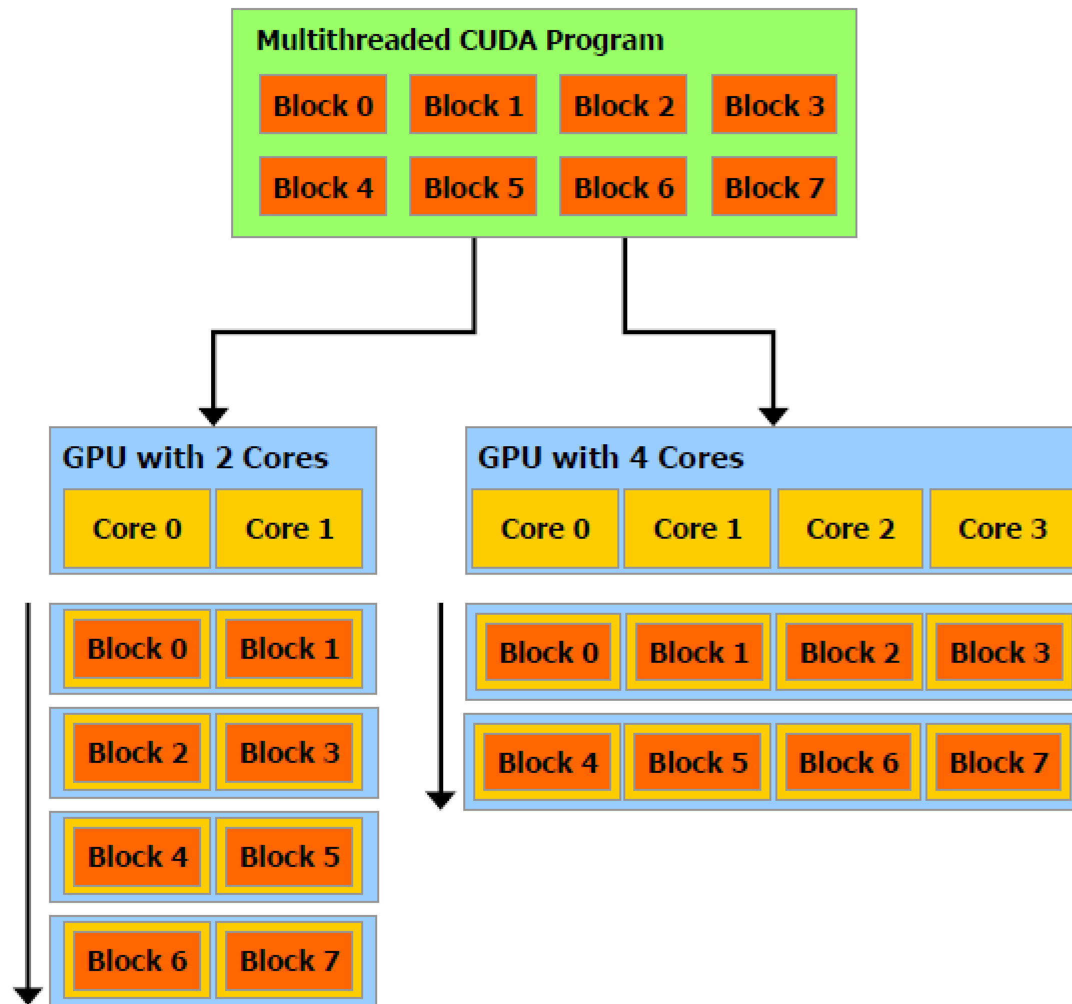
Number of threads inside a block is limited. All threads belonging to the same block are executed on the same processor (SM) and share limited memory resources.

Today single block of threads can contain limited number of threads, where today limit is usually 1024 threads.

CUDA C

```
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
      C[i][j] = A[i][j] + B[i][j];
}
int main()
{
   ...
   dim3 threadsPerBlock(16, 16);
   dim3 numBlocks(N /threadsPerBlock.x, N/ threadsPerBlock.y);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

CUDA



CUDA C

The fundamental condition is that blocks are required to be executable independently (executed in arbitrary order, simultaneously/parallel or sequential/serial)

Support for many other usable functions:

- `cudaMalloc(), cudaMallocPitch(), cudaMalloc3D()`
- `cudaFree()`
- `cudaMemcpy(), cudaMemcpy2D() , cudaMemcpy3D()`
- `dimBlock(), dimGrid()`
- `atd.`

Declarations:

__global__ void KernelFunc(...); //kernel function, runs on device

__device__ int GlobalVar; //variable in device memory

__shared__ int SharedVar; //variable in per-block shared memory

Special variables:

dim3 threadIdx; dim3 blockIdx; dim3 blockDim; dim3 gridDim;

Incrementing large array of elements – CUDA

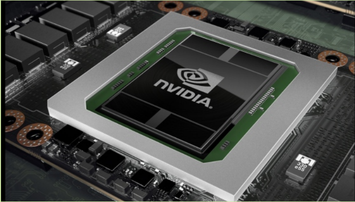
```
#include <stdio.h>
__global__ void increment(int* out, int* in) {
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  out[idx] = in[idx]+1;
}

int main (int argc, char** argv)
{
  int* num_h;                  // pointer to array
  int* num_d;                  // pointer to array in global memory
  int* num_out_d;              // pointer to output array in global memory
  size_t num_size = 128*512;   // size of array (too high for single block of threads)
  int num_threads_per_block = 128;                  // number of threads in one block
  int num_blocks = num_size/num_threads_per_block;  // lattice size
  size_t num_size_bytes = sizeof (int)*num_size;    // array size in bytes

  num_h = (int*)malloc (num_size_bytes);
  cudaMalloc ((void**) &num_d, num_size_bytes);     // global memory allocation
  cudaMalloc ((void**) &num_out_d, num_size_bytes); // global memory allocation

  for (unsigned int i = 0; i < num_size; i++) {
    num_h[i] = i;
  }

  cudaMemcpy (num_d, num_h, num_size_bytes, cudaMemcpyHostToDevice);
  increment<<<num_blocks, num_threads_per_block>>> (num_out_d, num_d);
  cudaThreadSynchronize();
  cudaMemcpy (num_h, num_out_d, num_size_bytes, cudaMemcpyDeviceToHost);

  cudaFree(num_d);  cudaFree(num_out_d);  free(num_h);
  return 0;
}
```

CUDA Unified Memory

- Introduced 2014 with CUDA 6 and the Kepler
  cudaError_t cudaMallocManaged(void** ptr, size_t size);
- pre-Pascal GPUs (Tesla K80) allocates memory on the GPU
- Pascal, Volta, … pages can migrate to any processor's memory, populated with pagetables on demand
- cudaMemPrefetchAsync(ptr, length, destDevice, stream)
- cudaMemAdvise(ptr, length, advice, device)
  cudaMemAdviseSetReadMostly,
  cudaMemAdviseSetPreferredLocation,
  cudaMemAdviseSetAccessedBy
- Pascal and later NVLINK supports native atomics in hardware. PCI-e will have software-assisted atomics.

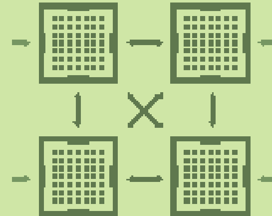
Nvidia Pascal based TESLA P100

Pascal Architecture



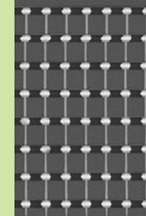
Highest Compute Performance

NVLink



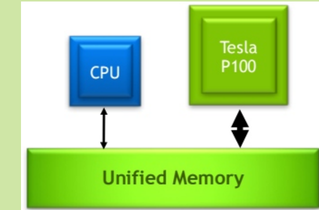
GPU Interconnect for Maximum Scalability

CoWoS HBM2



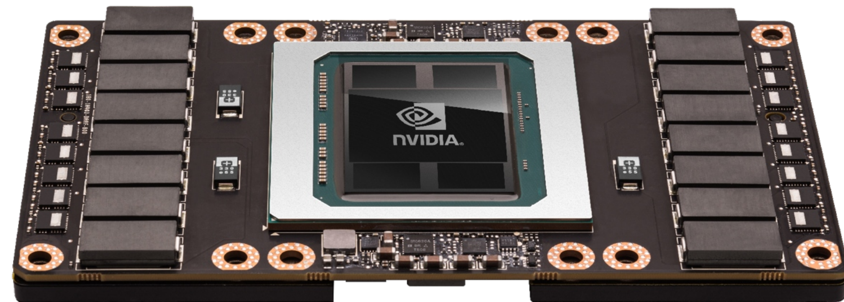
Unifying Compute & Memory in Single Package

Page Migration Engine



Simple Parallel Programming with Virtually Unlimited Memory Space

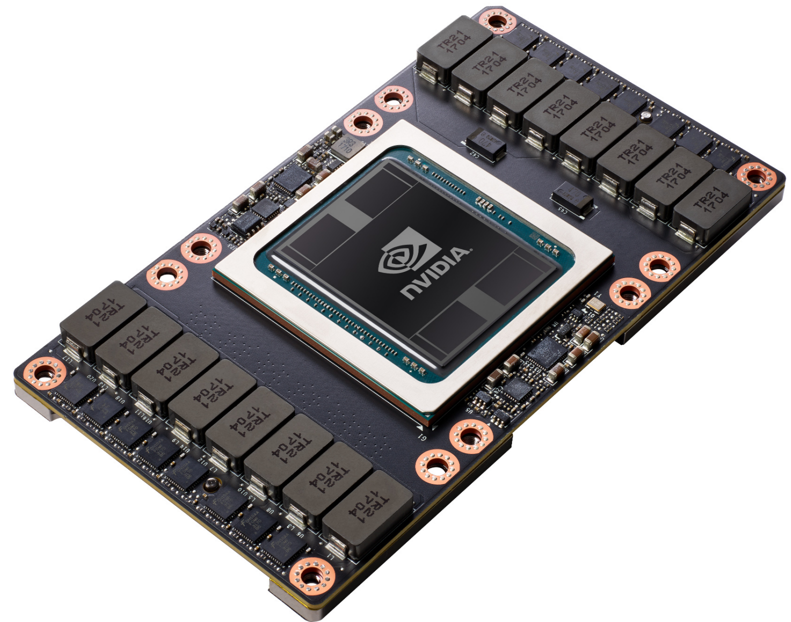
- 3584 CUDA cores
- 4.7 FP64 TFLOPS
  9.3 FP32 TFLOPS
- 160 GB/s NVLink



Source: Tesla Volta / DGX-1v by Ralph Hinsche

Nvidia Volta Based TESLA V100

- 5,120 CUDA cores
- 640 NEW Tensor cores
- 7.5 FP64 TFLOPS
  15 FP32 TFLOPS
- 120 Tensor TFLOPS
- 20MB SM RF 16MB Cache
  16GB HBM2 @ 900 GB/s
- 300 GB/s NVLink



Source: Tesla Volta / DGX-1v by Ralph Hinsche

OpenCL

- Is CUDA C the only possibility to accelerate computations?
- OpenCL – The open standard for parallel programming of heterogeneous systems

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

OpenCL:

```
kernel void VecAdd(global const float* A, global const
   float* B, global const float* C)
{
   int i= get_global_id(0);
   C[i] = A[i] + B[i];
}
```

Jacket

- Matlab support

```
A = gdouble(B); % to push B to the GPU from the CPU
B = double(A);  % to pull A from the GPU back to the CPU
```

```
X = gdouble( magic( 3 ) );
Y = gones( 3, 'double' );
A = X * Y
```

```
GPU_matrix = gdouble( CPU_matrix );
GPU_matrix = fft( GPU_matrix );
CPU_matrix = double( GPU_matrix );
```

Goose

Controlled by compiler directives

```
#pragma goose parallel for loopcounter(i, j)
for (i = 0; i < ni; i++)
   for (j = 0; j < nj; j++)
      for (k = 0; k < 3; k++)
         C[k] = A[j][k] - B[i][k];
```

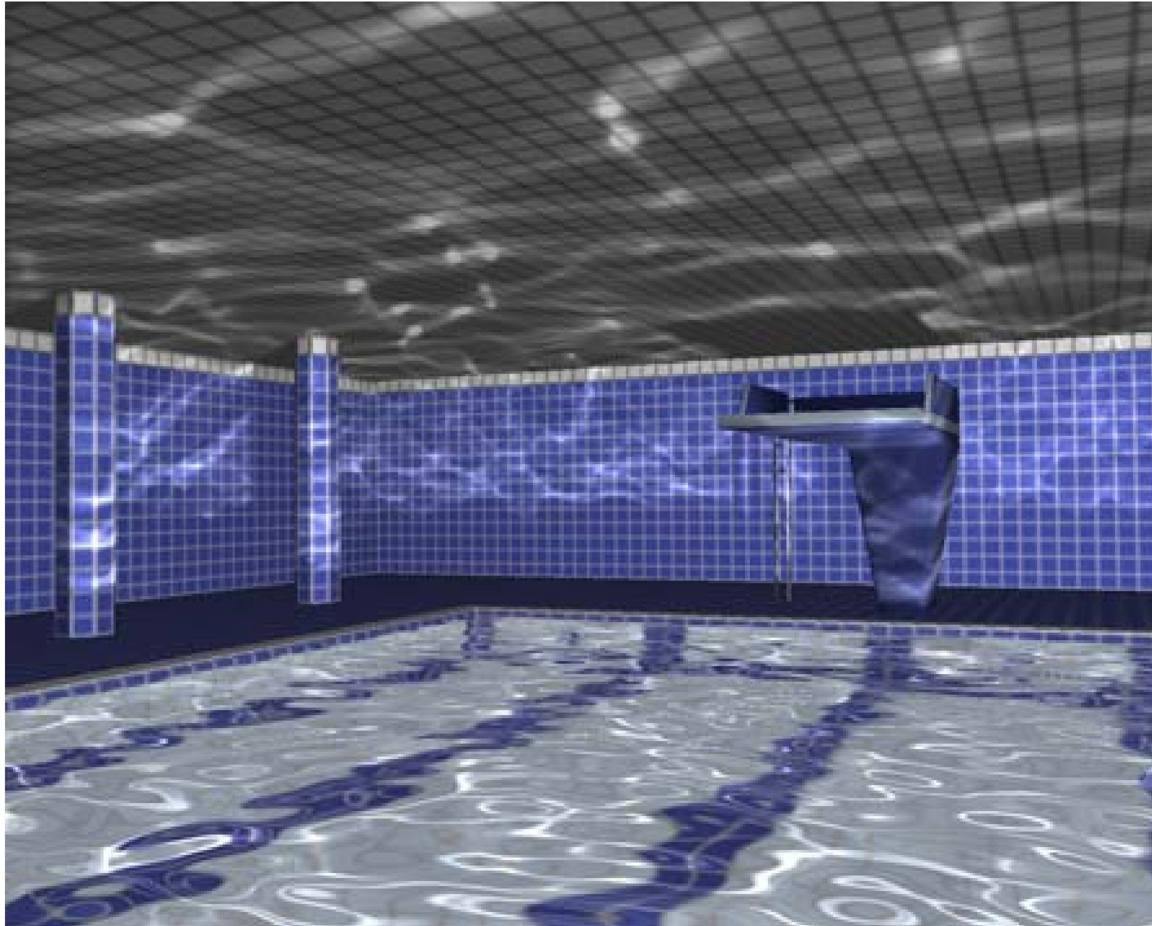
Also other frameworks available:

- PGI Accelerator
- CAPS HMPP
- Ct from Intelu
- **Brook** stream programming language (Stanford University)
- Support: Java, Python, C++, .NET, Mathematica

Is today support ideal?

- Control flow – instructions are executed in SIMD manner across all threads of one warp. Divergent branch results in creation of two separate thread groups which are executed sequentially. Explicit synchronization point (reconvergence point) can help to increase throughput.
- Memory – „intensity“ of memory accesses (mainly global)
- Data sharing – inter-threads communication
- It is necessary to consider the time spent on the efforts made to achieve maximum throughput (optimization) vs. time obtained by optimizing itself …

Applications

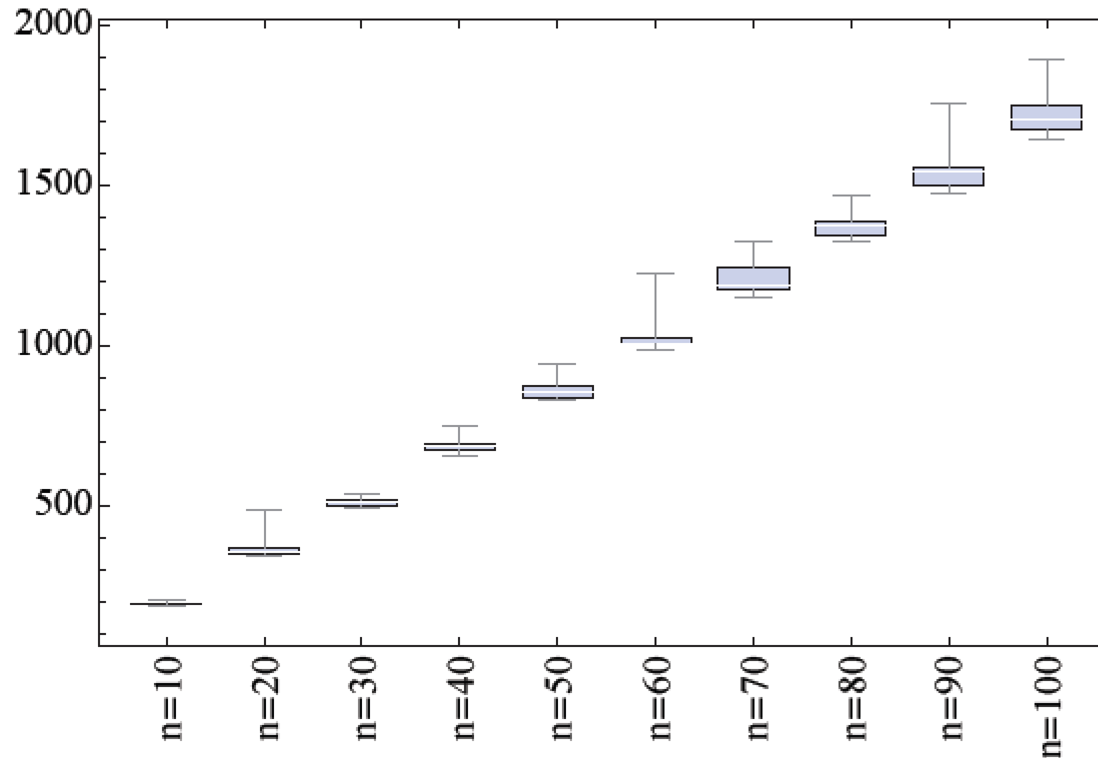


KRÜGER J., BÜRGER K., WESTERMANN R.: Interactive screen-space accurate photon tracing on GPUs. In *Eurographics Symposium on Rendering (June* 2006), pp. 319–329.

Applications

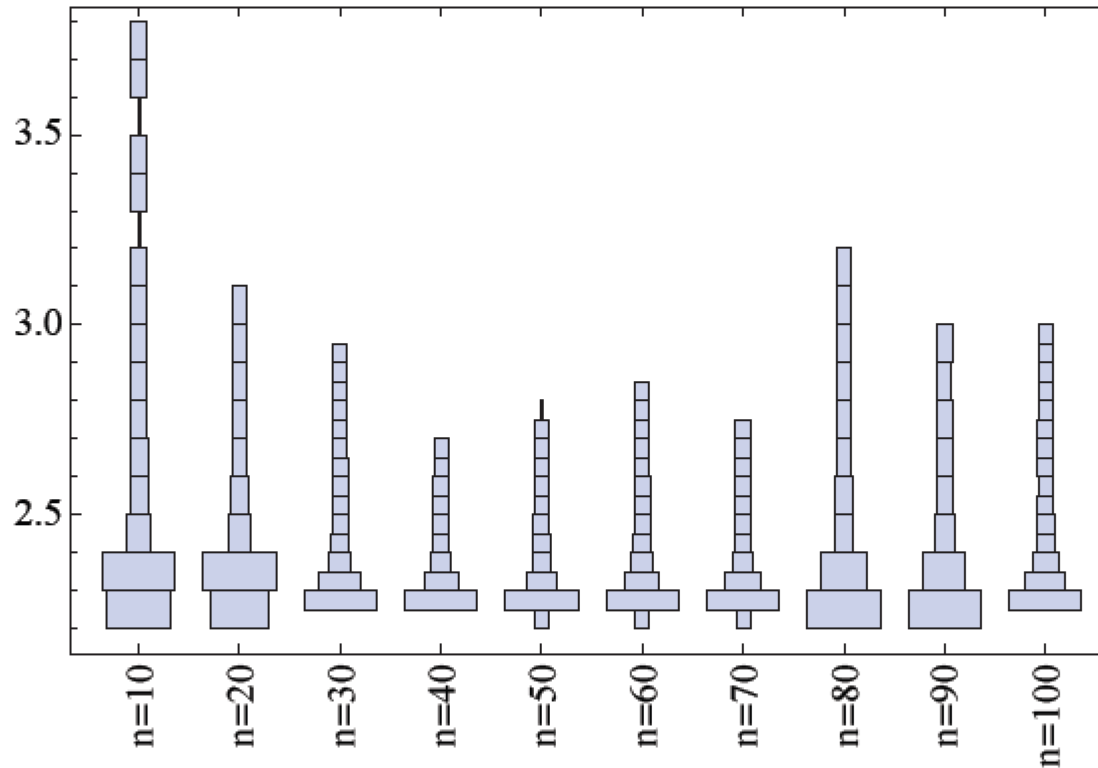
- linear algebra
- basic and partial differential equations (head conduction, fluids flow, stress of mechanical structures, vibrations,..)
- signal processing,
- images processing,
- analysis of chemical compounds, drug search
- evolutionary and genetic algorithms
- optimizations
- neural networks
- drug research
- …

Neural network on CPU



Source of data: Zdeněk Buk

Neural network on GPU – CUDA



Source of data: Zdeněk Buk

References and Links

- Computer Organization and Design MIPS Edition, 6th Edition, The Hardware/Software Interface by David Patterson John Hennessy ISBN: 9780128201091, Morgan Kaufmann, 2020
  - Online appendix C: Graphics and Computing GPUs, John Nickolls, Director of Architecture, NVIDIA, David Kirk, Chief Scientist, NVIDIA
- OpenGL ES 3.0, Programming Guide, Second Edition, Dan Ginsburg, Budirijanto Purnomo
- Linux Graphics Drivers: an Introduction, Version 3, Stéphane Marchesin, 2012
- Imagination University Programme, Introduction to mobile graphics architectures https://vimeo.com/user128660478