GPGPU Programming

Donato D'Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy
mailto: donato.dambrosio@unical.it
homepage: http://www.mat.unical.it/~donato

Academic Year 2020/21

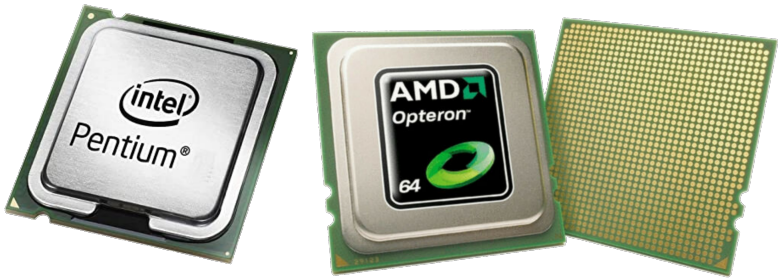
Table of contents

Heterogeneous Parallel Computing

Heterogeneous Parallel Computing

Heterogeneous Parallel Computing

- Old microprocessor based on a single central processing unit (CPU) were able to reach GFLOPs[1] to the desktop and TFLOPs[2] to datacenters.



- Since year 2003, due to energy consumption and heat dissipation issues, vendors have switched to multi-core architectures to increase the processing power.
- As a consequence, parallel programming has become necessary, with multiple threads of execution cooperating to complete the work faster (concurrency revolution).

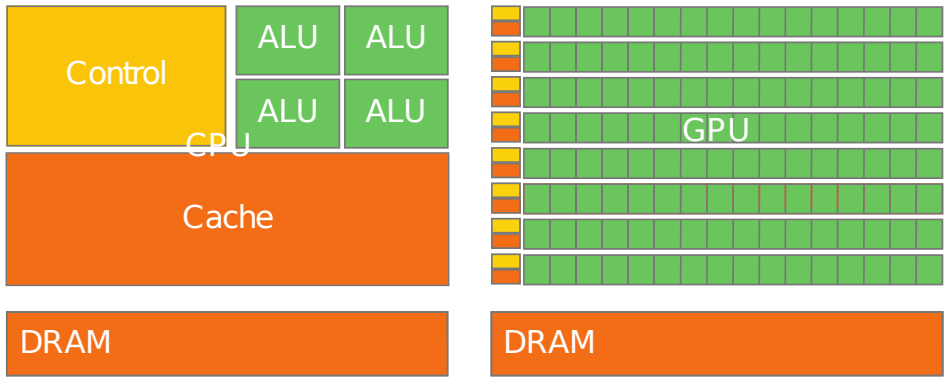
[1] GFLOPs: Giga Floating-Point Operations per Second
[2] TFLOPS: Tera Floating-Point Operations per Second

Heterogeneous Parallel Computing

- Actually, since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessors:
  - The multi-core trajectory (tens of cores) seeks to maintain the execution speed of sequential programs while moving into multiple cores.
  - In contrast, the many-core, or many-thread trajectory (thousands of cores) focuses more on the execution throughput of parallel applications.
- Current many-core processor are more than 10X faster than multi-core sulutions.
  - These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips[3].

[3] https://en.wikipedia.org/wiki/Graphics_processing_unit

Heterogeneous Parallel Computing



- CPU are optimized for sequential code performance.
  - Sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution.
  - Large cache memories (many megabytes) to reduce the instruction and data access latencies of large complex applications.
- Graphics chips have been operating at approximately 10x the memory bandwidth of CPUs.
  - A GPU must be capable of moving extremely large amounts of data in and out of its main Dynamic Random Access Memory (DRAM).

Heterogeneous Parallel Computing

- GPUs hardware takes advantage of a large number of threads to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations.
- Small cache memories are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM.
- It should be clear now that GPUs are designed as parallel, throughput-oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well.
  - For programs that have few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs.
  - When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs.

Heterogeneous Parallel Computing

- Applications can use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.
- Until 2006, General-Purpose Programming using a GPU (GPGPU) was very difficult since the OpenGL or Direct3D graphics APIs were the only option to program theese devices
- The Nvidia CUDA programming model, as well as the Khronos OpenCL API, besides others, support heterogeneous computing on CPU/GPU systems.
- A typical CUDA-capable GPU is organized into an array of highly threaded streaming multiprocessors (SMs).

Architecture of a Modern GPU

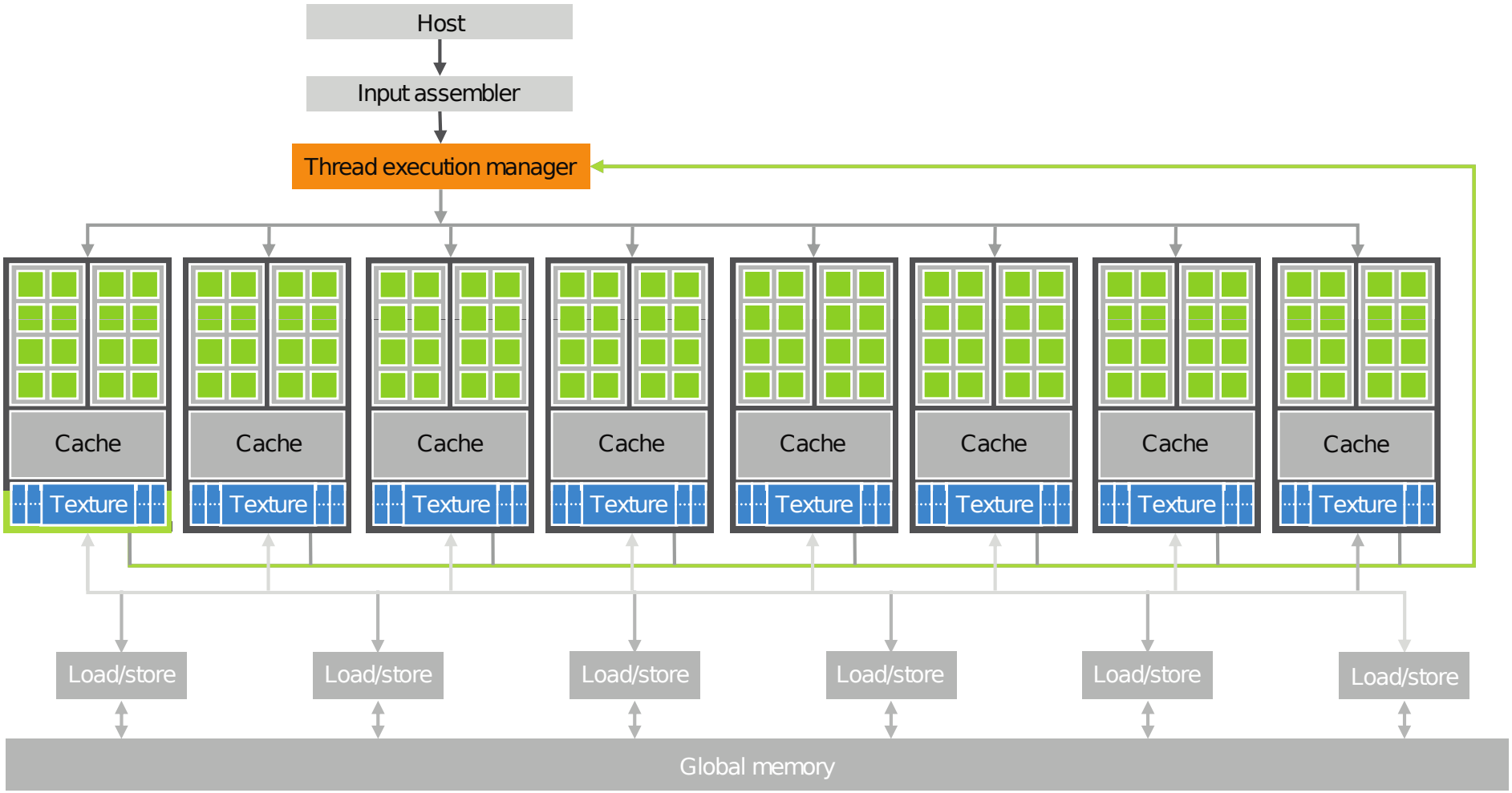


Figure: High level view of the architecture of a typical CUDA-capable GPU

Architecture of a Modern GPU

- Each SM has a number of streaming processors (SPs) that work in lock-step sharing control logic and instruction cache (as we will see, this execution model can limit performance in case of selection instructions...).
- GPUs come with gigabytes of Graphics Double Data Rate (GDDR), Synchronous DRAM (SDRAM), called Global Memory.
  - For graphics applications, they hold video images and texture information for 3D rendering (frame buffer).
  - For computing, they function as very high-bandwidth off-chip memory, though with somewhat longer latency than typical system memory.
- For massively parallel applications, the higher bandwidth makes up for the longer latency.
- Recent products may use High-Bandwidth Memory (HBM) or HBM2 architecture (DRAM).

Architecture of a Modern GPU

- CUDA architecture had a communication link to the CPU core logic over the PCI-Express interface.
- PCI-E Gen2 can transfer data from/to the system memory to/from the global memory at 4 GB/s (for a total of 8 GB/s).
- More recent GPUs use PCI-E Gen3 or Gen4, which supports 8–16 GB/s in each direction.
- Starting from the Pascal family GPUs also supports NVLINK, a CPU–GPU and GPU–GPU interconnect that allows transfers of up to 40 GB/s per channel.
- As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E or NVLINK.

Architecture of a Modern GPU

- A GPGPU application must run thousands and thousands of threads simultaneously to take advantage of the high number of SPs.
- Such an approach can also make up the global memory latency.
- If the application contains what we call data parallelism, it is often possible to achieve a 10X speedup over sequential execution on a single CPU core with just a minimun effort.
- When an application is suitable for parallel execution, a good GPGPU implementation can achieve a speedup of more than 100X.

Architecture of a Modern GPU

- Some examples of application that are suitable for parallel execution are:
  - Neural Networks (especially in the learning process)
  - Simulation of complex phenomena in Natural Sciences (Physics, Chemistry, Biology and Medicine, Geology, and so on)
  - High-resolution Audio/Video encoding/deconding
  - Simulation of Physics in Video Games
- All these applications usually have massive amounts of data being processed (the Big Data has become a household phrase).
  - Much of the computation can be done on different parts of the data in parallel, although they will have to be reconciled at some point.

Speeding Up Real Applications

- If $p \in [0, 1]$ is the part that can be parallelized, and s the achieved speed-up of that part, the overall speed-up for the application is given by the Amdahl's law:

$$S_{peed-up} = \frac{1}{(1-p) + p/s}$$

  - $p = 0.5 \ \wedge s = \infty \implies S = 2$
  - $p = 0.9 \ \wedge s = \infty \implies S = 10$
  - $p = 0.99 \wedge s = \infty \implies S = 100$
- Obtaining a 100X speedup or more generally requires sequential code re-organization so that more than 99.9% of the application can be parallelized.
- In practice, straightforward parallelization of applications often saturates the memory (DRAM) bandwidth, resulting in only about a 10X speedup.

Speeding Up Real Applications

- GPU on-chip memories can be used to drastically reduce the number of accesses to the DRAM.
- One must, however, further optimize the code to get around limitations such as limited on-chip memory capacity.
- However, in some applications CPUs perform very well, making it harder to speed up performance using a GPU.
- Most applications have portions that can be much better executed by the CPU. In such a case, code must be written so that GPUs complement CPU execution, thus properly exploiting the heterogeneous parallel computing capabilities of the combined CPU/GPU system.
- Some parallel algorithms can add large overheads over their sequential counter parts so much that they can even end up running slower.

Designing Efficient Parallel Algorithms

- The execution speed of many applications is limited by memory access speed. We refer to these applications as memory-bound, as opposed to compute-bound, which are limited by the number of instructions performed per byte of data.
- Achieving high-performance parallel execution in memory-bound applications often requires novel methods for improving memory access speed.
- Many real world problems are most naturally described with mathematical recurrences. Parallelizing these problems often requires nonintuitive ways of thinking about the problem and may require redundant work during execution.
- Fortunately, most of these challenges have been addressed. There are common patterns across application domains that allow us to apply solutions derived from one domain to others.

Parallel Programming Languages and Models

- MPI (Message passing interface) for scalable cluster computing.
- OpenMP for shared memory multiprocessor systems.
  - An OpenMP implementation consists of a compiler and a runtime.
  - Directives (commands) and pragmas (hints) about a loop generate parallel code transparently.
  - The runtime system supports the execution of the parallel code by managing parallel threads and resources. OpenMP was originally designed for CPU execution.
  - From release 4, OpenMP supports parallel execution on GPUs.
  - A variation called OpenACC has been designed to support heterogeneous parallel execution.
- Both have become standardized programming interfaces supported by major computer vendors.

Parallel Programming Languages and Models

- As for CPU and GPU communication, CUDA previously provided very limited shared memory capability.
- New runtime support for global address space and automated data transfer in heterogeneous computing systems (CUDA Unified Memory).
  - The runtime hardware and software transparently maintains coherence by automatically performing optimized data transfer operations on behalf of the programmer as needed.
- OpenCL is similar to CUDA in many respects. It is mainly an API (not a set of compiler extensions like CUDA), it is open (it is managed by the Khronos Group consortium), supported by virtually all vendors and runs on virtually all computing devices. It is a bit more difficult to learn with respect to CUDA. Nevertheless, a CUDA programmer can easily learn OpenCL since the key-concepts are the same.