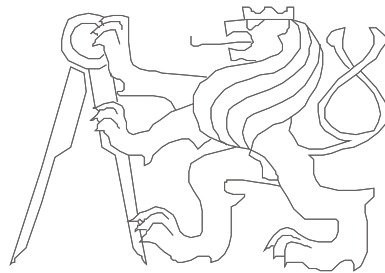


# Pokročilé architektury počítačů

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



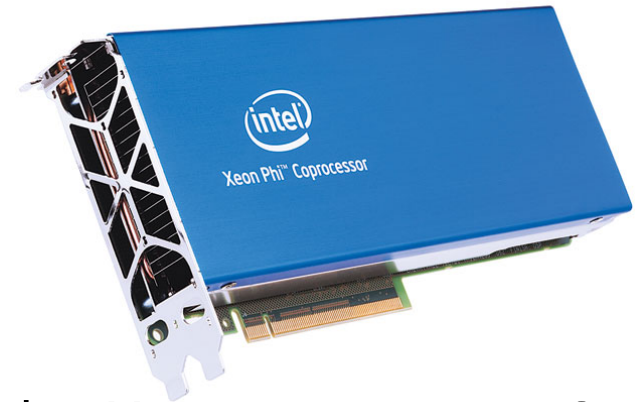
České vysoké učení technické, Fakulta elektrotechnická  
Autor materiálu: Michal Štepanovský

# Motivace

- **Tianhe-1A** - Chinese Academy of Sciences' Institute of Process Engineering (CAS-IPE)
- Molekulární simulace 110 miliard atomů (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 jáder / 12 vláken)
- „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/>  
z čehož plyne: 633 GFlop/kWatt (K Computer - 830 GFlop/kWatt)
- Používá vlastní propojovací síť: [Arch](#), 160 Gbps
- V Číně jsou nyní tři superpočítače využívající grafické karty, Tianhe-1 (AMD Radeony HD 4870 X2), Nebulae (nVidia Tesly C2050) a Tianhe-1A
- <http://i.top500.org/system/176929>
- <http://en.wikipedia.org/wiki/Tianhe-1>

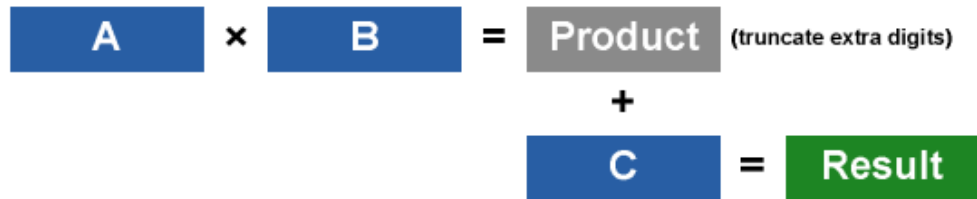
## Tianhe-2

- 33.86 Pflops
- spotřeba 17 MW
- Kylin Linux
- 16,000 uzlů, každý obsahující 2 Intel Ivy Bridge Xeon processory a 3 Intel Xeon Phi koprocesory (61 jader) = 32000 CPU a 48000 koprocesorů, spolu 3 120 000 jader
- 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) – místo Open-MP, CUDA, OpenACC, nebo OpenCL

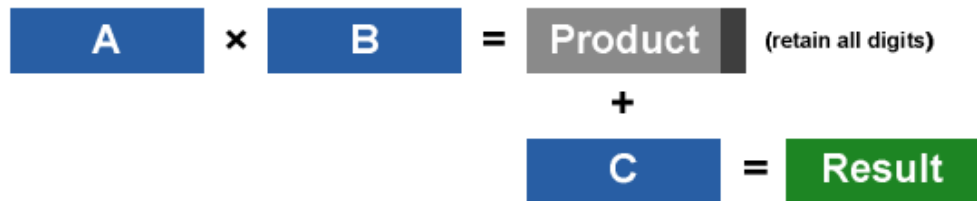


# Přesnější výsledky

## Multiply-Add (MAD):



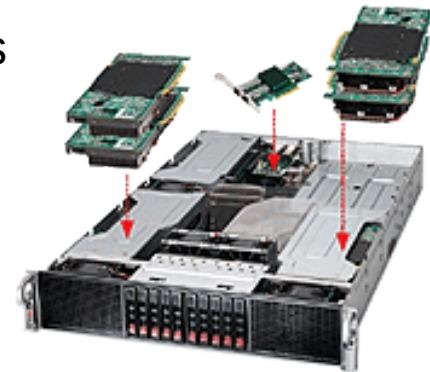
## Fused Multiply-Add (FMA)



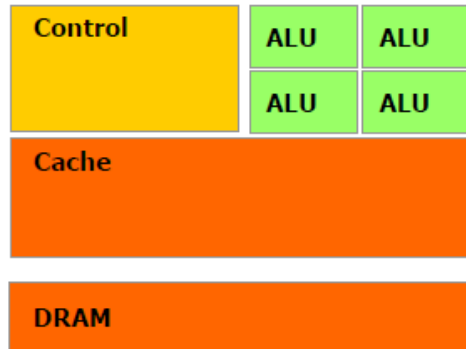
- **API** (Application Programming Interface): OpenGL, DirectX – z tohoto pohledu je GPU prakticky koprocесorem CPU

# Motivace

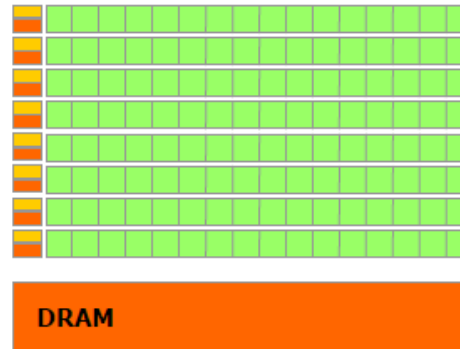
- **Estonia Donates Project:** Our [GPGPU](http://estoniadonates.wordpress.com/our-supercomputer) supercomputer is GPU-based massively parallel machine, employing more than thousand parallel streaming processors. Using GPU-s is very new technology, very price- and cost-effective compared to old CPU solutions. **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](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/>



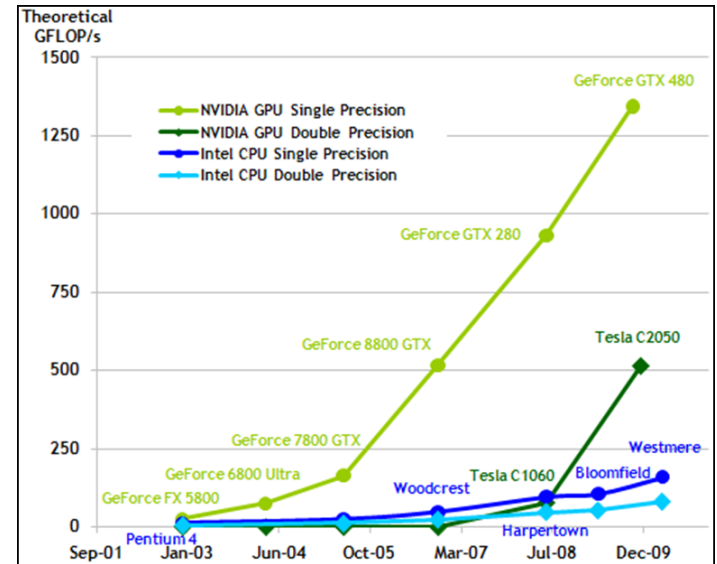
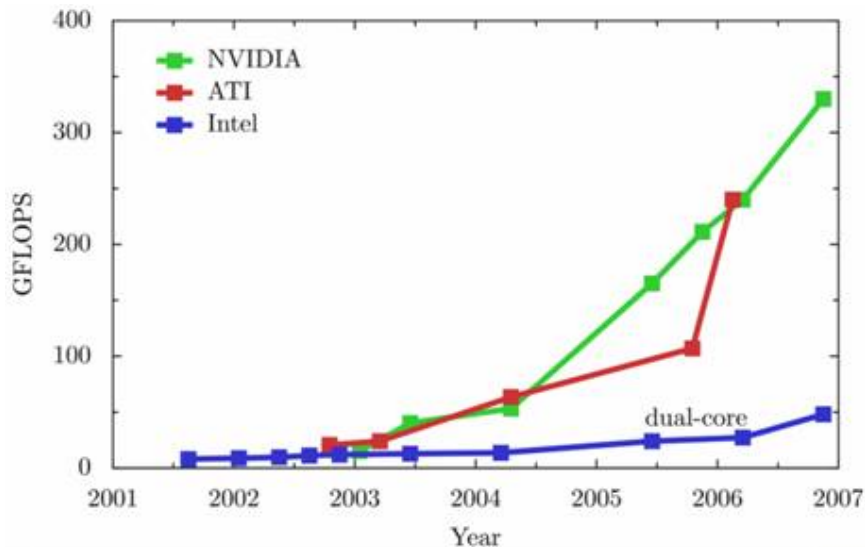
# CPU vs. GPU



CPU



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.“ Zdroj: [www.nvidia.com](http://www.nvidia.com)

## Výkonové metriky – Pamatujete si?

Nechť množina  $\{ R_i \}$  jsou vykonávací rychlosti programů  $i = 1, 2, \dots, m$  měřeny v MIPS (MFLOPS), resp. IPS (FLOPS)

- Střední aritmetický výkon: 
$$R_a = \sum_{i=1}^m \frac{R_i}{m} = \frac{1}{m} \sum_{i=1}^m R_i$$

$R_a$  je rovnoměrně váhován ( $1/m$ ) ve všech programech a je úměrný součtu IPC, avšak ne součtu vykonávacích časů (nepřímo úměrně). Proto střední aritmetický výkon selhává...

$$R_a = \frac{1}{2}(R_1 + R_2) = \frac{1}{2} \left( \frac{IC_1}{T_1} + \frac{IC_2}{T_2} \right) = \frac{1}{2} \left( \frac{IC_1}{IC_1 \cdot CPI_1 T_{CLK}} + \frac{IC_2}{IC_2 \cdot CPI_2 T_{CLK}} \right) =$$
$$\frac{1}{T_{CLK}} \left( \frac{IPC_1 + IPC_2}{2} \right) = \frac{1}{T_{CLK}} \left( \frac{IC_1}{2C_1} + \frac{IC_2}{2C_2} \right) \quad \text{avšak} \quad IPC_{1,2} = \frac{IC_1 + IC_2}{C_1 + C_2}$$

Pokud však  $C_1 = C_2$  (stejný celkový počet cyklů; tj. při téže frekvenci oba programy běží stejně dlouho) je  $R_a$  použitelný

## Výkonové metriky – Pamatujete si?

- Střední geometrický výkon: 
$$R_g = \prod_{i=1}^m R_i^{\frac{1}{m}}$$

Nesumarizuje reálný výkon, nemá inverzní relaci k celkovému času.  
Pro porovnávání s normalizovanými údaji vzhledem na referenční stroj.

- Střední harmonický výkon: 
$$R_h = \frac{m}{\sum_{i=1}^m \frac{1}{R_i}}$$

$$R_h = \frac{2}{\frac{1}{R_1} + \frac{1}{R_2}} = \dots = \frac{1}{T_{CLK}} \left( \frac{2}{CPI_1 + CPI_2} \right) = \frac{1}{T_{CLK}} \frac{2IC_1IC_2}{C_1IC_2 + C_2IC_1}$$

Pokud však  $IC_1 = IC_2$  (oba programy jsou stejně velké) je  $R_h$  použitelný

- Existují taktéž vážené verze těchto výkonů...



# 3D grafická pipeline

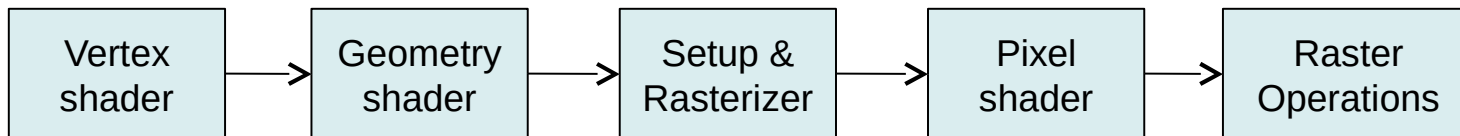
- jde o způsob zpracování obrazových dat k dosažení obrazu (vstupem je reprezentace 3D scény, výstupem 2D obraz)
- v zásadě se prochází těmito stupni:
  - transformace (škálování, rotace, translace,..) - maticový součin,
  - osvětlení (pouze vrcholy) – skalární součin vektorů,
  - pohledová transformace (do 3D souřadnic kamery) - maticový součin,
  - ořezání scény, rasterizace a texturování (odteď pixely)



- co je pro nás důležité -> vyžaduje se HW podpora – vývoj GPU

# GPU

- Jak to bylo kdysi?
  - úzce specializovaný jednoúčelový HW dle principu 3D grafické pipeline:
  - vertex shader (manipulace s 3D modelem, osvětlení vrcholů),
  - geometry shader (přidá/odstraní vrcholy,..)
  - pixel shader (lépe: fragment shader) – (vstupem je výstup z rasterizace; určuje barvu „pixelu“ (fragmentu) - textura..)
  - ROP unit (vytvoření pixelu z pixelových fragmentů, optimalizuje obraz pro zobrazení) ROP – Raster OPerator, (ATI: Element Render Back-End)



# GPU

- Jak to je (a kam to směřuje) nyní?
  - funkce HW v každém stupni více flexibilní, programovatelnost (nejenom vlastní „program“, ale i podpora control-flow primitiv)
  - nyní podpora 16, 24, 32, 64 floating point precision
  - unifikace shaderů (každý je schopen plnit funkce ostatních..) - (ATI Xenos, GeForce 8800) – výhoda?
    - málo detailní scéna (vertex shader vs. pixel shader)
    - hodně detailní scéna (vertex shader vs. pixel shader)
- Co máme z toho /  
co z toho můžeme vytěžit?



# GPU

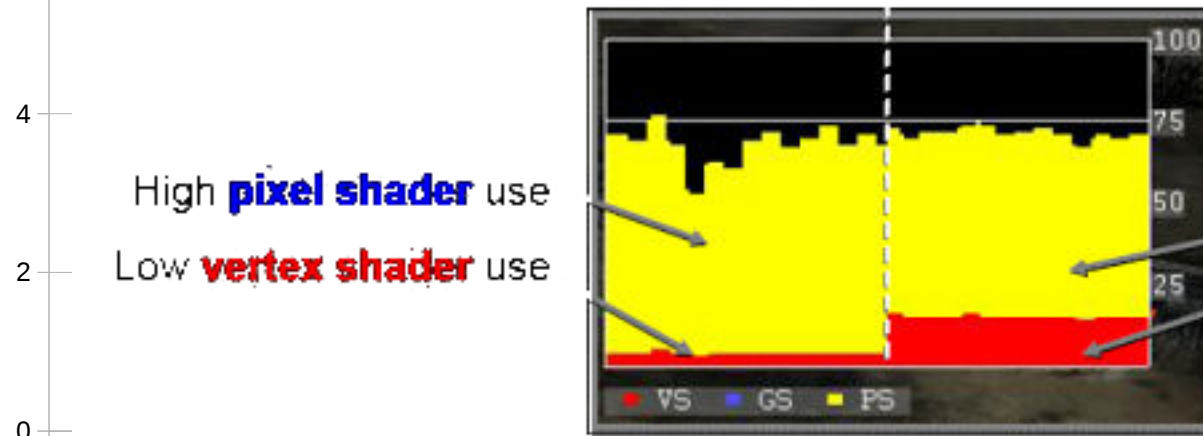


Less Geometry



More Geometry

column 1  
column 2  
column 3



High **pixel shader** use  
Low **vertex shader** use

Balanced use of  
**pixel shader** and  
**vertex shader**

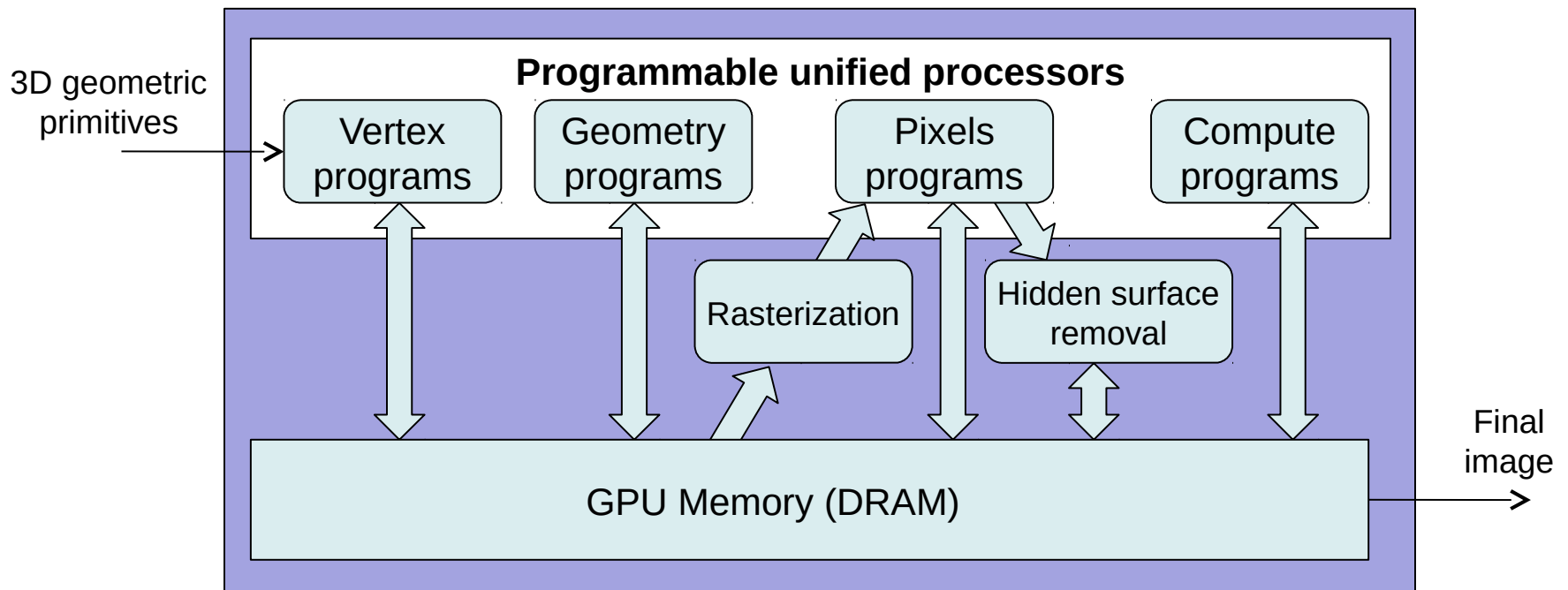
<http://techreport.com/articles.x/11211/3>

Row 3

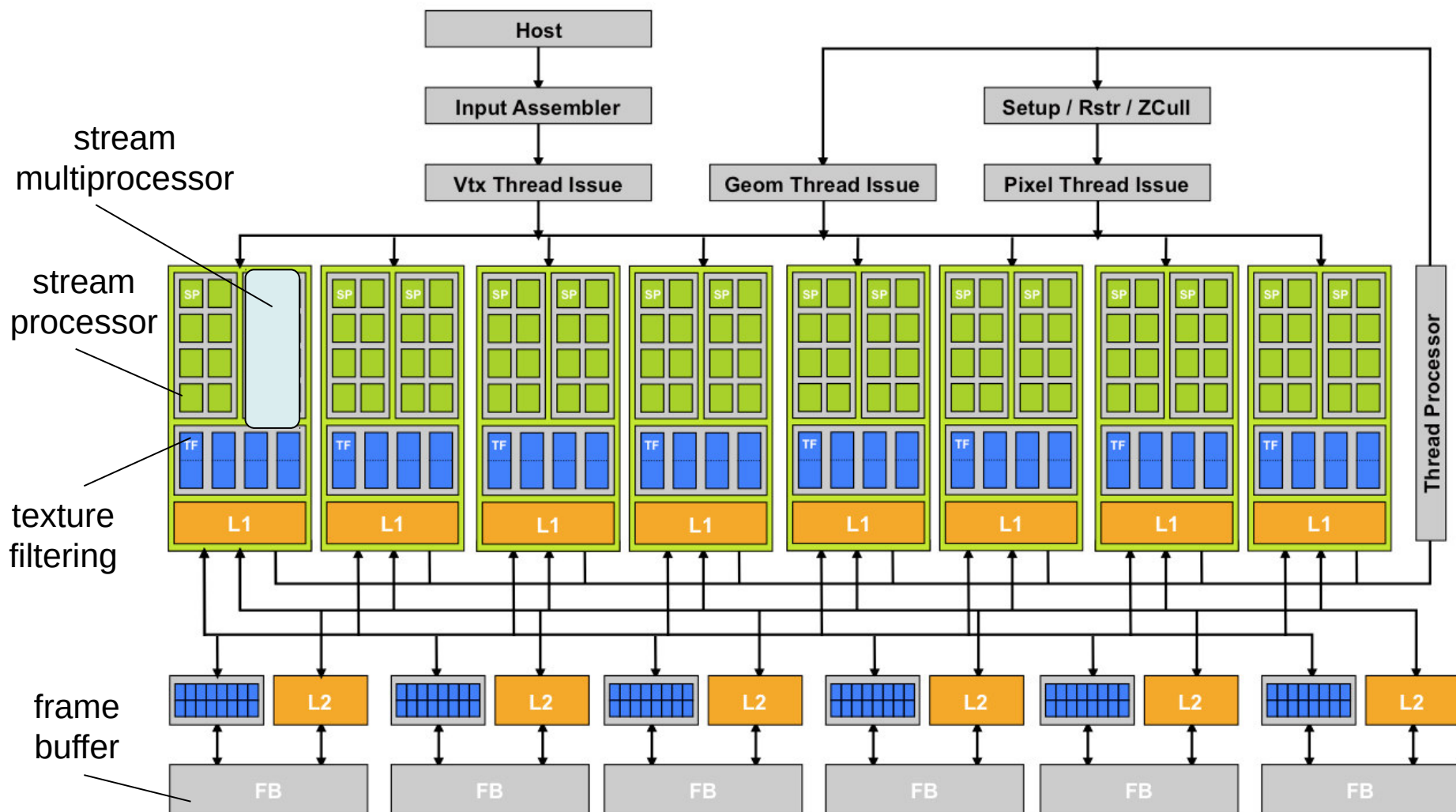
Row 4

# GPU

- Princip unifikace shaderů – architektura poskytuje jednu velkou množinu datově paralelních procesorů v plovoucí řádové čárce dostatečně obecných na to, aby mohli nahradit funkce jednotlivých shaderů

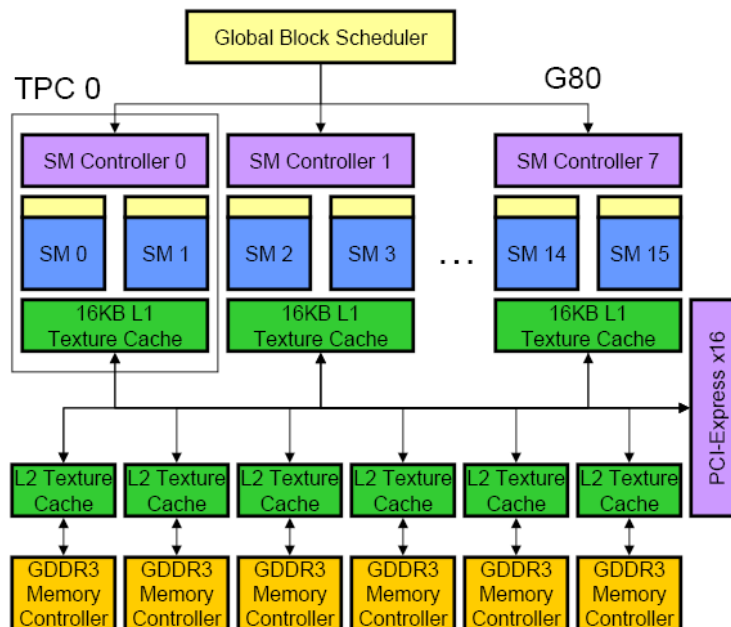


# GPU - GeForce 8800

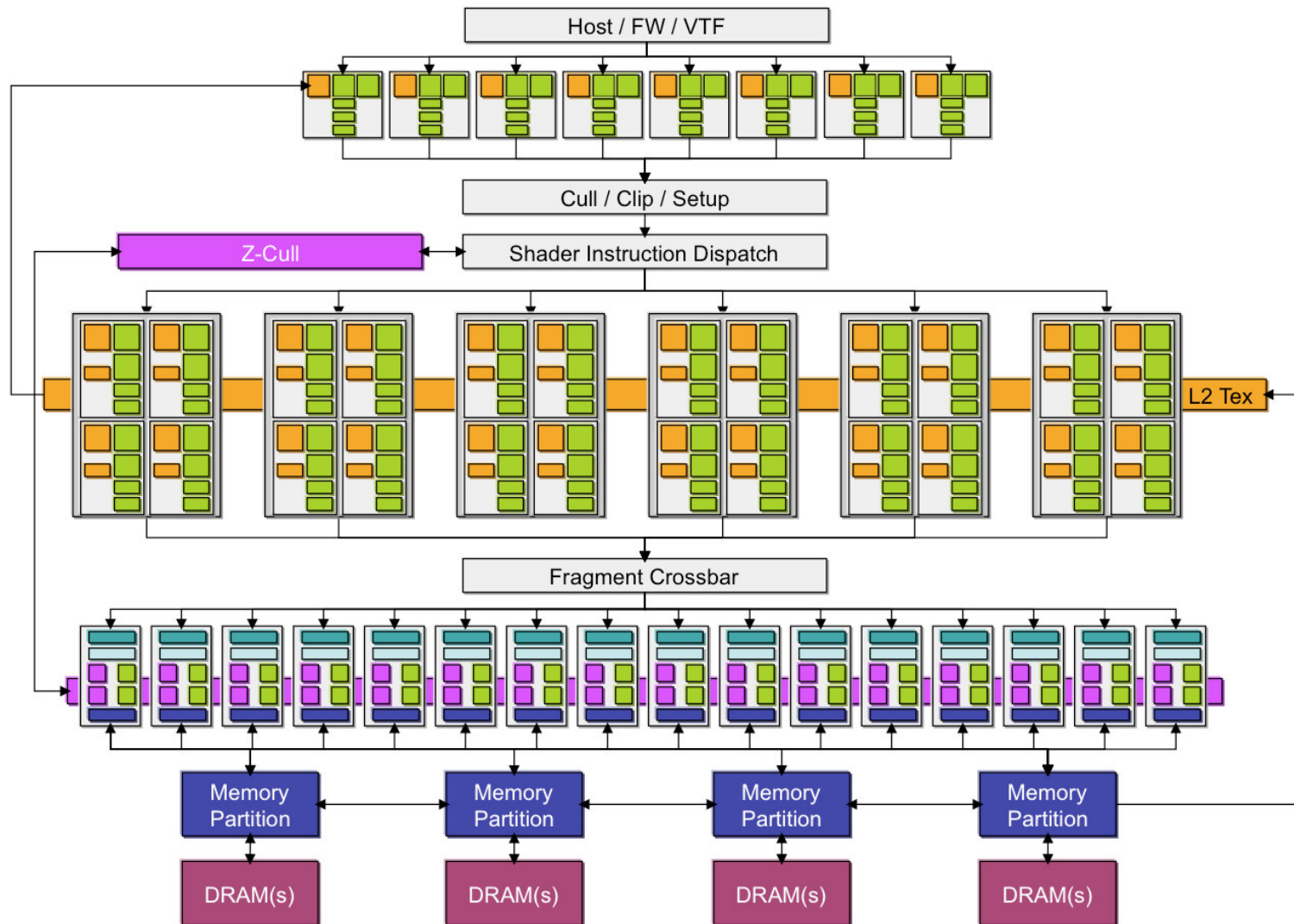


# GeForce 8800 - hardwarové omezení

- 512 vláken v jednom bloku
- 8 bloků na jeden SM
- 768 vláken na SM >  $768 \times 16 = 12\,288$  vláken celkově!
- 128 vláken současně
- 16 384 bytes sdílené cache na jeden SM
- dvě vlákna z různých bloků nemůžou spolupracovat



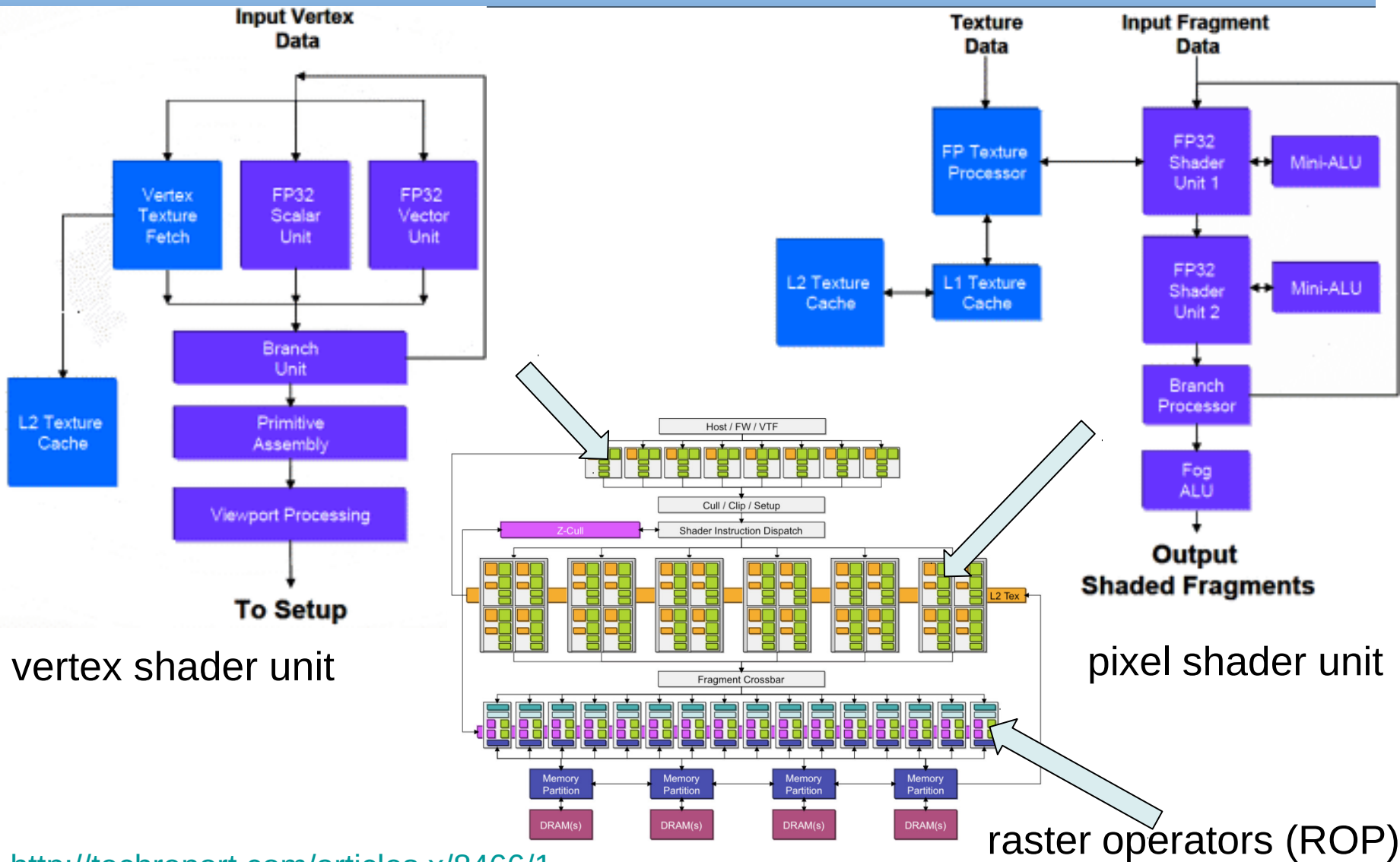
# GPU - GeForce 7800 - porovnání



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

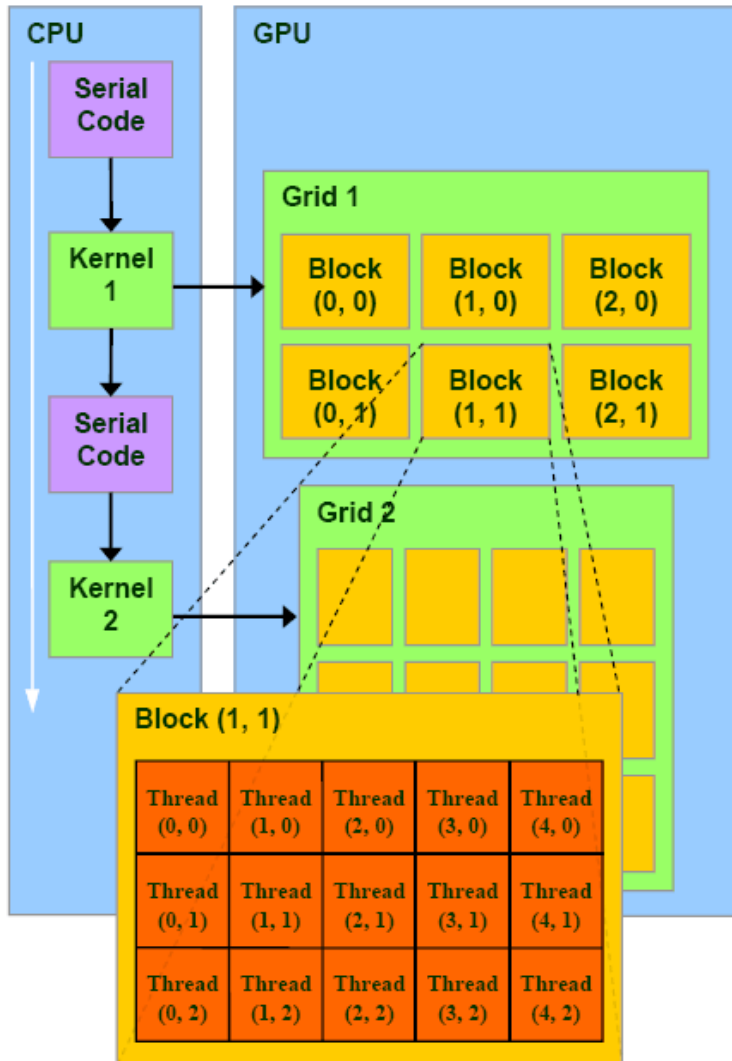


# GPU - GeForce 7800



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

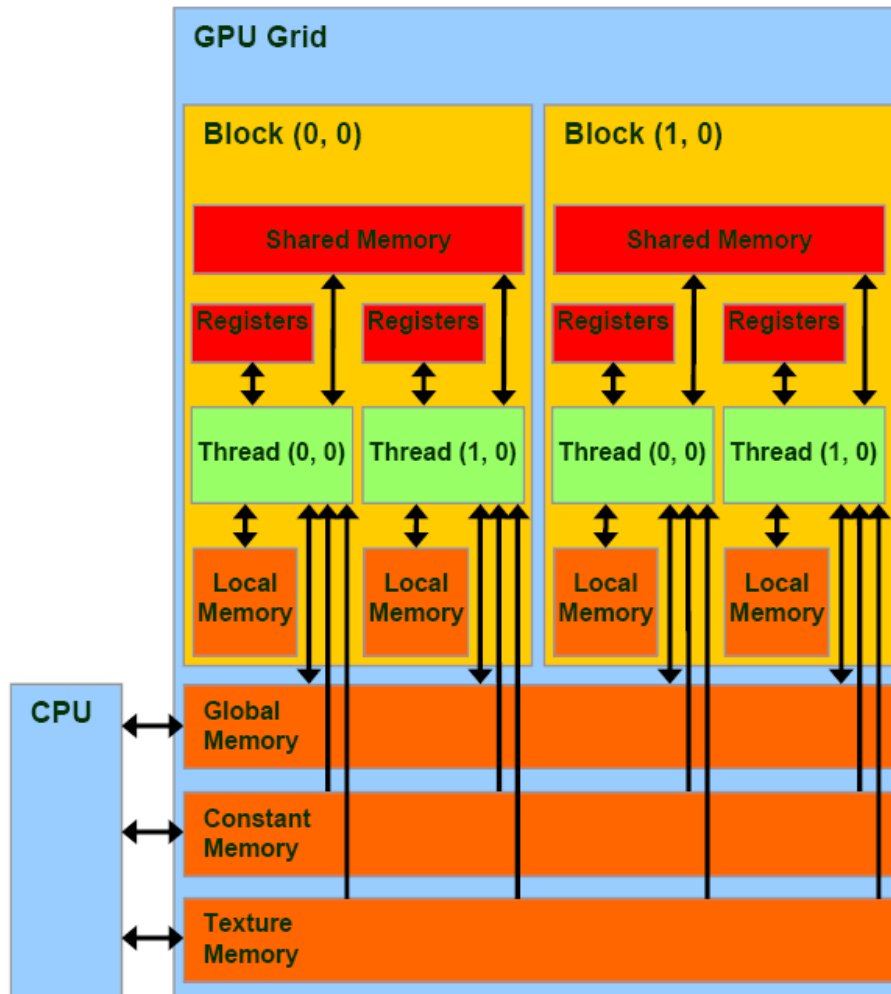
# CUDA (*Compute Unified Device Architecture*)



- Kernel – část aplikace běžící na GPU
- Kernel – vykonáván na Gridu
- Grid – mříž bloků vláken
- Blok vláken – skupina vláken začínající na té samé adrese a komunikující přes sdílenou paměť a synchronizační bariéry ( $\leq 512$ )
- Jeden blok jednomu procesoru (Streaming Multiprocessor - SM)
- Jedno vlákno uvnitř bloku jedné vykonávací jednotce (Streaming Processor core - SP core)

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

# CUDA (*Compute Unified Device Architecture*)



Paměťový model dle CUDA:

- Registry a sdílená paměť – na čipu
- Lokální paměť – frame buffer
- Constant Mem a Texture Mem – frame buffer, avšak jen pro čtení, kešovány na čipu, koherence?
- Globální paměť

červená = rychlá (na čipu)  
oranžová = pomalá (DRAM)

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

# CUDA C

- CUDA C je C s rozšířeními, ale i omezeními
- Kernel – uvedením `__global__`
- každé vlákno vykonávající kernel má své unikátní ID

```
// Definice funkce souctu vektoru:
    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()
{
    ...
    // Soucet vektoru delky N:
    VecAdd(N, A, B, C);
}
```

# CUDA C

- CUDA C je C s rozšířeními, ale i omezeními
- Kernel – uvedením `__global__`
- každé vlákno vykonávající kernel má své unikátní ID

```
// Definice Kernelu
__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()
{
    ...
    // Spuštění N vláken na GPU – součet vektoru delky N:
    VecAdd<<<1, N>>>(N, A, B, C);
}
```

<<<pocetbloku, pocetvlaken>>>

# CUDA C

- Z důvodu podpory nativní práce s vektory, 2D a 3D maticemi je proměnná *threadIdx* řešena jako 3-složkový vektor
- Pro 2D blok dimenze (Dx, Dy), má vlákno na pozici (x,y) své ID (x + y Dx)
- Pro 3D blok dimenze (Dx, Dy, Dz), má vlákno na pozici (x,y,z) své 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()  
{  
    ...  
    // Spuštění kernelu jako jeden blok N * N * 1 vláken  
    int numBlocks = 1;  
    dim3 threadsPerBlock(N, N);  
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);  
}
```

Počet vláken uvnitř bloku je limitován. Všechna vlákna uvnitř bloku jsou vykonávána tím samým procesorem (SM) a sdílejí omezené paměťové prostředky.

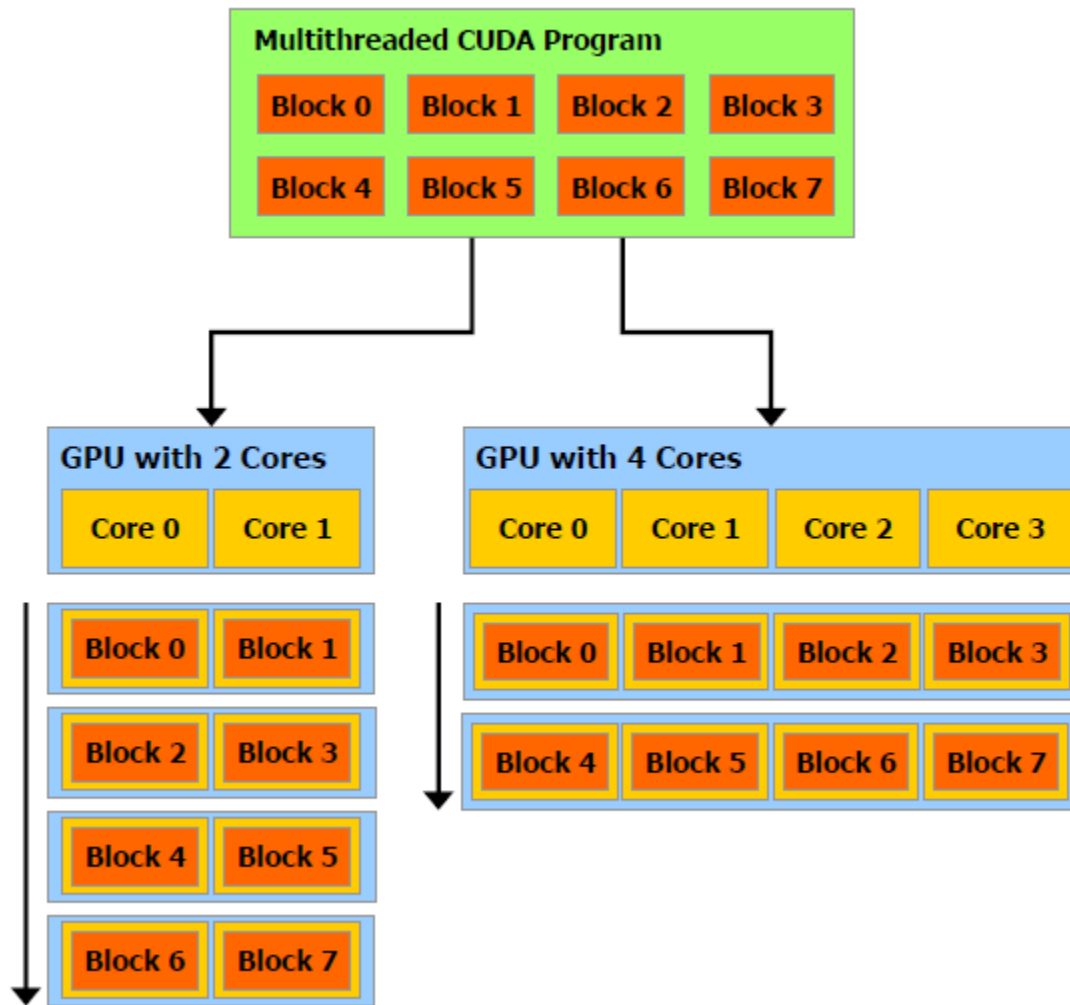
V současnosti jeden blok vláken může obsahovat 1024 vláken.

# 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

Podmínkou tedy je, že bloky musejí být vykonatelné nezávisle (vykonatelné v libovolném pořadí, paralelně nebo sekvenčně)

Podpora mnoha dalších užitečných funkcí:

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

Deklarace:

```
__global__ void KernelFunc(...); //kernel function, runs on device
__device__ int GlobalVar; //variable in device memory
__shared__ int SharedVar; //variable in per-block shared memory
```

Speciální proměnné:

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

# Inkrementace prvků pole - CUDA

```
#include <stdio.h>
__global__ void inkrementuj(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;           //ukazatel na pole
    int* num_d;           //ukazatel na pole v globalni pameti
    int* num_out_d;       //ukazatel na vystupni pole v globalni pameti
    size_t num_size = 128*512; //velikost pole
    int num_threads_per_block = 128;           //pocet vlaken na jeden blok
    int num_blocks = num_size/num_threads_per_block; //velikost mrizky
    size_t num_size_bytes = sizeof (int)*num_size; //velikost pole v bytech

    num_h = (int*)malloc (num_size_bytes);
    cudaMalloc ((void**) &num_d, num_size_bytes); //alokace v globalni pameti
    cudaMalloc ((void**) &num_out_d, num_size_bytes); //alokace v globalni pameti

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

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

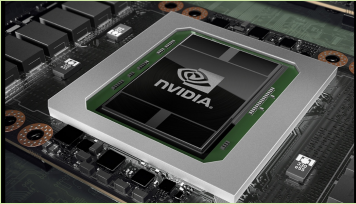
    cudaFree(num_d); cudaFree(num_out_d); free(num_h);
    return 0;
}
```

# 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

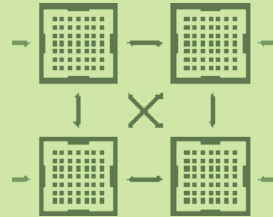
# Nvidia Pascal based TESLA P100

## Pascal Architecture



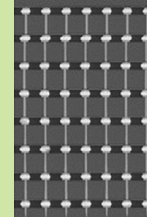
Highest Compute Performance

## NVLink



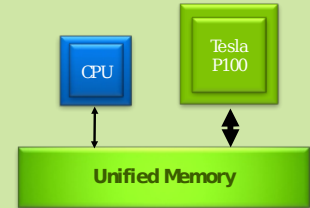
GPU Interconnect for Maximum Scalability

## CoWoS HBM2



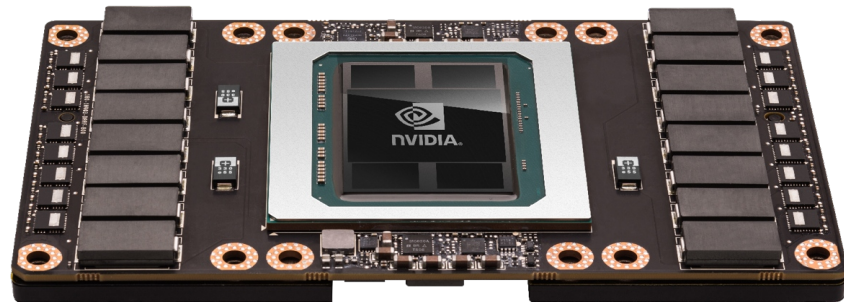
Unifying Compute & Memory in Single Package

## 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

- Je CUDA C jedinou možností?
- 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

- Podpora pro Matlab

```
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

## Označení pomocí direktiv

```
#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];
```

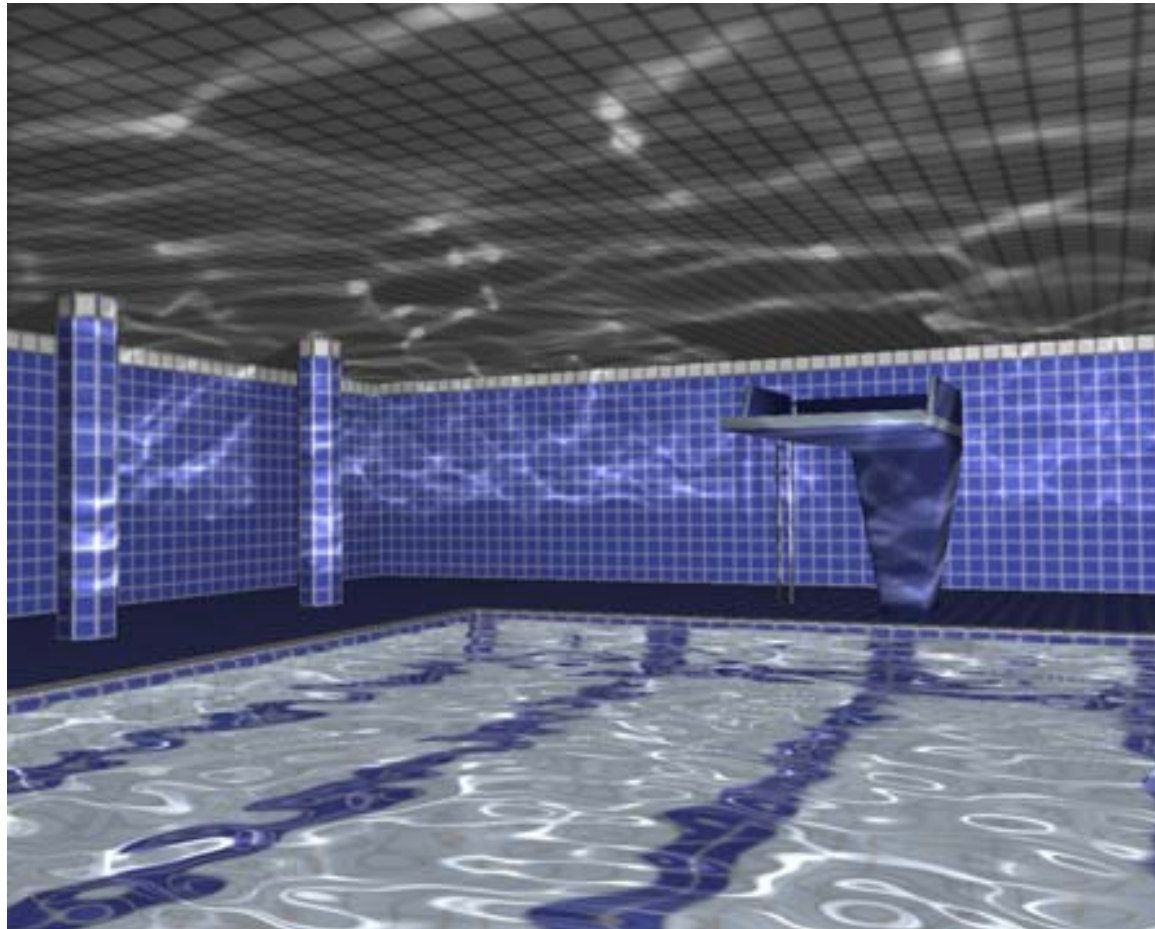
## Dále také:

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

## Je všechno tak ideální?

- Control flow – instrukce jsou vykonávány v režimu SIMD napříč všemi vlákny jednoho warpu. Divergentní větvení má za následek vytvoření dvou separátních skupin vláken, které jsou vykonány za sebou. Explicitní bod synchronizace (rekonvergence) může zvýšit propustnost.
- Paměť – „intenzita“ přístupu do paměti (zejména globální)
- sdílení dat – komunikace mezi vlákny
- je potřeba zvážit čas strávený úsilím vynaloženým pro dosažení maximální propustnosti (optimalizace) vs. čas získaný samotnou optimalizací...

# Aplikace



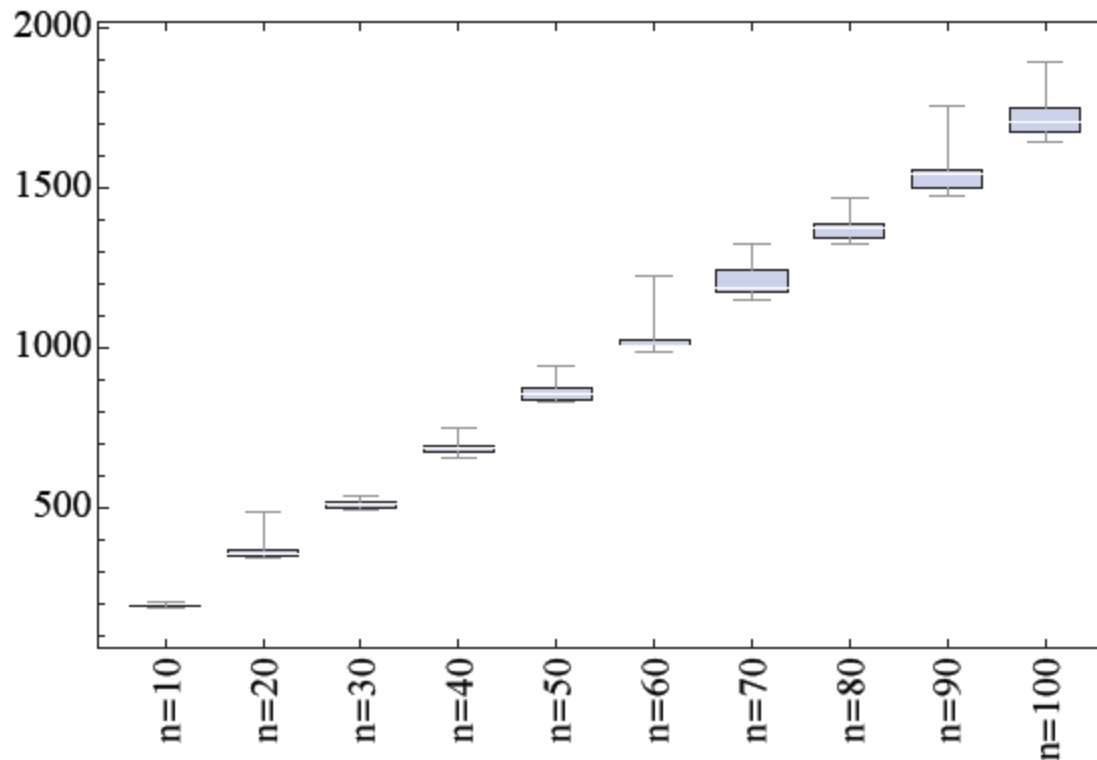
KRÜGER J., BÜRGER K., WESTERMANN R.: Interactive screen-space accurate photon tracing on GPUs. In *Eurographics Symposium on Rendering (June 2006)*, pp. 319–329.

# Aplikace

- lineární algebra
- obyčejné a parciální diferenciální rovnice (vedení tepla, proudění tekutin, namáhání mechanických konstrukcí, kmity,..)
- zpracování signálů,
- zpracování obrazu,
- analýza chemických sloučenin, hledání léčiv
- evoluční a genetické algoritmy
- optimalizace
- neuronové sítě
- ...

# Neuronová síť na CPU

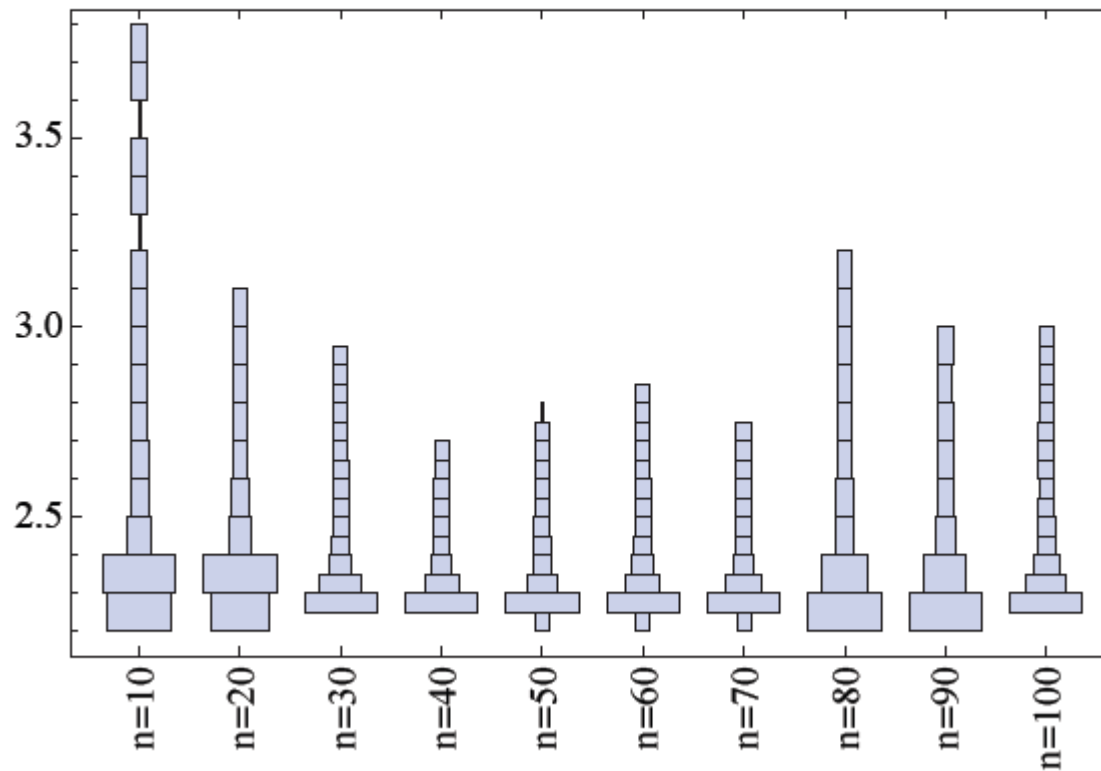
Time [ms] of evaluation of 100 networks with 'n' neurons  
for 50 time steps (average of 20 runs)  
(CPU implementation on 'HP server')



Poskytnul: Zdeněk Buk

# Neuronová síť na GPU - CUDA

Time [ms] of evaluation of 100 networks with 'n' neurons  
for 50 time steps (average of 1000 runs)  
(Client - 'MacBook Pro', Server - 'PC', 100Mbit Ethernet)



Poskytnul: Zdeněk Buk