Parallel paradigm

David Celný

Department of Physical Chemistry, UCT Prague

celnyd@vscht.cz

September 2, 2021

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

Overview

Programming background
Memory
Building blocks
Conclusion

Hello programming

C

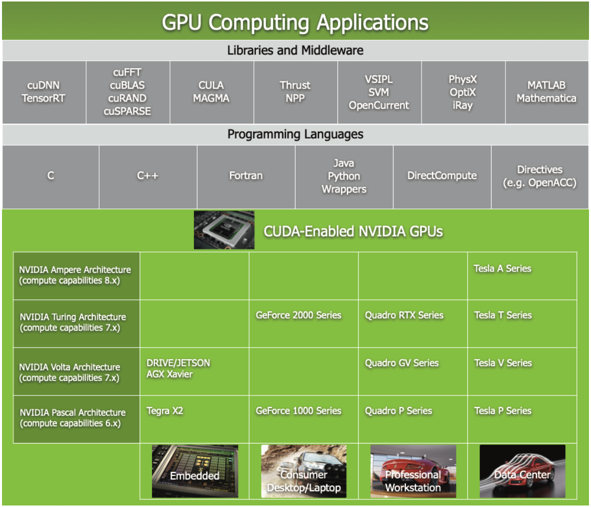
- Dennis Ritchie at 1972
- GP, procedural, imperative, statically typed
- direct memory control
- standardized *(current C17)*

C++

- Bjarme Stroustrup at 1985
- add OOP & functional
- standardized *(current C++20)*

CUDA

- Nvidia at 2007
- parallel computing platform, API to GPU, scalable *(across GPU)*
- works with C, C++, Fortran
- control of GPU from CPU *(not full)*
- new device memory space
- one code → split compilation for CPU/GPU
- dedicated libraries *(cuBLAS, cuFFT,cuRAND ...)*



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

Figure: Look at the GPU specialization on different tasks. © docs.nvidia.com

```
#include <stdio.h>
__global__ void hello_kernel()
{
    int thread_id = (blockDim.x*blockIdx.x + threadIdx.x);
    printf("Hello, World from thread: %d\n", thread_id);
}

int main ()
{
    hello_kernel<<<1,1>>>();
    \\ hello_kernel<<<1,48>>>(); \\ What happens here?
    \\ hello_kernel<<<4,12>>>(); \\ And here ?
    cudaDeviceSynchronize();
    return 0;
}
```

- multiple memory levels, caches
- specialized types *(constant, texture)*
- **high latency** require tricks *(multidispatch, swap)*
- memory **sensitive to coalesced access**
- adjustable caching ability *(L1 vs Shared mem.)*
- access pattern heavily influence efficiency

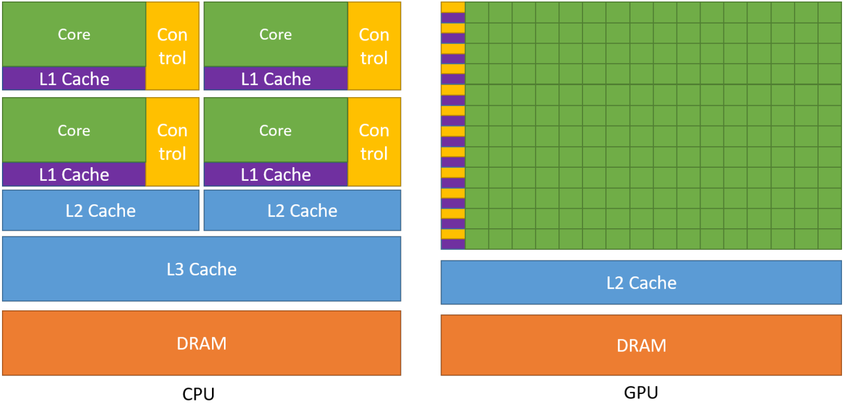


Figure: Schematical comparison of CPU/GPU memory spaces and cores.
@ docs.nvidia.com

New memory playground

Type	**Size***(deviceQuery)*	**Latency***(rule of thumb)*
1. RAM	1. 2GB - 64GB	1. 800-1000 x *(or more)*
2. global mem.	2. 1GB - 24GB*	2. 80-120 x
3. shared mem. *(cache)*	3. 64, 128kB*	3. 7-12 x
4. constant mem.	4. 32, 48, 64kB*	4. 6-10 x *(readonly)*
5. local mem. *(registers)*	5. 64kb for all threads	5. 1

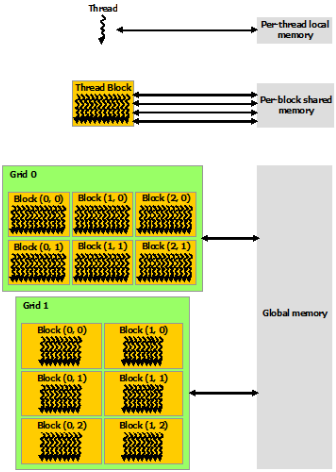
Operational groups

residence

1. core
2. SM
3. GPU
4. multiGPU

code specification

1. $<<<?, \#$ **threads** $>>>$
2. $<<< \#$ **blocks**$, ? >>>$
3. $<<<?, ? >>>$
4. $<<<?, ?, ?,$ **stream** $\# >>>$

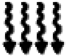


thread

- the smallest unit
- occupy single core
- located by threadIdx *(x,y,z)*
- use registers/local memory for storage
- can't communicate directly with other threads
- synchronized by *__syncthreads()*

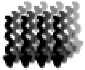
block

- group of threads
- run concurrently *(up to 32, divergence)*
- located by blockIdx *(x,y,z)*
- size determined by blockDim *(x,y,z)*
- use shared memory for storage/communication within block
- isolated from other blocks



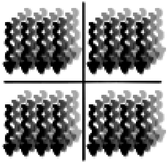
warp

- 32 threadblock *(halfwarp)*
- scheduled for evaluation *(max group)*



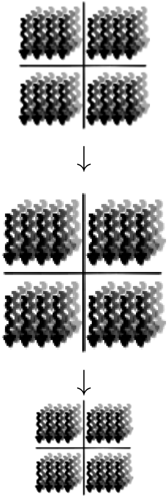
grid

- lattice of blocks
- occupy device *(stream on device)*
- dimension available in gridDim *(x,y,z)*
- use global memory for communication
- no precise control how it is distributed on device
- implicit synchronization at the end of kernel



stream

- flow of kernel launches with same purpose
- default stream, other require prior initialization
- can be multiple on single device
- utilize available resources *(priorities)*
- way how to parallelize on multiple GPU
- use global memory for communication
- explicit synchronization with *cudaStreamSynchronize()*



Summary

- what we work with
- memory is the key to speed
  - different types of it
- building blocks of program
  - and its hierarchy

References

Gerassimos Barlas (2015)
Multicore and GPU Programming: An Integrated Approach
Elsevier publishers ISBN: 978-0-12-417137-4

Thomas Sterling, Matthew Anderson & Maciej Brodowicz (2018)
High Performance Computing: Modern Systems and Practices
Elsevier publishers ISBN: 978-0-12-420158-3

Jason Sanders & Edward Kandrot (2011)
CUDA by Example: An introduction to General-Purpose GPU programming
Addison-Wesley ISBN-10: 978-0-13-138768-3

List of Nvidia graphics processing units (cited 2021)
https:\\en.wikipedia.org\wiki\List_of_Nvidia_graphics_processing_units

GPU Memory Latency's Impact and Updated Test (cited 2021)
https:\\chipsandcheese.com\2021\05\13\gpu-memory-latencys-impact-and-updated-test

- overview image is personal redraw
- until next time image [Cit. 02.09.2021]. Available from https:\\i.chzbgr.com\full\9591931648\hDC02ACF7\person-my-hacky-program-cpu-o-other-7-processor-cores-my-rtx-3090

Practical session

1. start/setup your development tool
2. get the source code
3. follow the **instructions in code**
   - if unsure → first think about it
   - if still lost → "google" it
   - if can't find → ask about it *(personally or mail)*
4. make sure your code **compiles**
5. make sure your code **works**
6. send your code to me *(celnyd@vscht.cz)*

Until next time