Grafické akcelerátory pro obecné výpočty
AVS – Architektury výpočetních systémů
Týden 13, 2024/2025

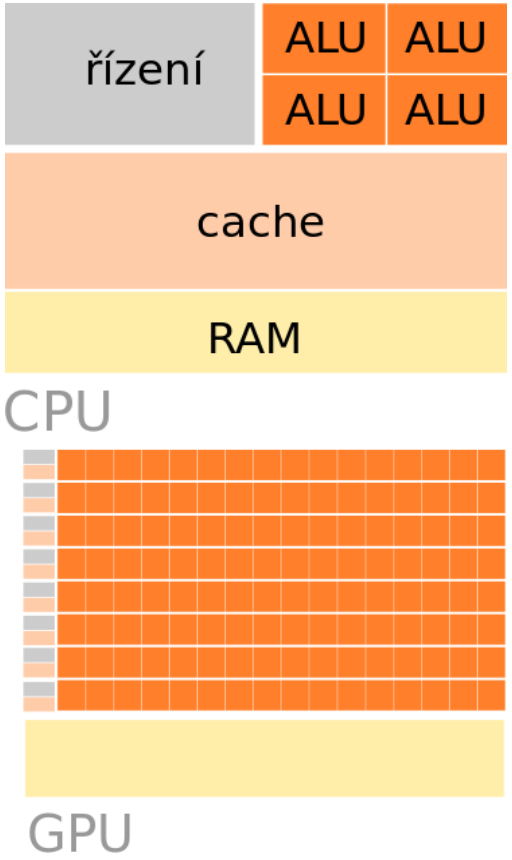
Jirka Jaroš

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole
jarosjir@fit.vutbr.cz

ARCHITEKTURA GRAFICKÝCH KARET

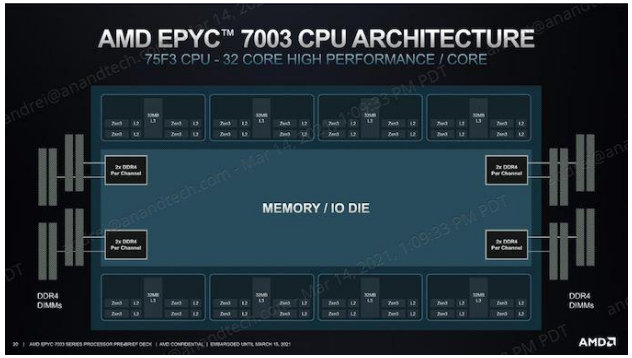
Porovnání CPU a GPU

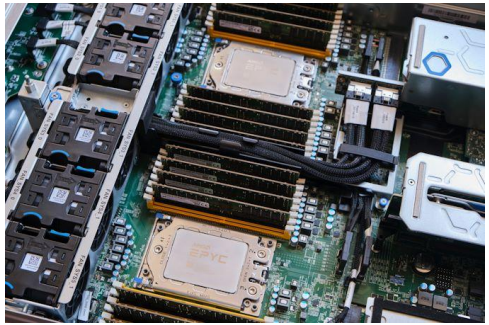
- **CPU**
  - Velmi rychlé a velké paměti cache
  - Dobře zpracované techniky predikce skoků
  - Vysoce výkonné zpracování sekvenčních programů
  - I/O, přerušení, virtuální paměť, izolace procesů
  - Vláknový a datový paralelismus
- **GPU**
  - Velké množství ALU – především MAD/FMA
  - Mnoho HW vláken – SMP
  - Rychlé lokální paměti
  - Paměti na desce s velkou propustností ale i latencí
  - Zpracování technikou SIMT (varianta SIMD)
  - In-order zpracování



Karolina - Výkon procesoru AMD EPYC 7763

- **64** fyzických jader doplněných o HyperThreading, 2,45 – 3,5GHz
- **SIMD** jednotky AVX2 (256b MAD), 32 op / takt v SP
- **256MB** L3 cache, 32MB L2 cache, 512 GB RAM





Teoretický výkon na procesor

- **2,7 DP TFLOPS**
- **5,3 SP TFLOPS**

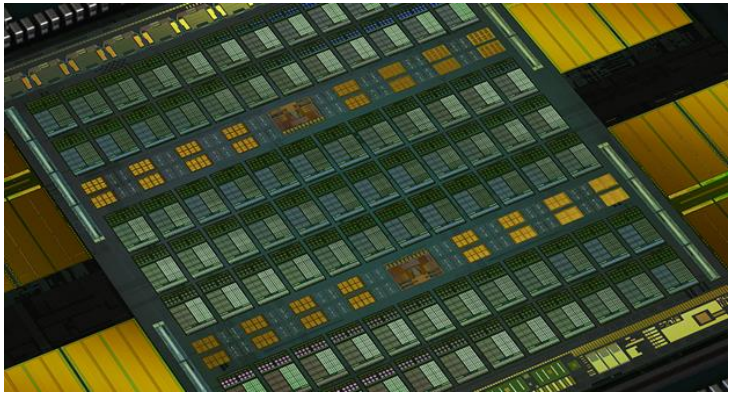
Propustnost do paměti

- **204** GB/s
- šířka sběrnice 512b

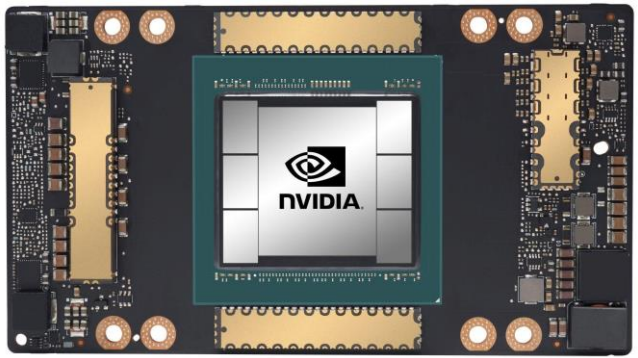
- https://www.anandtech.com/show/16529/amd-epyc-milan-review
- https://docs.it4i.cz/karolina/compute-nodes/

Karolina - Výkon GPU Nvidia Tesla A100

- **Obsahuje**
  - 108 SM procesorů, 6192 CUDA jader
  - 40 MB L2 cache
  - 40 GB HBMA RAM
  - Připojení na NV Link (600GB/s přes 4 GPU)
  - Max frekvence 1.4GHz

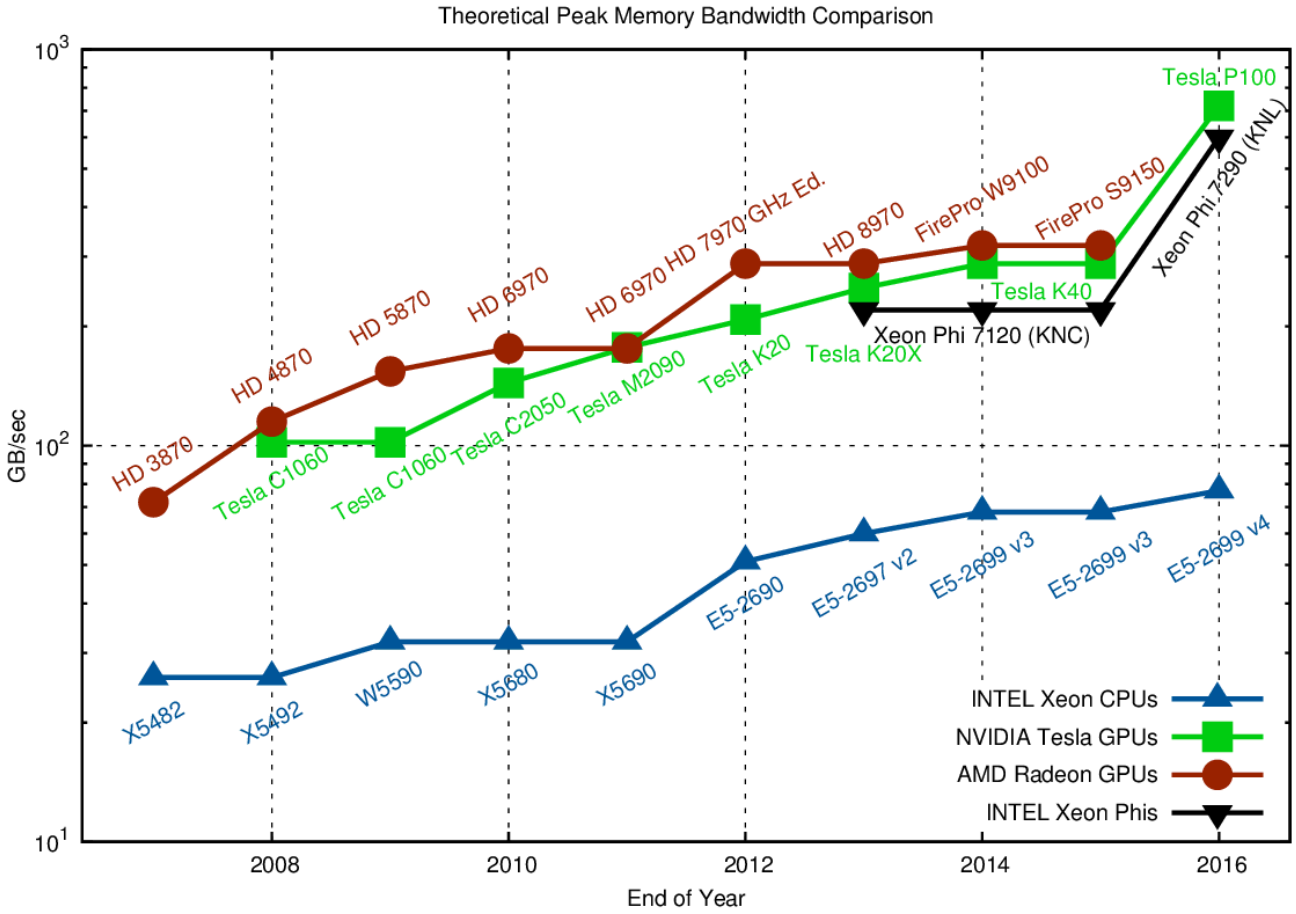


- **Teoretický výkon**:
  - **9,7 DP** **T**FLOPS (FP 64)
  - **19,5 SP** **T**FLOPS (FP 32)
  - **312 Tensor** **T**FLOPS (FP 16)
  - **1555 GB/s** – šířka sběrnice 5192b



- https://www.anandtech.com/show/15801/nvidia-announces-ampere-architecture-and-a100-products

Porovnání teoretické propustnosti paměti



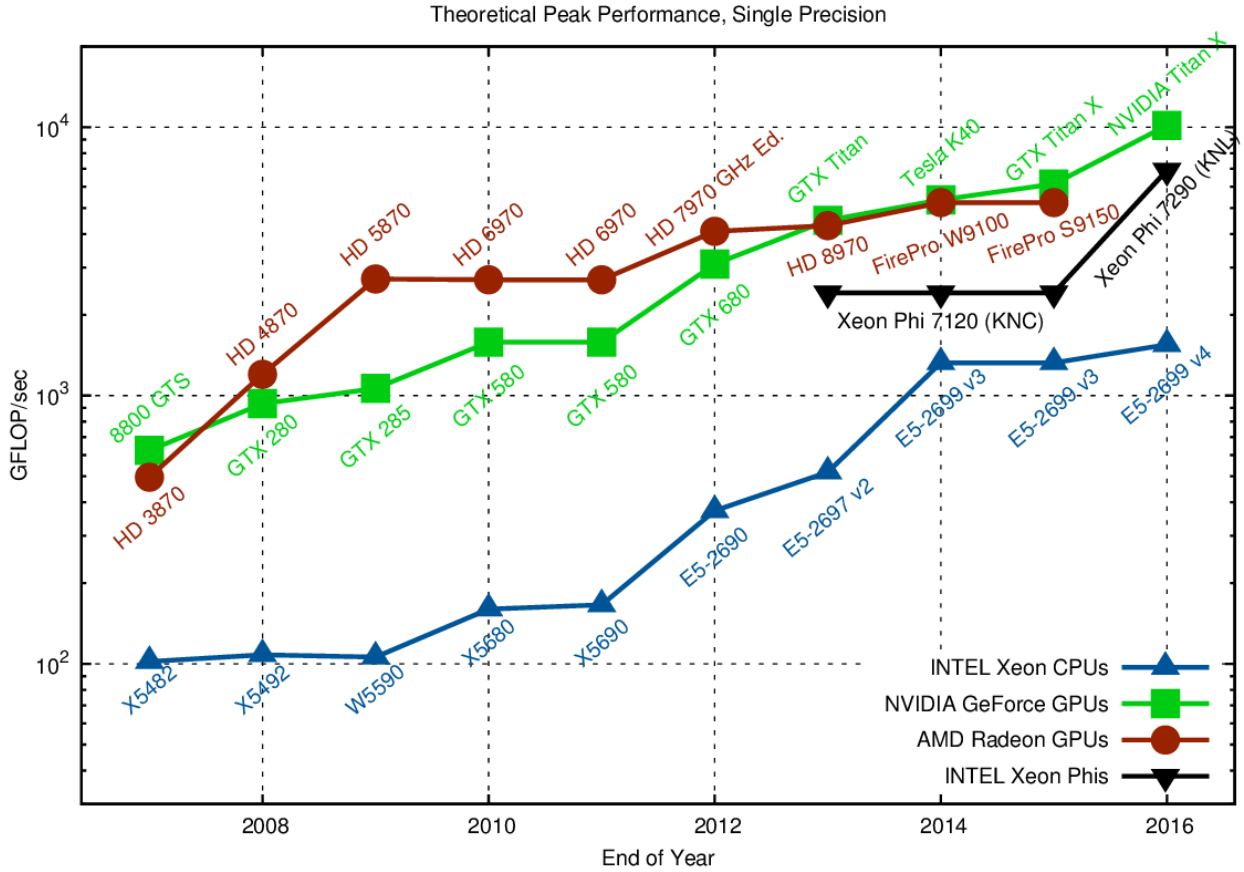
Barbora

Arch.	GB/s
V100	900
Intel 6240	141
Zrychlení	**6,38x**

Karolina

Arch.	GB/s
A100	1555
EPYC 7763	204
Zrychlení	**7,62x**

Porovnání teoretického výpočetního výkonu



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Barbora

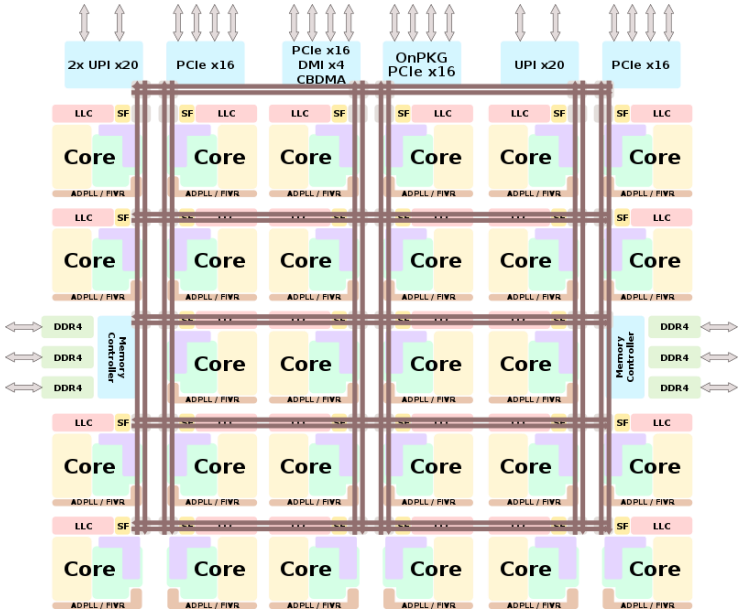
Arch.	GFLOPS
V100	15 700
Intel 6240	3 000
Zrychlení	**5,1x**

Karolina

Arch.	GFLOPS
A100	19 500
EPYC 7763	5 300
Zrychlení	**3,6x**

Barbora – Výkon procesoru Cascade Lake 6240

- **18** fyzických jader doplněných o HyperThreading, 2,6-3,9 GHz
- **SIMD** jednotky AVX-512 (512b MAD), 64 op/takt v SP
- **24,75 MB** L3 cache, 18 MB L2 cache, 192 GB RAM



Teoretický výkon na procesor

- **1,5 DP** TFLOPS
- **3,0 SP** TFLOPS

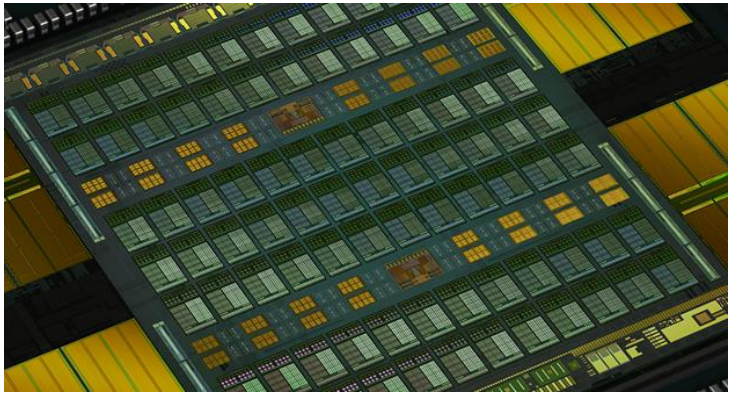
Propustnost do paměti

- **141** GB/s
- šířka sběrnice 384 b

- https://www.anandtech.com/show/15039/the-intel-core-i9-10980xe-review
- https://docs.it4i.cz/barbora/compute-nodes/

Barbora – Výkon GPU Nvidia Tesla V100

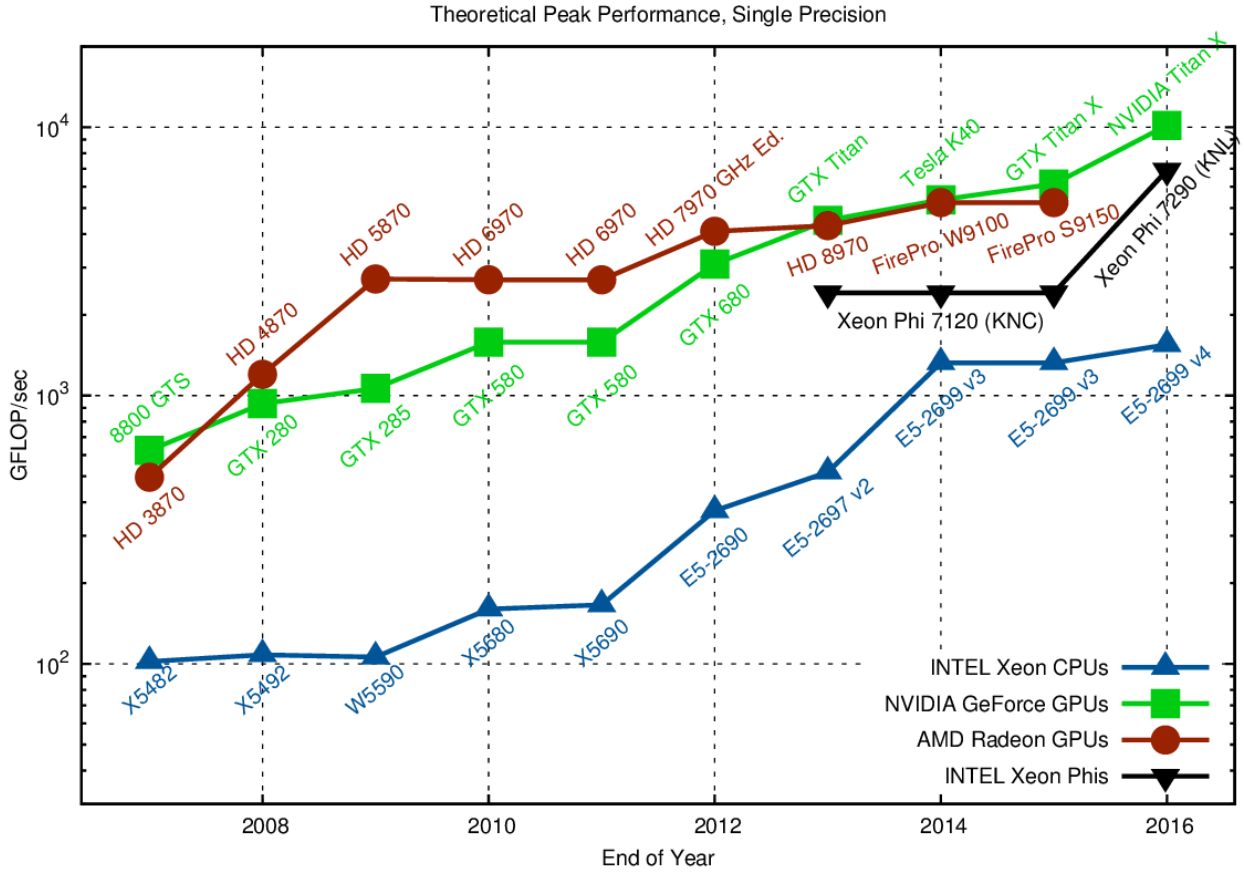
- **Obsahuje**
  - 80 SM procesorů, 5120 CUDA jader
  - 6144 kB L2 cache
  - 16 GB HBMA RAM
  - Připojení na NVLink (300 GB/s přes 4 GPU)
  - Max frekvence 1,5 GHz



- **Teoretický výkon**:
  - **7,8 DP** **T**FLOPS (FP 64)
  - **15,7 SP** **T**FLOPS (FP 32)
  - **125 Tensor** **T**FLOPS (FP 16)
  - **900 GB/s** – šířka sběrnice 4096 b

- https://www.anandtech.com/show/11367/nvidia-volta-unveiled-gv100-gpu-and-tesla-v100-accelerator-announced

Porovnání teoretického výpočetního výkonu



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

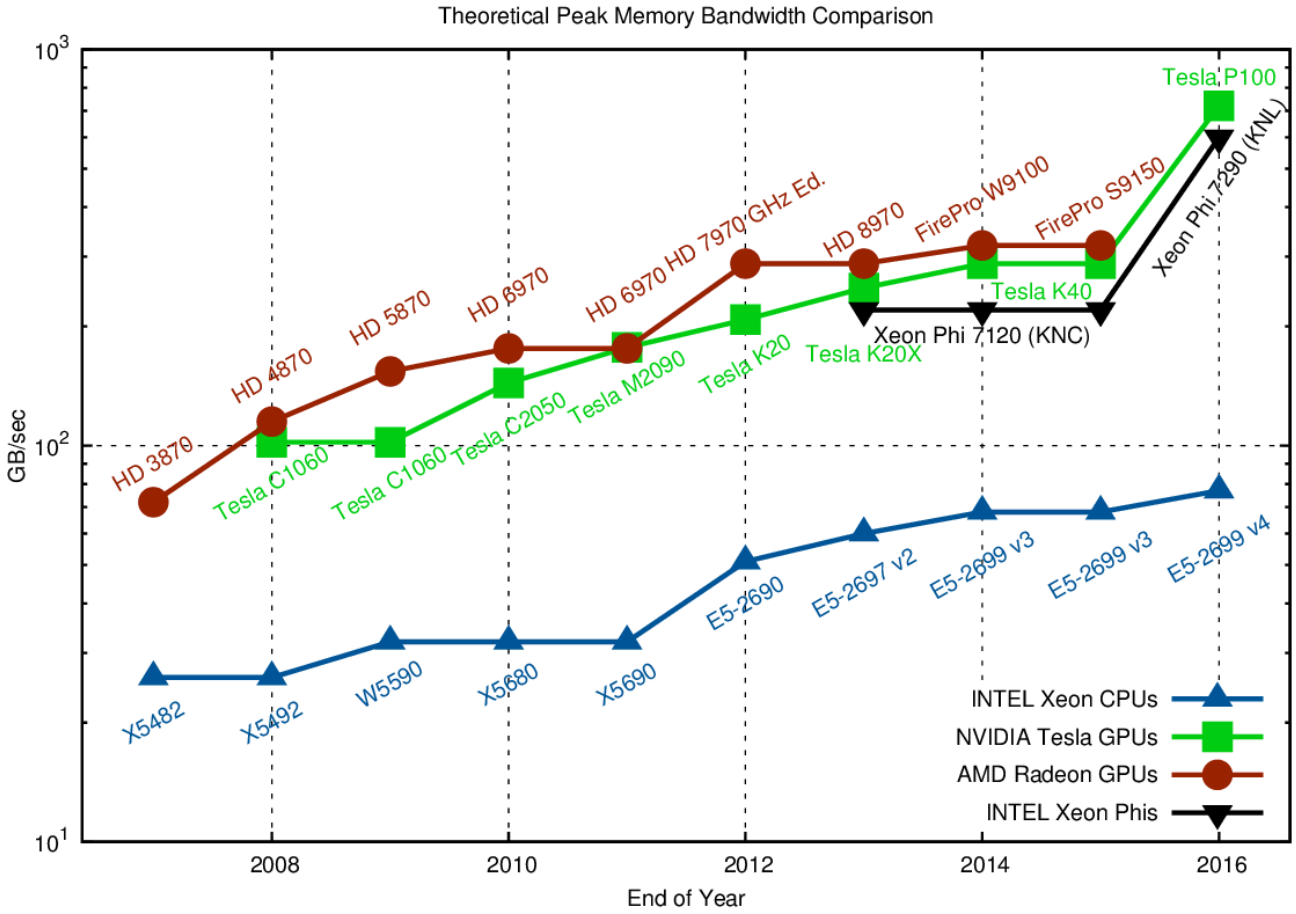
Barbora

Arch.	GFLOPS
V100	15 700
Intel 6240	3 000
Zrychlení	**5,1x**

FIT O204

Arch.	GFLOPS
GTX 970	3920
i5-4460	243
Zrychlení	**16,1x**

Porovnání teoretické propustnosti paměti



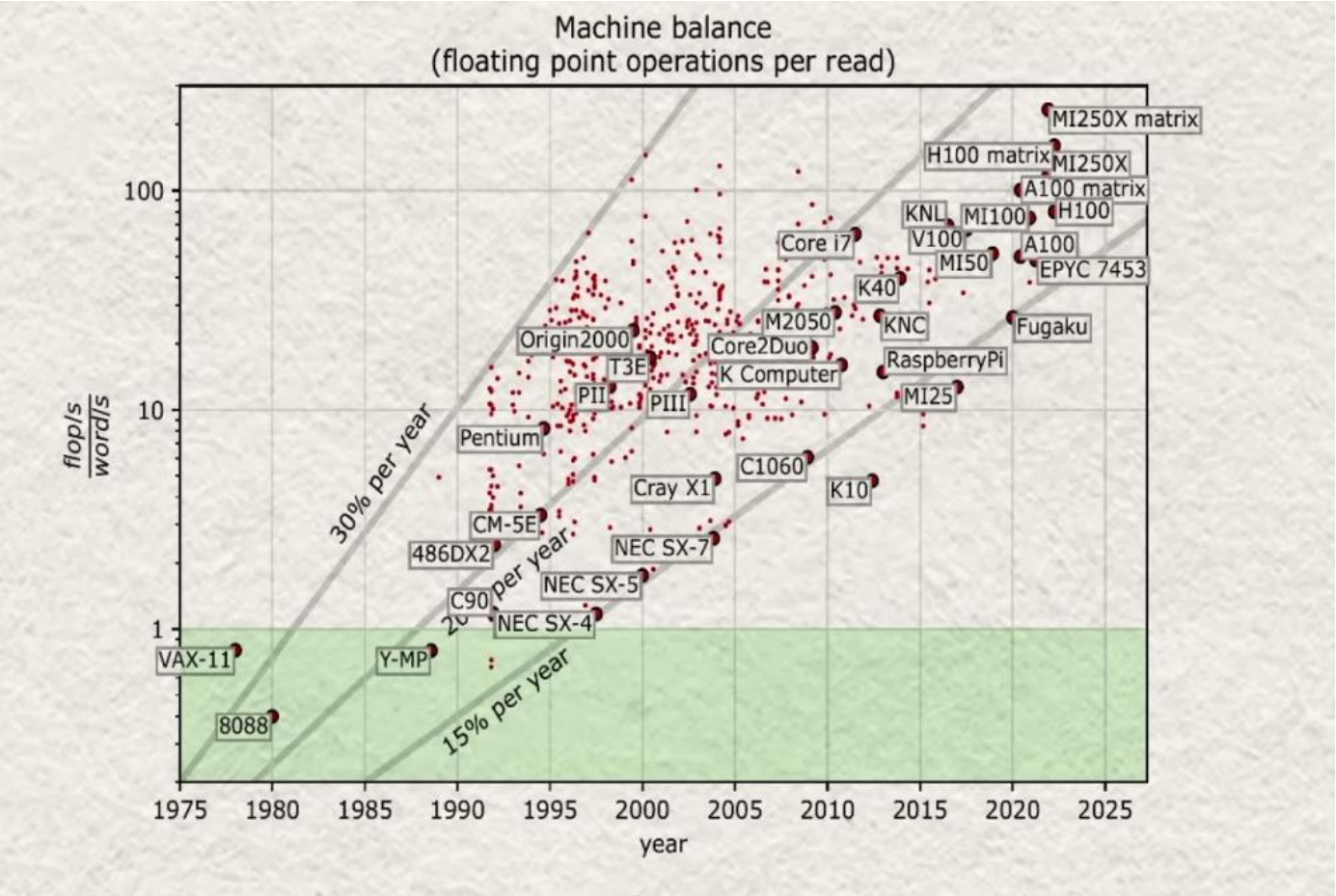
Barbora

Arch.	GB/s
V100	900
Intel 6240	141
Zrychlení	**6,38x**

FIT O204

Arch.	GB/s
GTX 970	224
i5-4460	25,6
Zrychlení	**8,75x**

Aritmetická intenzita nutná pro plné vytížení HW



Jak pracuje GPU – Single Instruction Multiple Threads

- Výpočet rozčleněn do velkého množství vláken.
- Všechna vlákna mají stejnou strukturu.
- SIMT zpracovává jednu instrukci napříč několika vlákny
  - Balíky instrukcí se vykonávají pomocí SIMD stroje
  - Každé vlákno z balíku zpracovává stejnou instrukcí nad jinými daty.
  - Balík vláken zpracovávaný v jednom okamžiku se nazývá WARP. Jeho velikost bývá závislá na počtu výpočetních jednotek.
- Divergence vláken na podmínce způsobí sekvenční vykonávání větví programu.
- Důležitou podmínkou je zanedbatelná latence pro přepínání warpů.
- Pro překrytí paměťových latencí je nutné mít velké množství vláken.

SIMD vs. SIMT zpracování

Součet dvou matic

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    c[i*N+j] = a[i*N+j] + b[i*N+j];
```

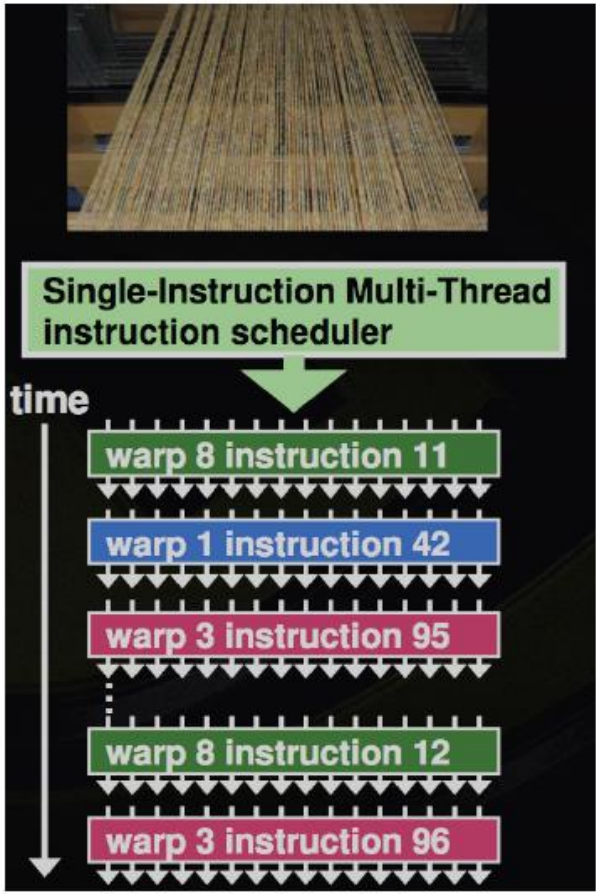
- Matici distribuujeme po blocích na vlákna
- Výpočet každého řádku vektorizujeme pomocí AVX
  - smyčka se rozbalí
  - HW zpracovává 8/16 elementů

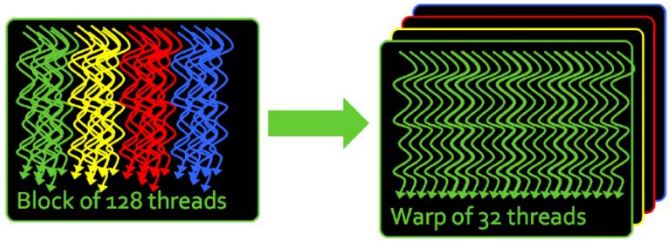
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  #pragma omp simd
  for (int j = 0; j < N; j++)
    c[i*N+j] = a[i*N+j] + b[i*N+j];
```

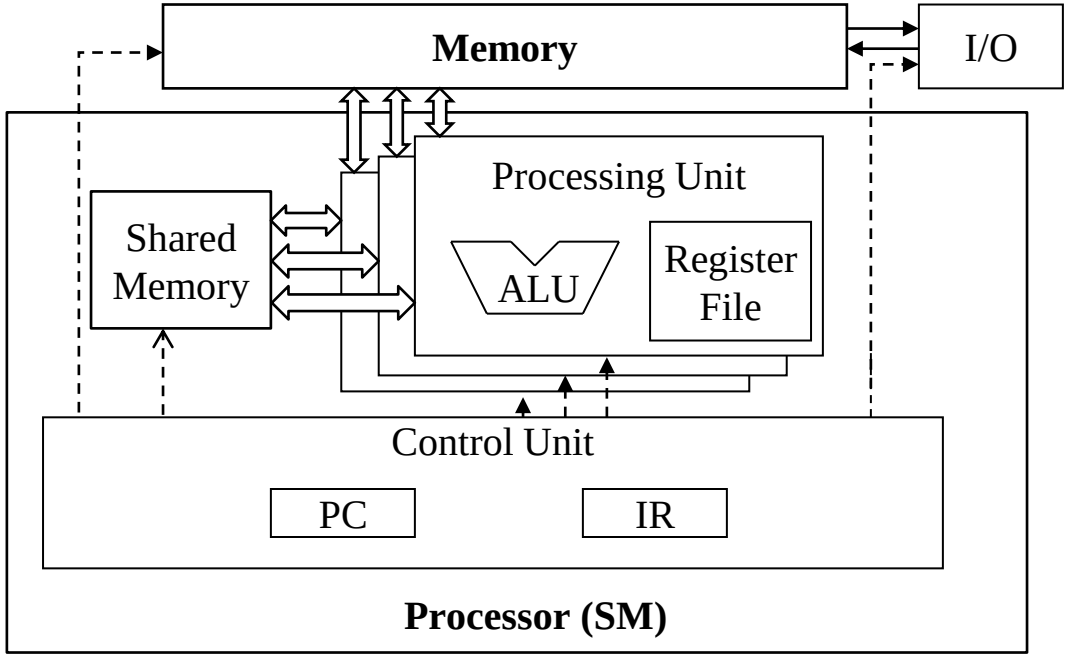
- Sestrojíme kód pro výpočet 1 prvku
  - každé vlákno umí najít svůj prvek
- Vlákna seskupíme do bloků a gridu
  - 1D/2D/3D organizace
  - o plánování vláken se stará HW

```
int i = blockIdx.y * gridDim.y +
        threadIdx.y;
int j = blockIdx.x * gridDim.x +
        threadIdx.x;
if (i < N && j < N)
c[i*N+j] = a[i*N+j] + b[i*N+j];
```

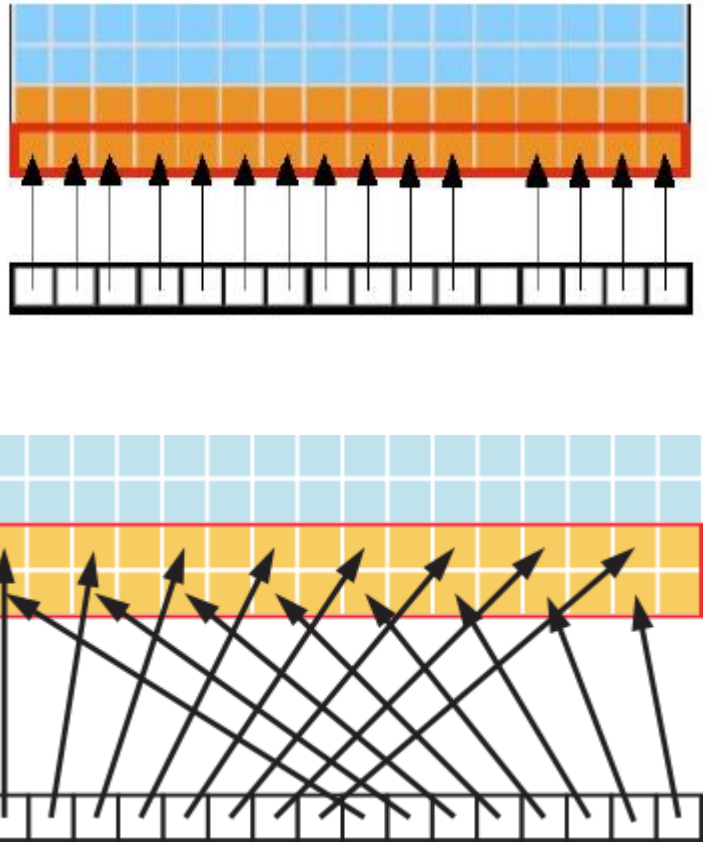
Single Instruction Multiple Threads







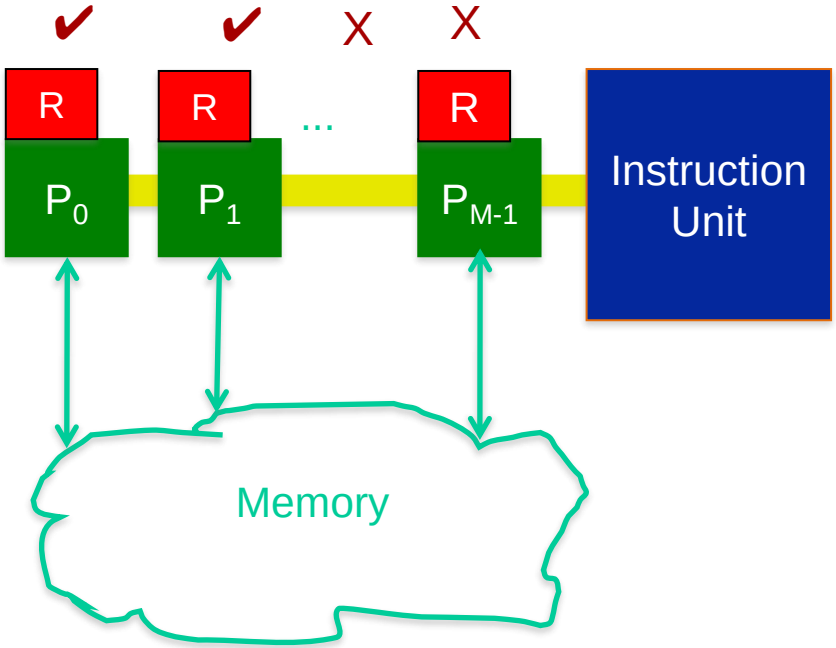
Datová lokalita přístupu do paměti

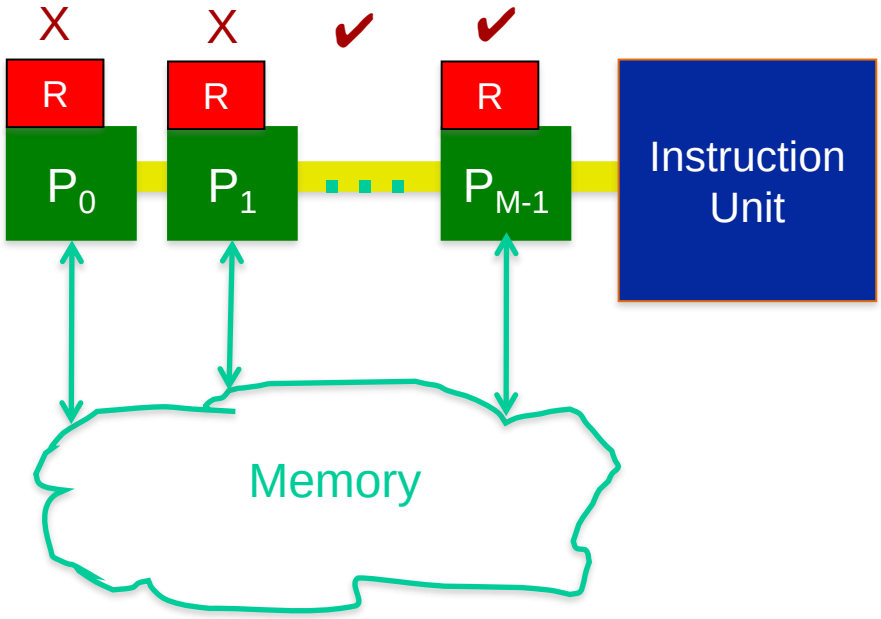


Sousední vlákna musí číst ze sousedních lokací – jinak je problém

Divergence vláken – větvení kódu

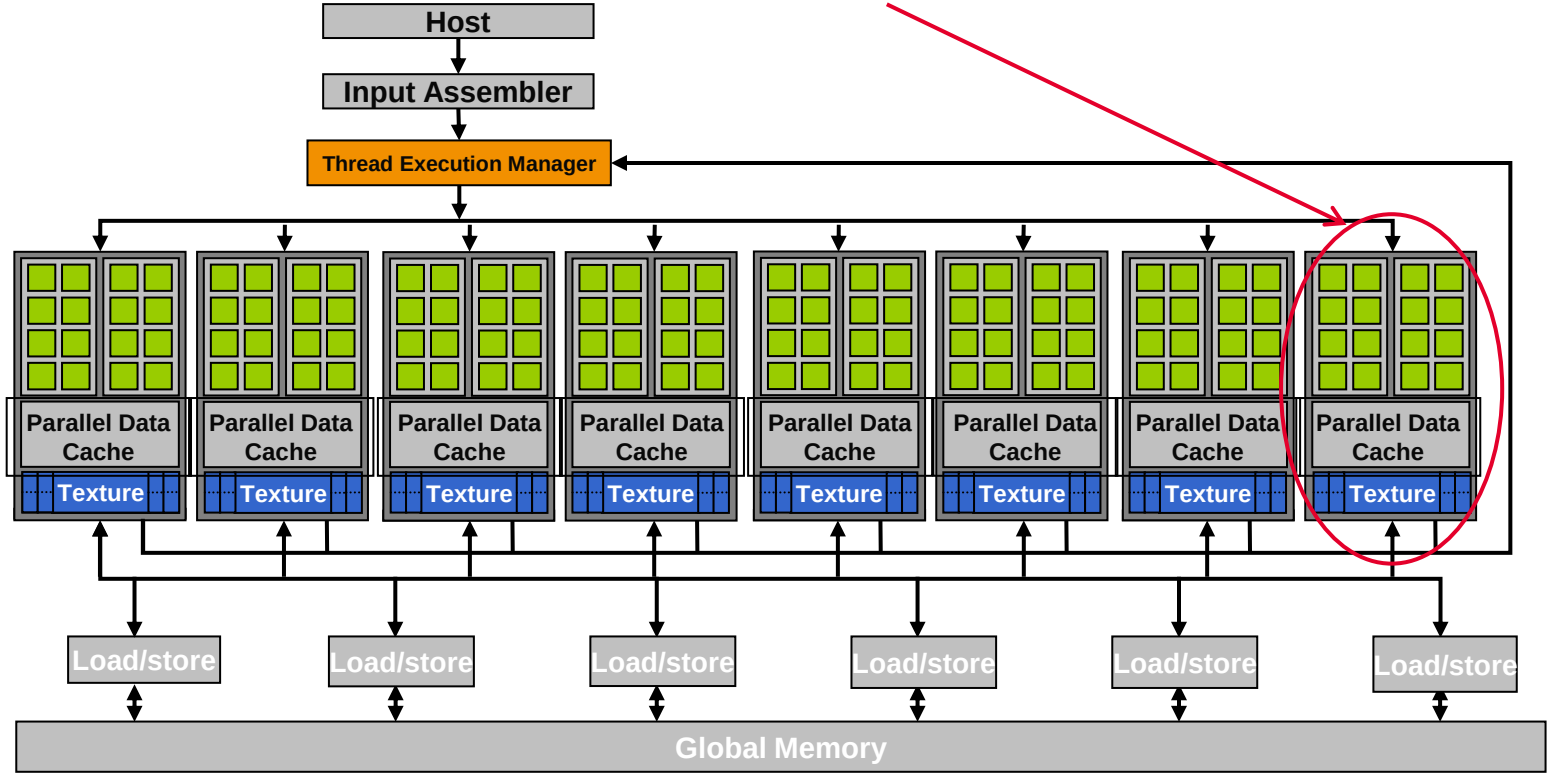
```
if (threadIdx >= 2)
  out[threadIdx] += 100;
else
  out[threadIdx] += 10;
```



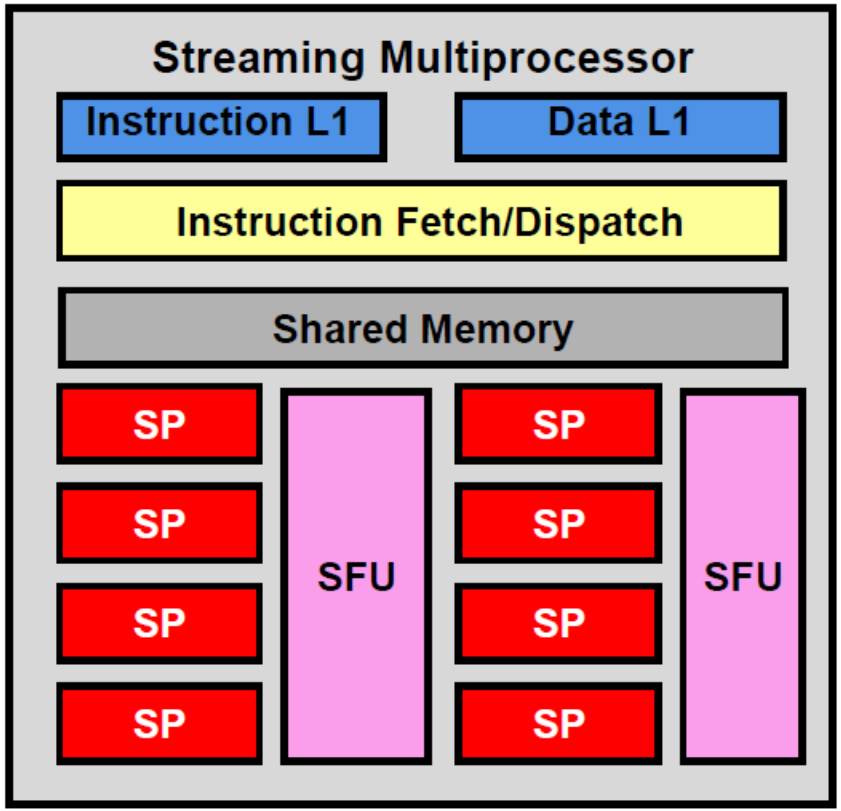


Unifikovaná architektura Nvidia G80

- G80 první jádro určené pro GPGPU s architekturou CUDA
- Jádro členěno do několika Streaming Multiprocesorů (SM)

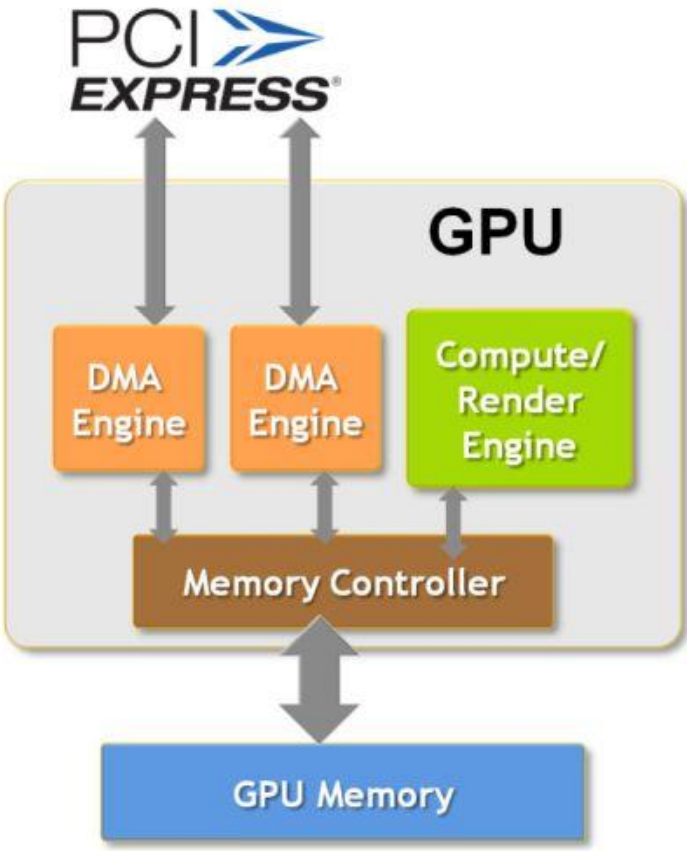


Streaming processor G80



- SM je složeno z
  - 8 CUDA jader (SP) – obsahuje skalární ALU (FMA)
  - 2 Super Function Units (SFU) – speciální jednotka pro počítání dělení, sin, cos, ln,…
  - 8k registrů
- Načítaní/vydávání instrukcí z více vláken
  - až 768 aktivních vláken
  - vždy se načítá instrukce pro 32 vláken (warp)
- 16 KB sdílené paměti

Copy Engine (od Fermi)



- GPU obsahuje DMA Engine pro přímé kopírování dat mezi pamětí počítače a GPU
- GPU grafické karty se tedy o kopírování dat nestará a ani o něm nemusí vědět
- Lze překrývat přenosy dat a výpočet
- Využitelné pro přístup do paměti jiné grafické karty v Multi-GPU systémech

Architektura Nvidia Volta (GV100)

- Až 84 SM procesorů
  - 64 FP32 jednotek
  - 64 INT32 jednotek
    - Float a INT jde současně!
  - 32 FP64 jednotek
  - 8 Tenzorových jader

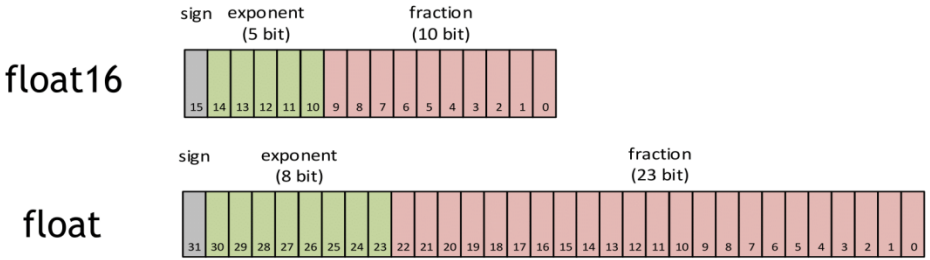
$$D = A \times B + C$$

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 | FP16 | FP16 | FP16 or FP32

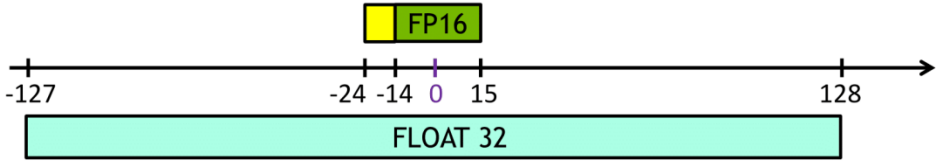
- Zvýšena efektivita L1
- Nový SIMT model
- Superskálární procesor, in-order procesor
- Nová L0 instrukční cache

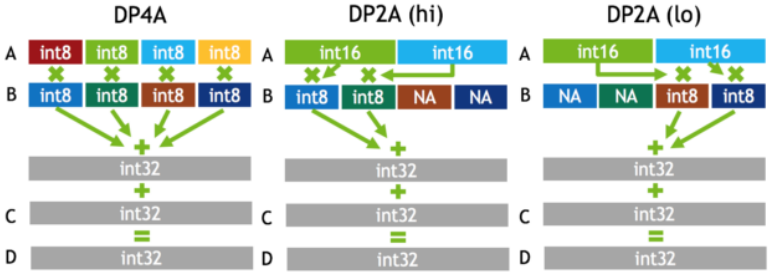
HALF-PRECISION FLOAT (FLOAT16)



FLOAT16 has wide range (2^{40}) … but not as wide as FP32!

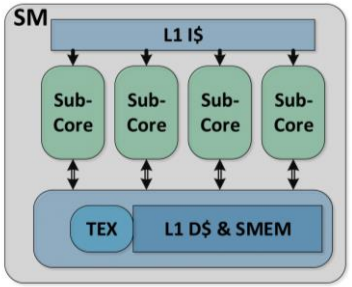
Normal range: $[\ 6\times10^{-5}\ ,\ 65504\ ]$
Sub-normal range: $[\ 6\times10^{-8}\ ,\ 6\times10^{-5}\ ]$





VOLTA GV100 SM

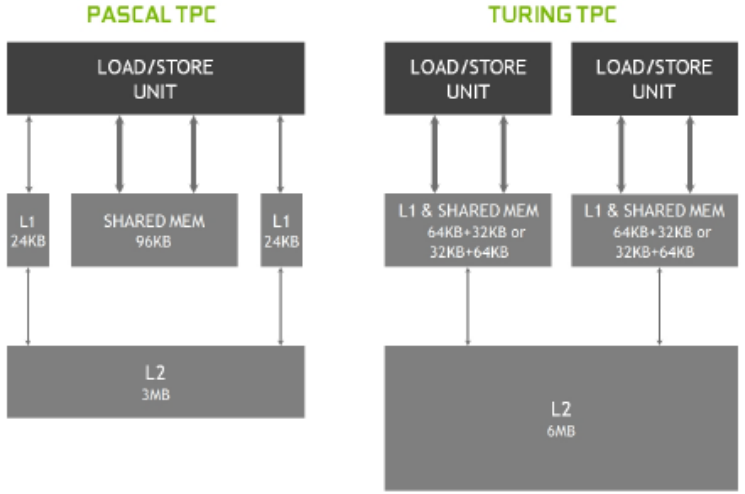
Redesigned for Productivity and Accessible Performance



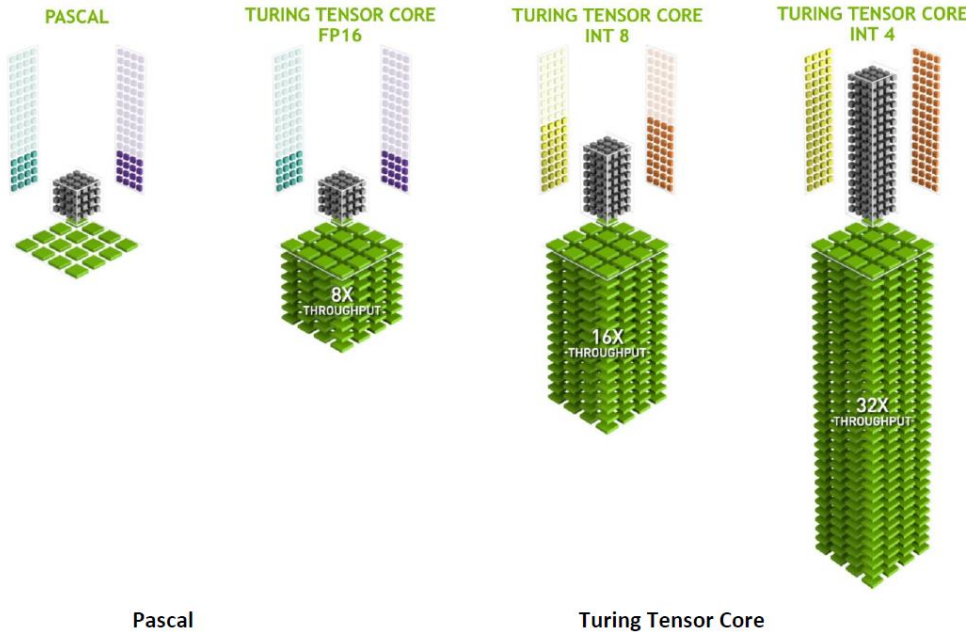
- Twice the schedulers
- Simplified Issue Logic
- Large, fast L1 cache
- Improved SIMT model
- Tensor acceleration
- +50% energy efficiency vs GP100 SM

Architektura Nvidia Turing (TU102)

- Superskalární procesor
  - INT a FP32 současně
- Unifikovaná L1 a sdílená paměť
  - Zdvojnásobena propustnost i kapacita
- Přidána RTX jádra

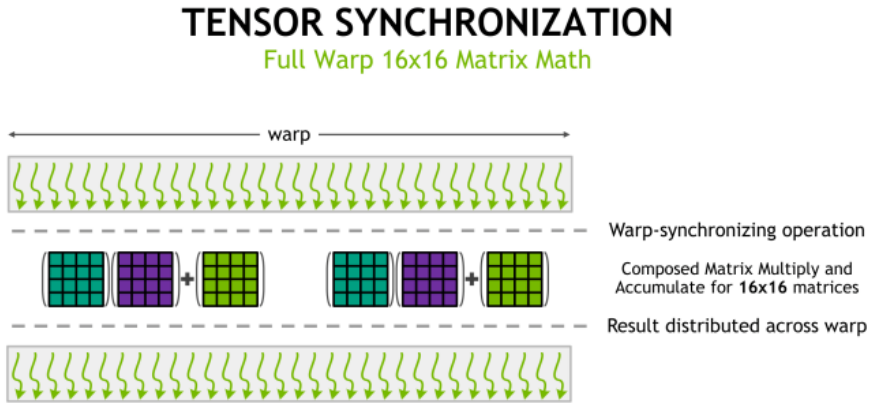


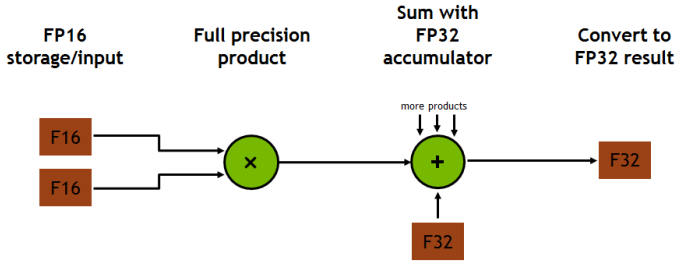
Nvidia Turing (TU102) – Tenzorová jádra



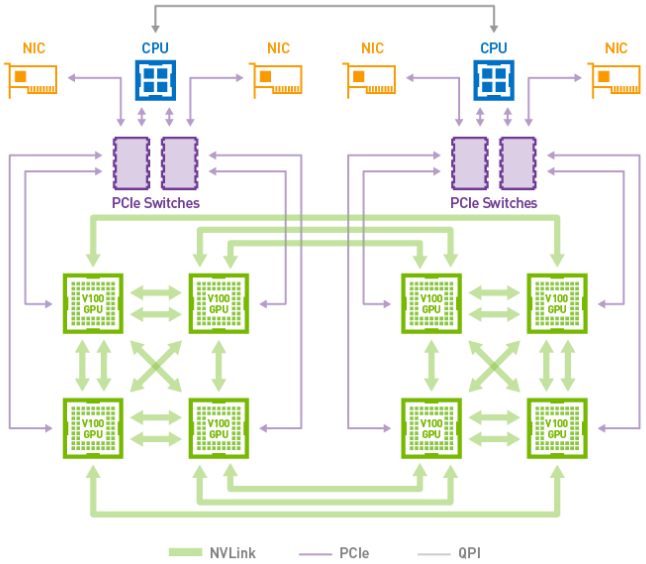
Pascal

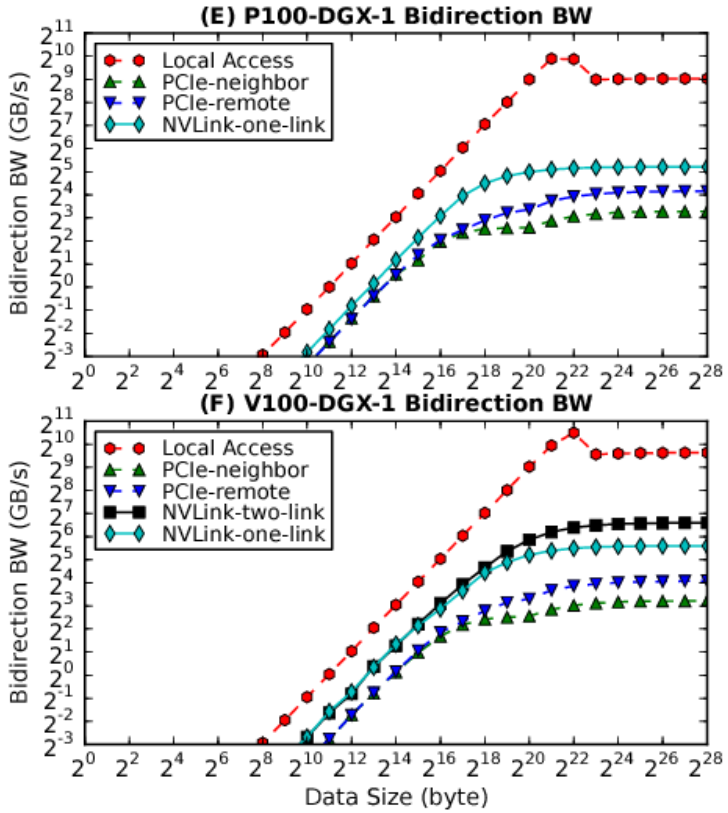
Turing Tensor Core





Nvidia NVLink propojení

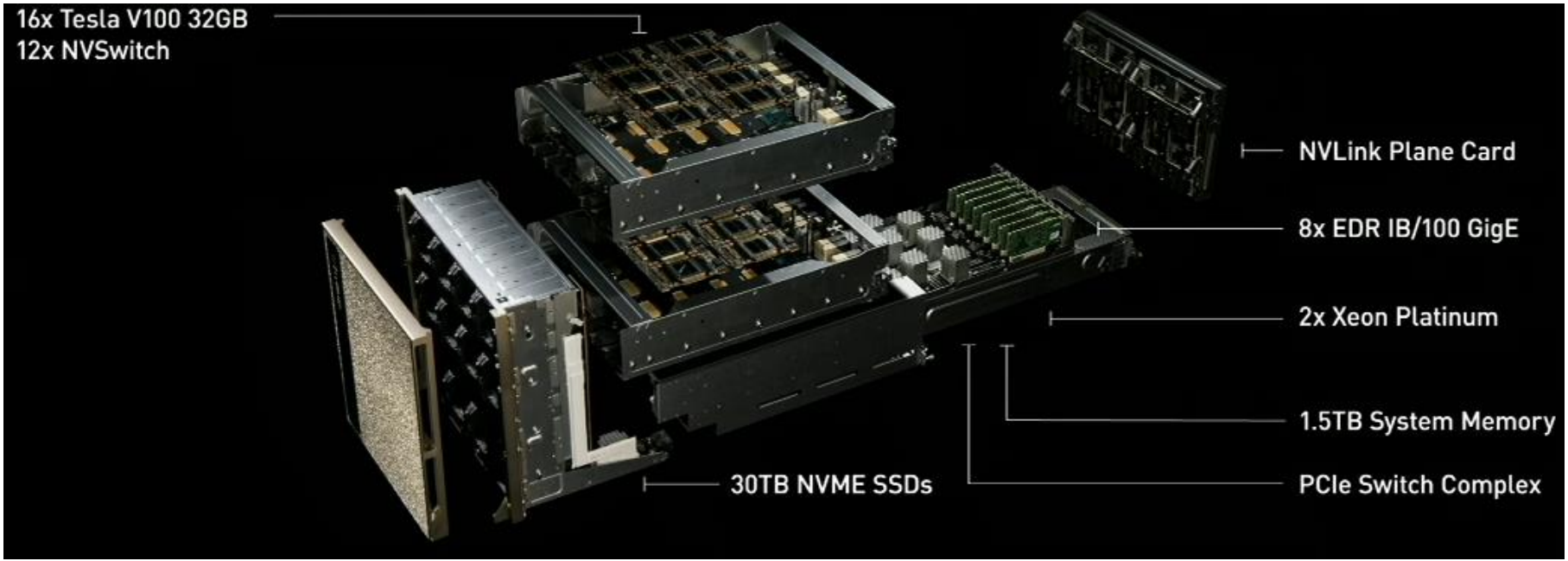




- Vysokorychlostní propojení GPU
- GPU lze připojit i k CPU s podporou NVLink
- Hybrid Cube-mesh propojení
- Při 8 GPU až 160 GB propustnosti

Nvidia DGX-2

- Ostravský nejvýkonnější GPU server
- **https://www.youtube.com/watch?v=gAByU0i6G_E**



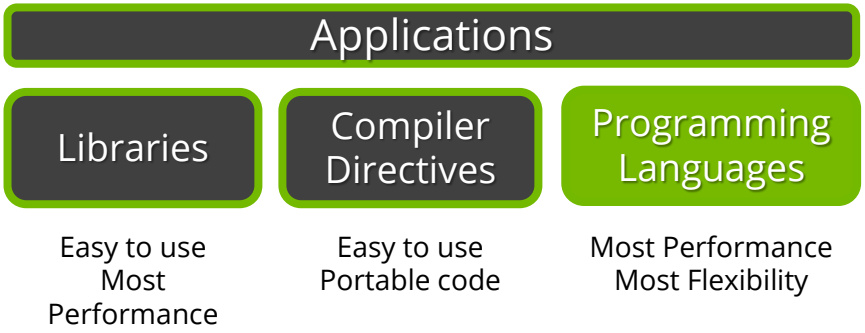
PROGRAMOVÁNÍ GRAFICKÝCH KARET

Programové prostředky pro GPGPU

- **GPGPU – "General-purpose"** computing on graphics processing units

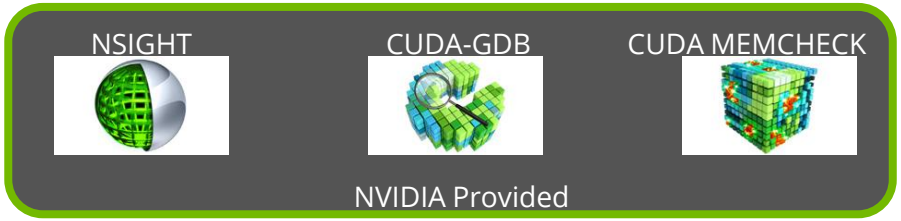
- **Vysokoúrovňové jazyky**
  - **OpenMP** od verze 4.5 obsahuje konstrukci **target** jenž umožní vykonání kódu na akcelerátoru
  - **OpenACC** – (PGI/NVIDIA) – obdoba OpenMP
  - **MATLAB** – přetypováním na **gpuArray**
  - **NVIDIA Thrust** – C++ interface pro GPU NVIDIA

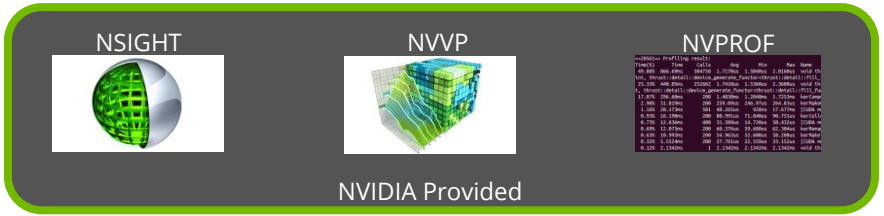
- **Nízkoúrovňové jazyky**
  - **CUDA** – (NVIDIA), vysoce optimalizováno pro karty NVIDIA, obsahuje sadu důležitých knihoven FFT, BLAS, RAND …
  - OpenCL – (Khronos) – podpora i pro karty AMD, CPU, XeonPhi, DSP, FPGA
  - HIP - C++ Heterogeneous-Compute Interface for Portability – AMD

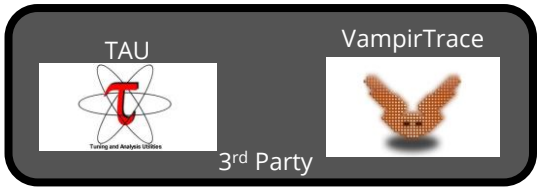


Nvidia CUDA

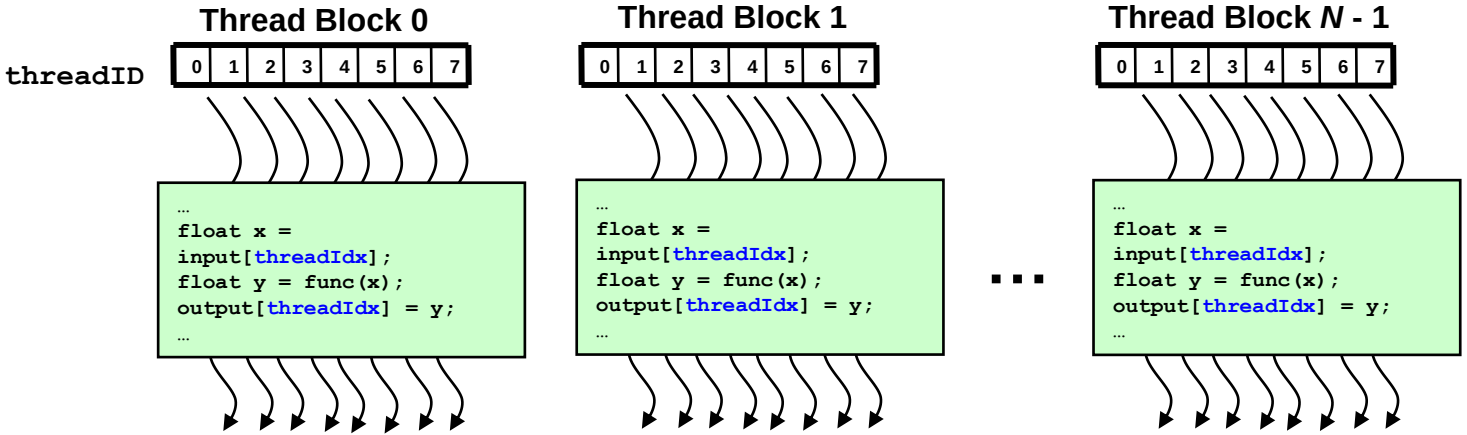
- „Compute Unified Device Architecture”
- Rozšíření jazyka *C*/*C++* a knihovní funkce pro využití GPU jako obecné platformy
- Binding pro Fortran, Python, Java, Matlab, Ruby,...
- Obsahuje knihovny: cuBLAS, cuFFT, cuRand, Cuda Math Library, Thrust, Magma, NPP
- Debuggery a profilery
- Obsahuje funkce pro
  - Přenos dat do paměti grafického adaptéru a zpět
  - Podpora pouze pro statická 1D pole a CUDA arrays
  - Rozčlenění výpočtu do vláken a bloků
  - Funkce pro synchronizaci vláken na úrovni bloků
  - Spuštění výpočtu na grafické kartě
  - https://developer.nvidia.com/cuda-zone





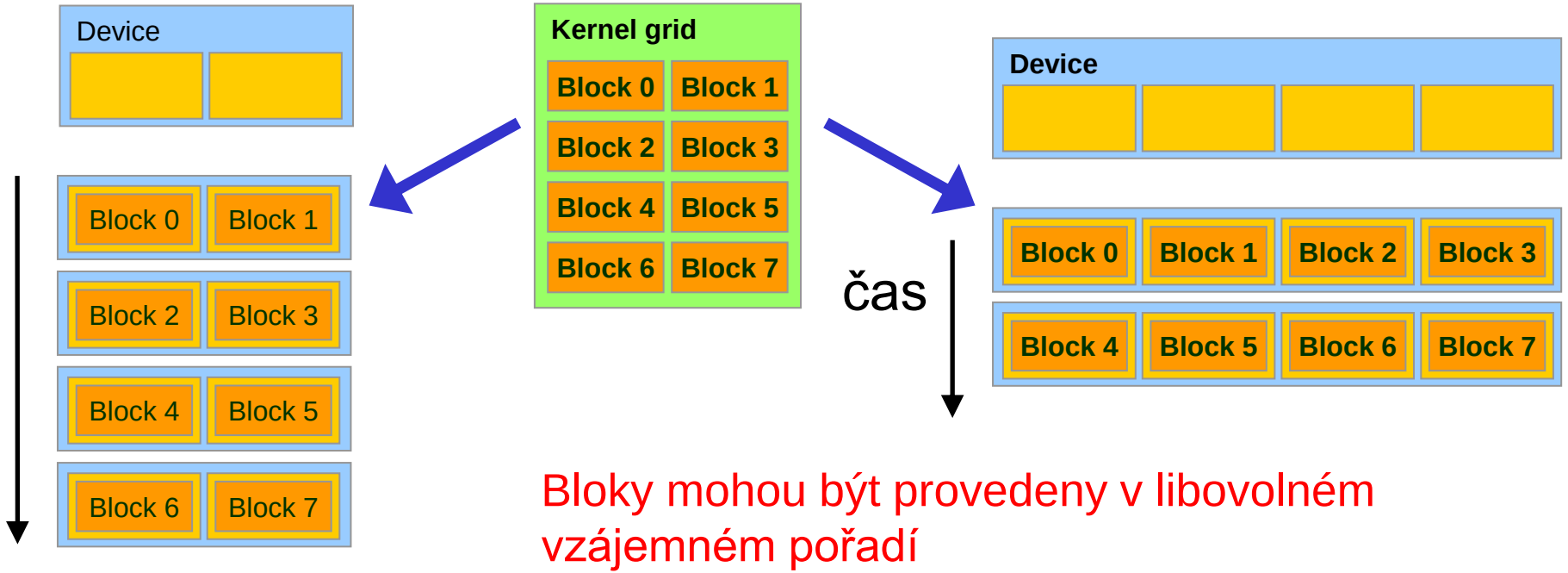


- **Funkce akcelerované pomocí GPU se nazývá kernel**
  - Kód se zapisuje z pohledu jednoho vlákna, ale vždy je nutné brát v potaz, že běží celý WARP.
- **Monolitická sada vláken je rozdělena do bloků**
  - Vlákna v rámci bloku spolupracují pomocí sdílené paměti, atomických operací a bariérové synchronizace
  - Vlákna v různých blocích **NEMOHOU** spolupracovat



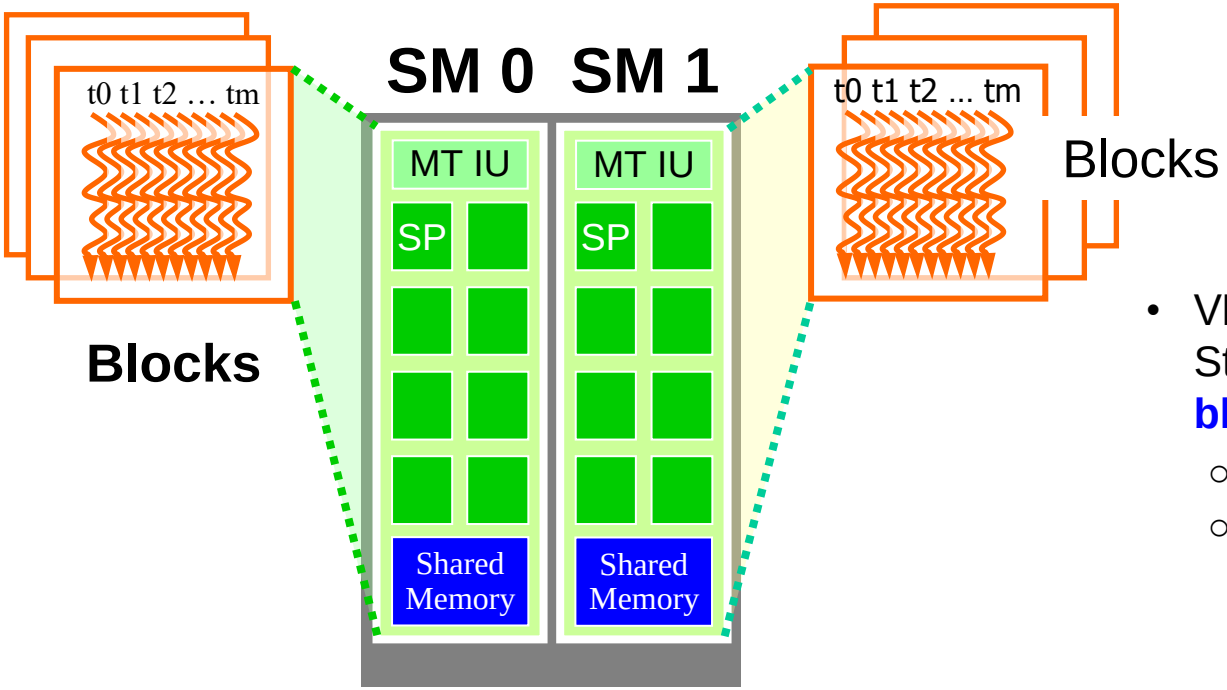
Transparentní škálovatelnost

- GPU může rozhodnout o libovolném přiřazení bloků na SM procesory
  - Spuštěný kernel (grid – sada všech bloků) je dobře škálovatelný na libovolném počtu SM procesorů.



Bloky mohou být provedeny v libovolném vzájemném pořadí

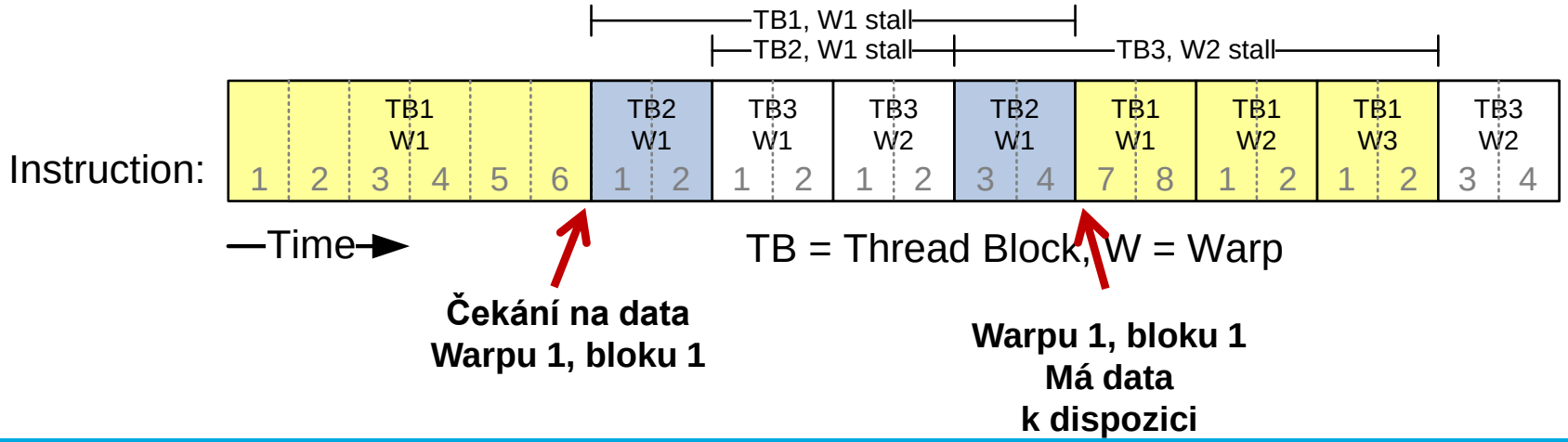
Alokace GPU zdrojů



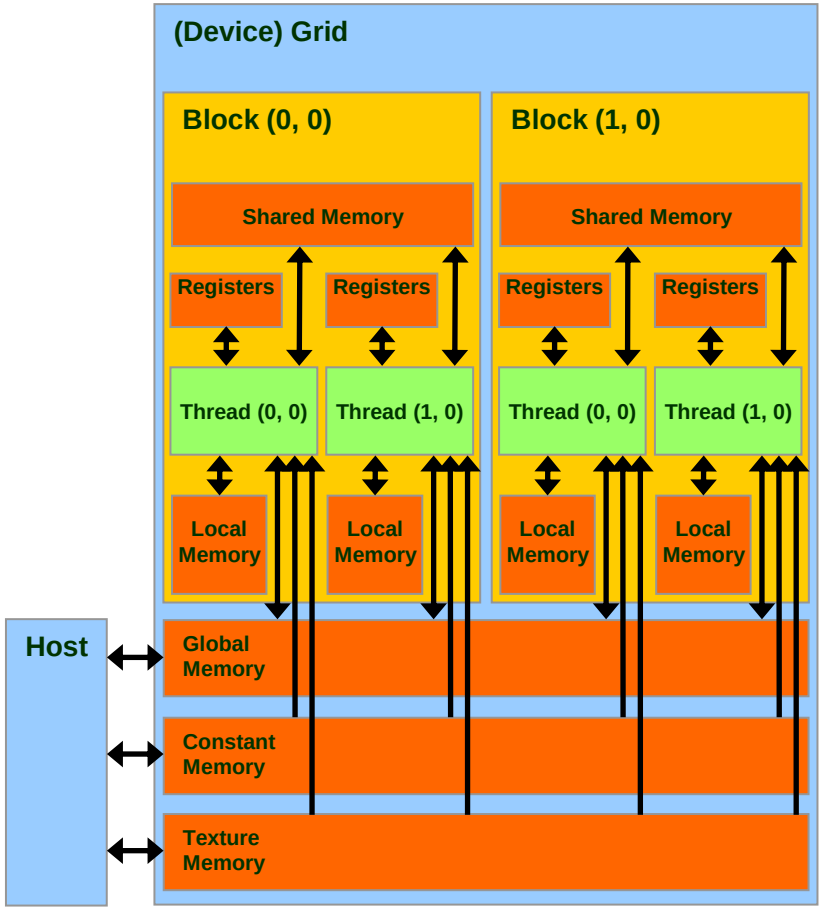
Flexibilní alokace zdrojů (registry, sdílená paměť)

- Vlákna jsou přiřazována na Streaming Multiprocesory **po blocích**
  - Až **8** bloků může sdílet zdroje SM
  - SM v G80 zvládne až **768** threads
    - Např. 256 (vláken na blok) * 3 bloky
    - Nebo 128 (vláken na blok) * 6 bloků
    - atd.
- Vlákna běží současně
  - SM udržuje ID vláken/bloků
  - SM plánuje provedení vláken

- **SM dokáže přepínat warpy s nulovou režií**
  - Všechna **vlákna v rámci warpu** vykonávají **stejnou instrukci** SIMT
  - V jeden okamžik může být vykonáván pouze jeden warp (Kepler a Maxwell dovolují více warpů)
  - Pouze warpy jejichž následující instrukce má připraveny všechny operandy mohou být spuštěny (ready).
  - Ready warpy jsou vybírány pro spuštění na základě priorit.
  - Pokud nějaký warp zahájí čtení/zápis z/do globální paměti (až 200 taktů čekání) je odložen a nahrazen jiným, který může běžet

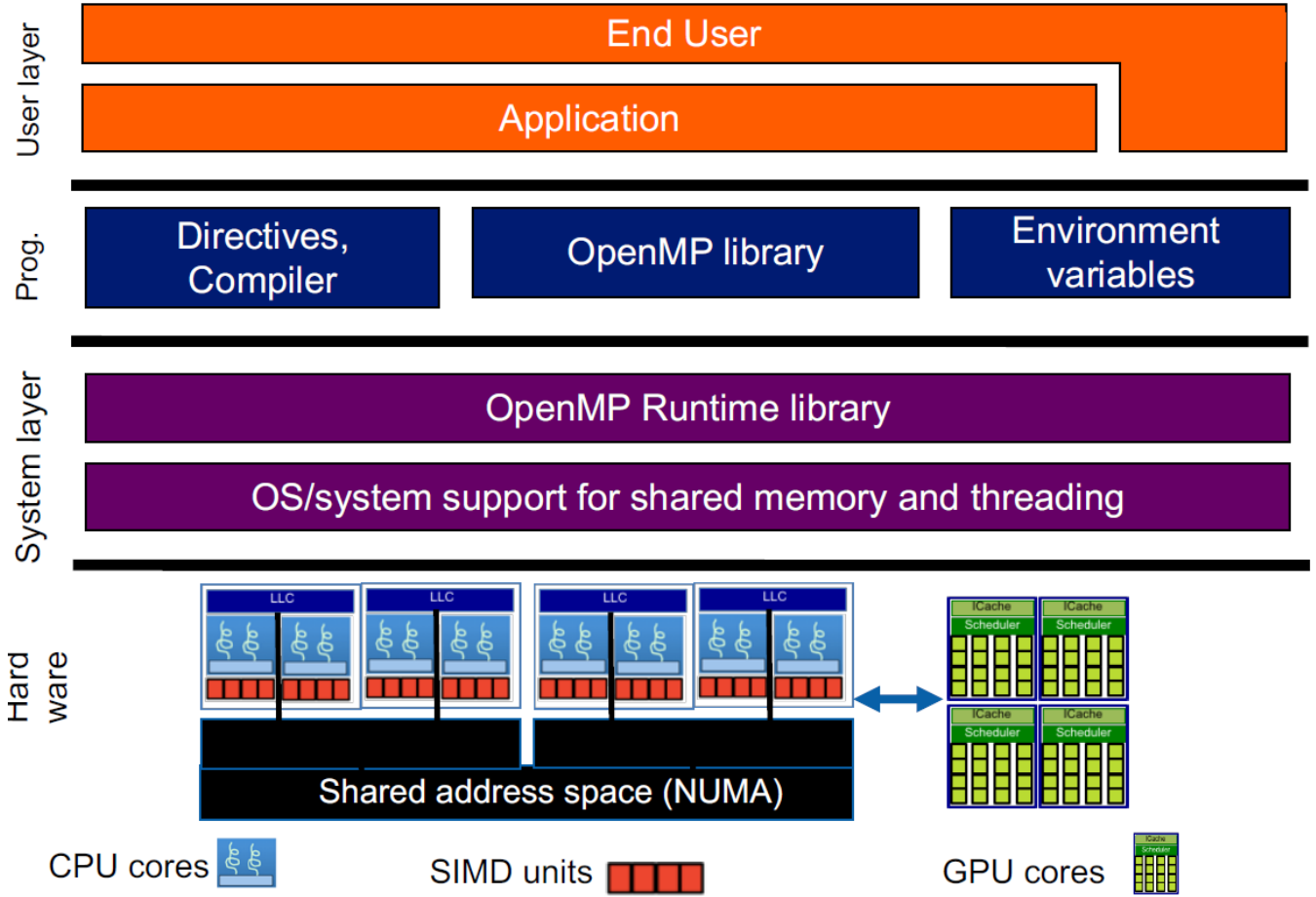


- **Vlákna CUDA pracují s**
  - R/W registry vlákna
  - R/W lokální paměť vlákna
  - R/W sdílená paměť bloku
  - R/W globální paměť gridu
  - R konstantní paměť gridu
  - R texturní paměť gridu
- **Host může přistupovat do**
  - R/W globální paměti
  - R/W konstantní paměti
  - R/W texturní pamětí



OPENMP FOR ACCELERATORS

Rozšíření OpenMP o podporu akcelerátorů



- **Direktiva `target` předává (`offload`) výpočet na akcelerátor (`device`)**

#pragma omp target
{….} // a structured block of code

1. Program begins. Launches **initial thread** running on the **host device**.
2. Implicit parallel region surrounds entire program
3. Initial task begins execution

4. Initial task encounters the **target** directive.
5. Initial task generates a target task which is a mergable, included task
6. Target task launches target region on the device

7. A new initial thread runs on the device.
8. Implicit parallel region surrounds device program
9. Initial task executes code in the target region.

10. Initial task on host continues once execution associated with the target region completes

Mapování dat v rámci target regionu

- **Pamatujte: host (CPU) a device (GPU) mají oddělené paměťové prostory**
  - OpenMP používá kombinaci implicitních a explicitních data transferů.
  - Data mohou migrovat mezi host a device pouze v určitých, přesně definovaných místech
    - **Typicky na začátku a na konci target regionu**

```
#pragma omp target
{ // Data se kopírují z host na device
  …
} // Data se kopírují z device na host
```

  - Implicitně se přenáší:
    - Skalární proměnné (`int N`) a to vždy jako **`firstprivate`**. Vždy pouze na device!
    - Pole, pokud je známá velikost (`double A[1000]`). Provádí se kopie tam i zpět!
    - **Pointery se kopírují jako firstprivate – ALE už ne data na která ukazují!!!**

Příklad: Transfery dat mezi host a device

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {

    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

1. Variables created in host memory.

2. Scalar **N** and stack arrays **A** and **B** are copied *to* device memory. Execution transferred to device.

3. **ii** is **private** on the device as it's declared within the target region

4. Execution on the device.

5. stack arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Paralelní vykonávání kódu na host a device

```
#pragma omp target nowait
{
// code defines a target region
}
```

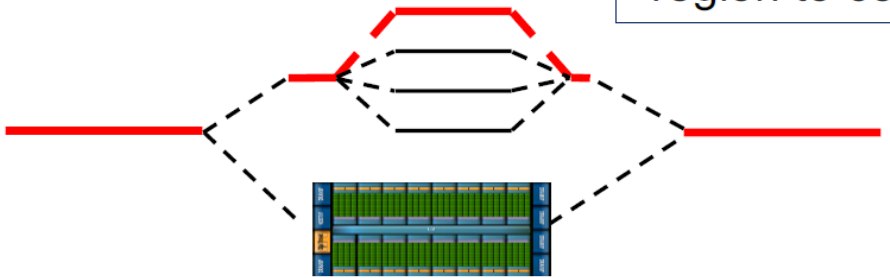
- Make the target task (running on the host) execute as a deferred task

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    big_stuff(i);
}
```

- The thread's implicit task can define other (potentially parallel) work.

```
#pragma omp taskwait
```

- The implicit task running on the host waits for the target region to complete.



Sémantika pragmy target

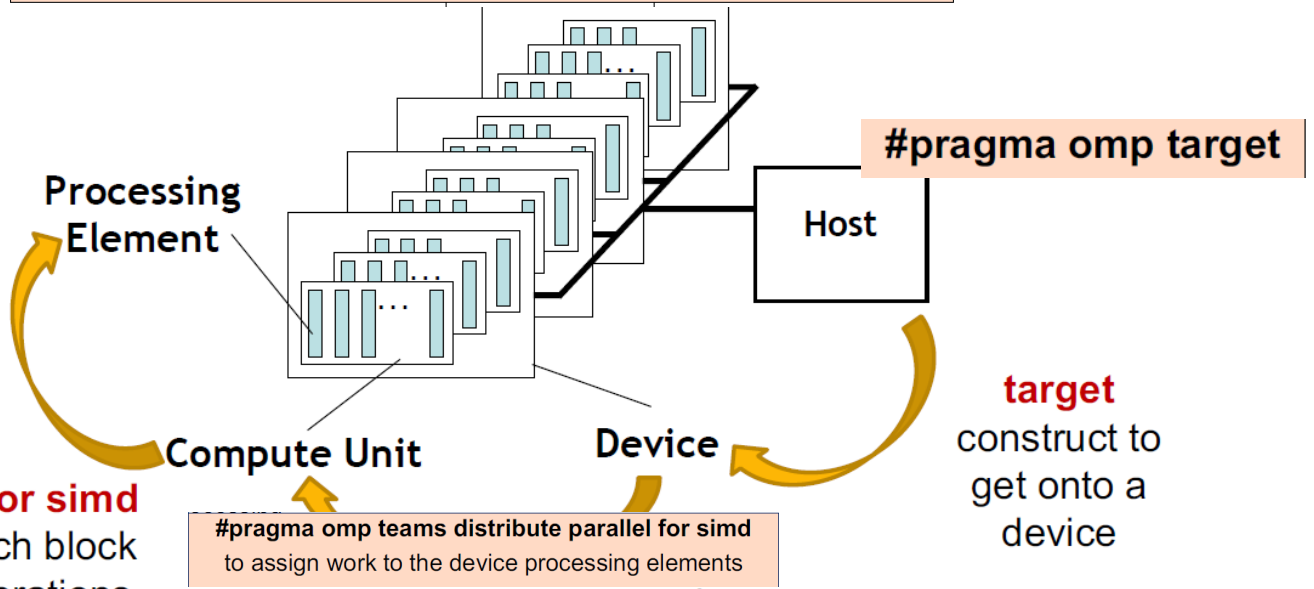
#pragma omp target [clause[[,]clause]...]
structured-block

- **if(scalar-expression)**
  - Pokud se vyhodnotí jako true, vykonání proběhne na device, v opačném případě na hostu
- **device(integer-expression)**
  - ID device, které se má použít
- **private(list) firstprivate(list)**
  - Seznam privátních a first privátních proměnných
- **map(map-type: list)**
  - Kopie polí na device.
    - to – na začátku oblasti target kopíruj data na device. Např. `map(to:A[100:200])` – vždy od:kolik
    - from – na konci oblasti target kopíruj data zpět na host. Např. `map(from:B[100:200])`
    - tofrom – kopíruj data oběma směry
    - alloc/delete – nic nekopíruj, pouze alokuj a pak zlikviduj pole na GPU (typicky pomocná pole)
- **nowait**
  - Nečekej na dokončení výpočtu na device

```
int  N = 1024;
int* A = malloc(sizeof(int)* N);
#pragma omp target map(A[0:N])
{
   // N, ii and A all exist here
   // The data that A points to DOES exist here!
} 0
```

Model výpočetní platformy

Typical usage ... let the compiler do what's best for the device:



target construct to get onto a device

parallel for simd to run each block of loop iterations on the processing elements

#pragma omp teams distribute parallel for simd to assign work to the device processing elements

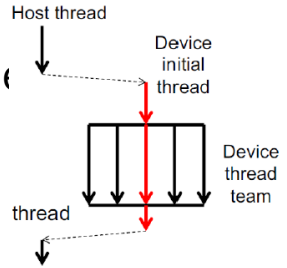
teams construct to create a league of teams with one team of threads on each compute unit

distribute construct to assign blocks of loop iterations to teams

Pragmy teams a distribute

```
#pragma omp target
#pragma omp parallel for
 for (i=0;i<N;i++)
   …
```

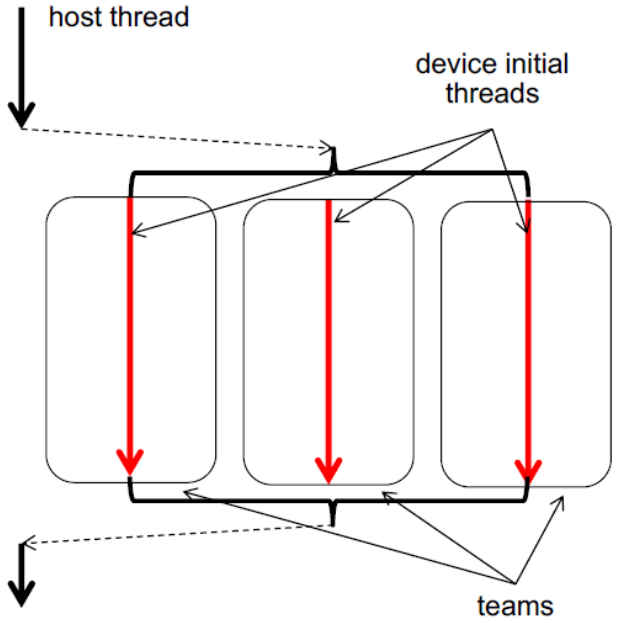
- **The teams construct**
  - Similar to the parallel construct
  - It starts a league of teams
  - Each team in the league starts with one initial thread – i.e. a team of one thread
  - Threads in different teams cannot synchronize with each other
  - The construct must be “perfectly” nested in a target construct
- **The distribute construct**
  - Similar to the for construct
  - Loop iterations are workshared across the initial threads in a leagu
  - No implicit barrier at the end of the construct
  - dist_schedule(kind[, chunk_size])
    - if specified, scheduling kind must be static
    - Chunks are distributed in round-robin



Spuštění více týmů

- teams construct
- distribute construct

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
 for (i=0;i<N;i++)
   …
```



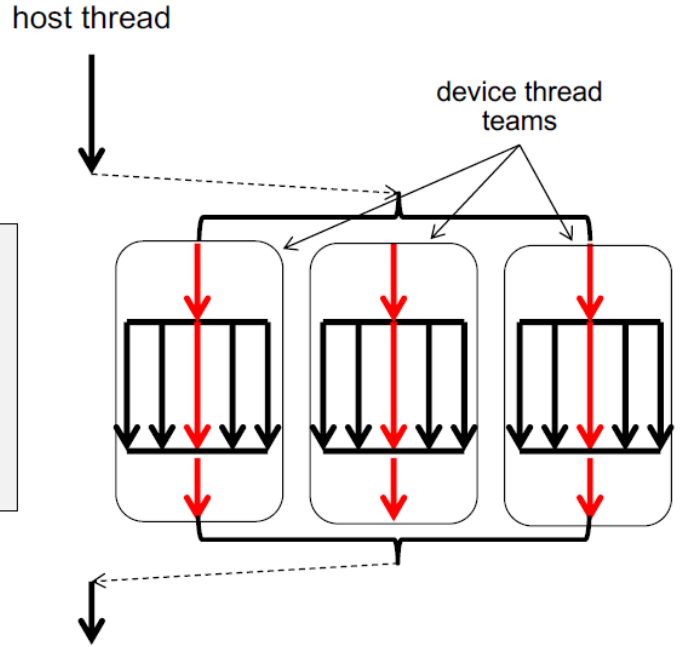
- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

Note: number of teams is implementation defined, good for portable performance. Compilers can choose how they map teams and threads.

Paralelizace kódu na device

- teams distribute
- parallel for simd

```
#pragma omp target
#pragma omp teams distribute
 for (i=0;i<N;i++)
 #pragma omp parallel for simd
 for (j=0;j<M;j++)
   …
```



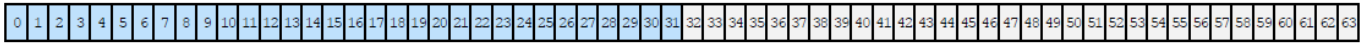
- Transfer execution control to MULTIPLE device initial threads (one per team)
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the master thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for simd)

Ukázka distribuce práce

```
#pragma omp target teams distribute parallel for simd \
  num_teams(2) num_threads(4) simdlen(2)
 for (i=0; i<64; i++)
    …
```

64 iterations assigned to 2 teams;
Each team has 4 threads;
Each thread has 2 SIMD lanes

Distribute iterations across 2 teams



In a team, **workshare** (parallel for) iterations across 4 threads

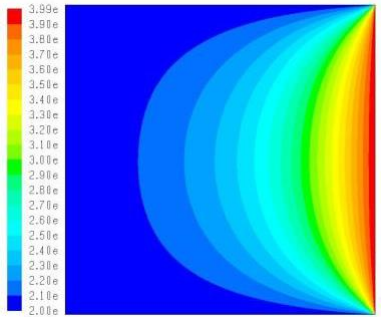
In each thread use **SIMD** parallelism

Ukázka: Paralelní stencil operace

```
// Compute the next timestep, given the current timestep
void solve(const int n, const double alpha, const double dx, const double dt, const double * restrict u,
double * restrict u_tmp) {
  // Finite difference constant multiplier
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;

  // Loop over the nxn grid
  #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
  #pragma omp teams distribute parallel for simd collapse(2)
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {

      // Update the 5-point stencil, using boundary conditions on the edges of the domain.
      // Boundaries are zero because the MMS solution is zero there.
      u_tmp[i+j*n] =  r2 * u[i+j*n] +
      r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
      r * ((i > 0)   ? u[i-1+j*n] : 0.0) +
      r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
      r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
}}}
```



Add the BUD to the loops
Use collapse clause to increase parallelism

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\alpha k}{h^2}(u_{i,j+1}^{n} + u_{i,j-1}^{n} + u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n})$$

Teorii znáte, tak hurá do praxe

Veselé Vánoce a šťastný Nový rok přeje Jirka Jaroš a kol.