GPU as a Hardware Accelerator

Beomki Yeo

UC Berkeley and LBNL

Computational HEP Traineeship

Summer School 2024

May 20th, 2024

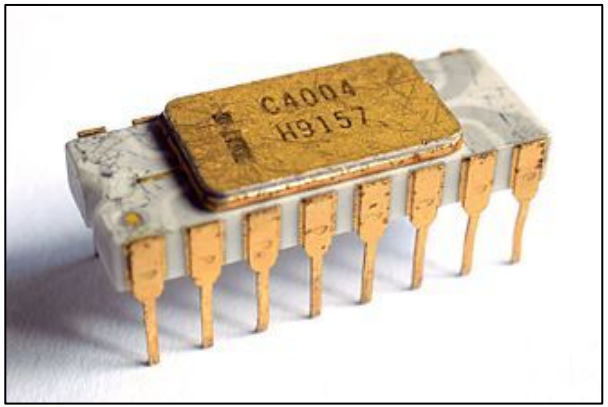
Outline

1. Evolution of Microprocessor Design

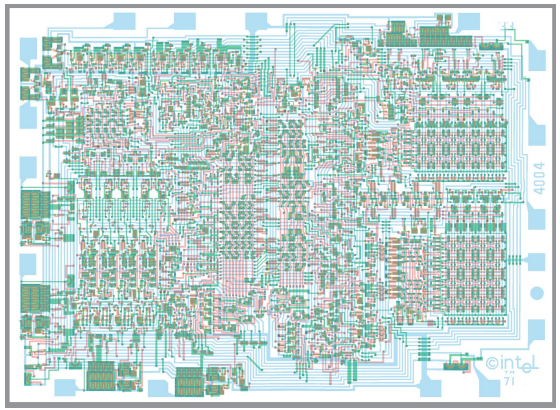
2. GPU as a Hardware Accelerator
   a. Difference with CPU
   b. Basic mechanisms and how-to-optimize

Microprocessor

- Microprocessor: A single Integrated Circuit which can do data processing and logic control
- Integrated Circuit: A chunk of transistors
- Transistor: A minimal building block of electronics



Intel 4004 (1971), one of the first microprocessors



Intel 4004 chipset design (2300 transistors)

CPU: Traditional Microprocessor

- **CPU (Central Processing Unit)**
  - Made of Cores, Caches and Control Units
- **Core**: Algorithm Logical Units (ALUs) and registers
  - **ALU**: performs mathematical operations
  - **Register**: small storage which stores data being processed
- **Cache**: On-chip memory
- **Control Unit:** Distribute operations to other units

Intel Core i9-14900K

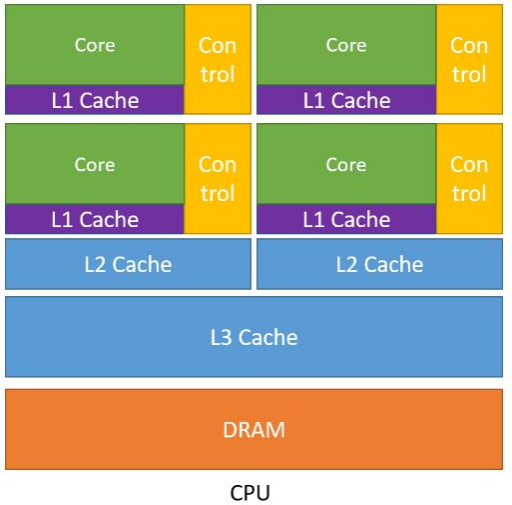


Image credit

Evolution of CPU Design in the Past

- Transistor gets smaller **(Moore's law)** → More power-efficient
  - This gives rooms to CPU architects to improve the chipset design
- Advance in core microarchitecture design
  - Branching precision, out-of-order execution, cache memory etc.
- Cache memory hierarchy with multiple levels
  - Solving the memory bottleneck

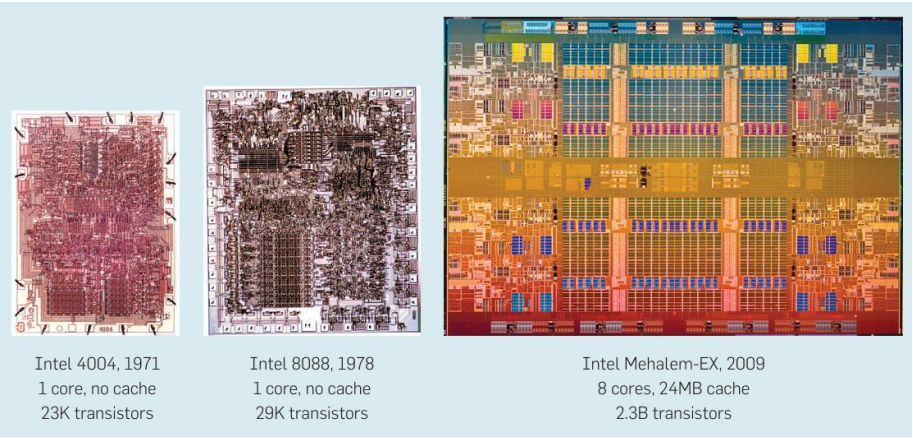
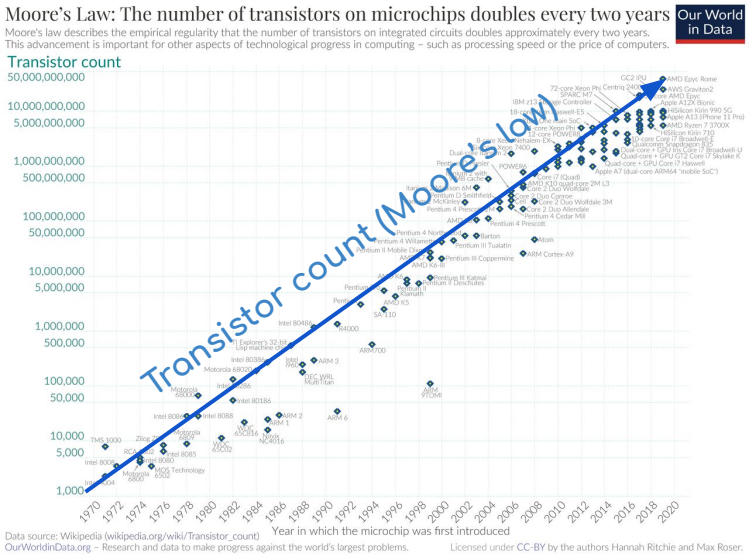


Image credit

Moore’s law

- An observation that the number of transistors in processors doubles every two years
  - The dimension of transistor is reduced by √2
- Smaller transistors *used to* increase the performance of CPU



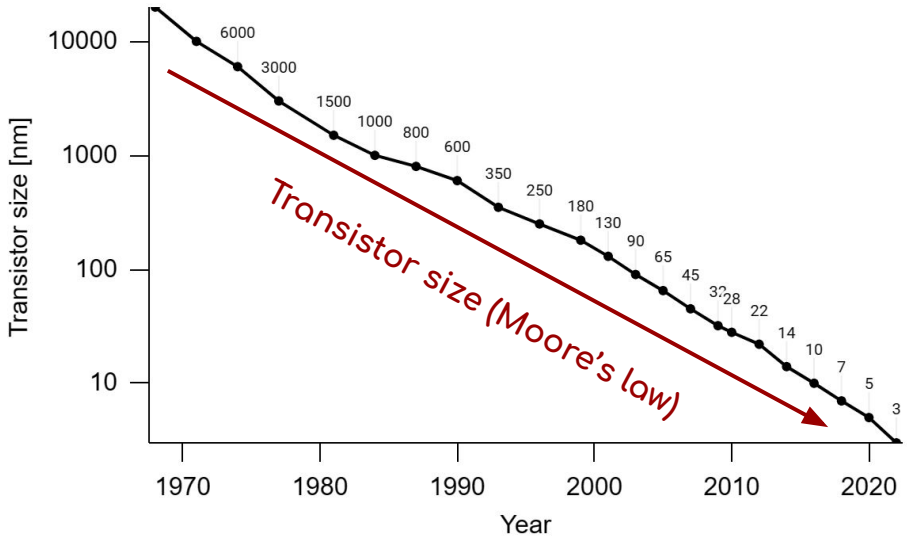


Image credit

Data from

Why were Small Transistors Supposed to be Good?

- Dennard Scaling
  - **Free scaling of the frequency (f)** for the same power consumption (P)
  - $P = \alpha C V^2 f$
  - Capacitance (C) and operating voltage (V) are linearly reduced with the size of transistor

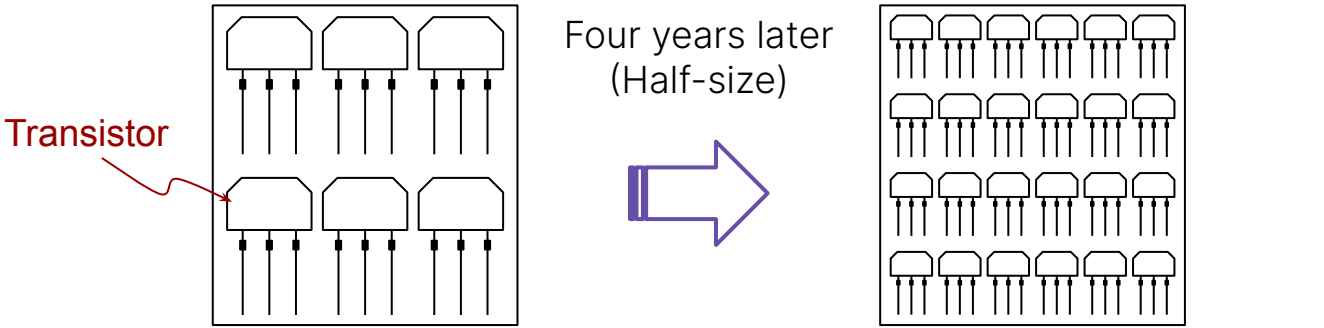
- Pollack's rule
  - **Observation of Performance** $\propto \sqrt{N}$ (N = the number of transistors)
  - Moore's law allows more number of transistors (N) on the same chip size

N.B. These days, Dennard scaling is dead and the actual gains from the Pollack's rule is usually less than its expectation

Ideal Performance Scaling

- Dennard scaling and Pollack's rule orchestrate to scale the performance with the Moore's law

- Ideally, the performance is increased by 4 every four years
    - Factor of 2 from Dennard scaling and Pollack's rule, respectively



$$P = \alpha\, N\, C\, V^2\, f$$

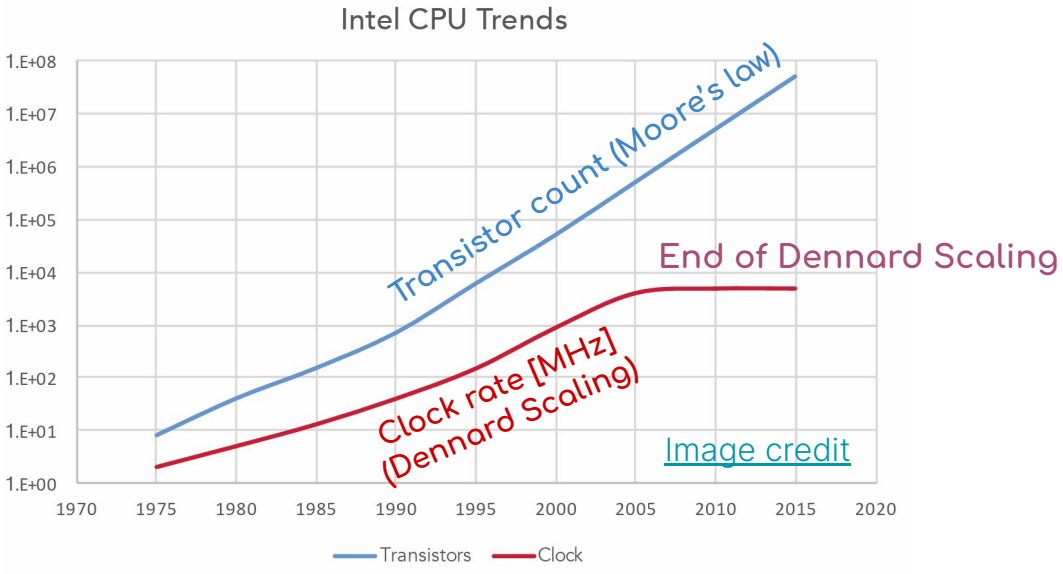
$$P = \alpha\, (4N)\, (C/2)\, (V/2)^2\, (2f)$$

End of Dennard Scaling

- Below the transistor size of 65 nm (since yr. 2005), the current leakage ($I_{leakage}$) is not negligible anymore
  - No free scaling of frequency with the same power cost

$$P = \alpha N C f V^2 + V I_{leakage}$$

- **Power matters** in the chipset design
  - Industries has stopped increasing the clock rate



New paradigm in CPU: Vectorization and Parallelization

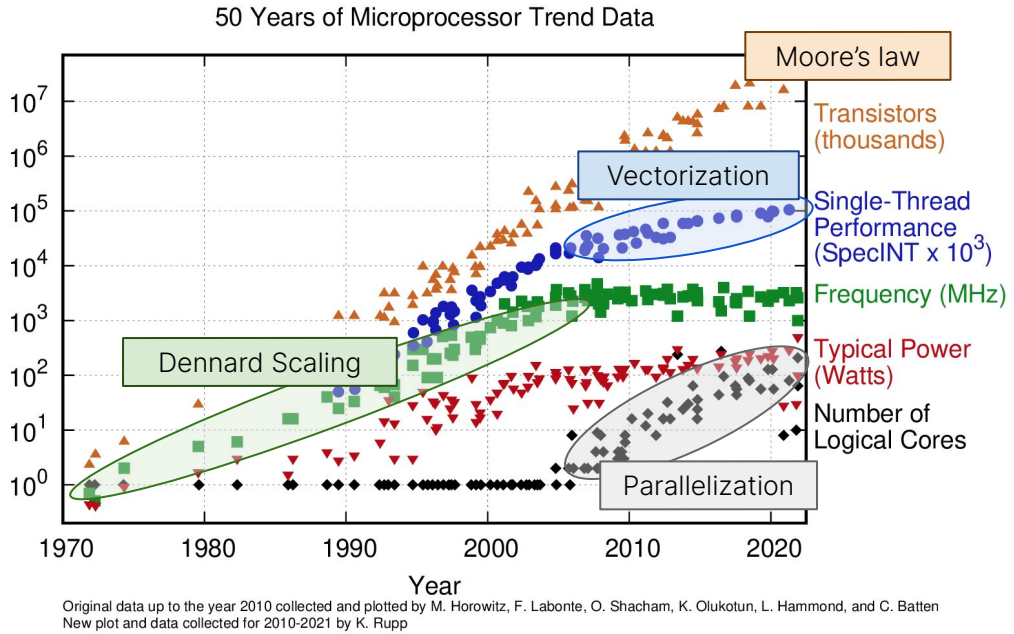


Image credit Slide after

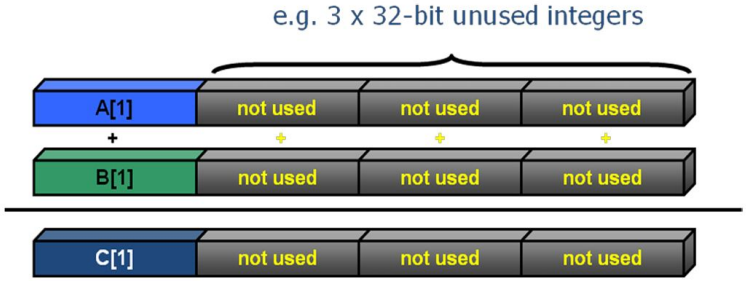
- Recently Industries has improved the performance with extra transistors in different ways
- Increasing the register size
  - **Vectorization**
- Increasing the number of logical cores
  - **Parallelization**

Vectorization

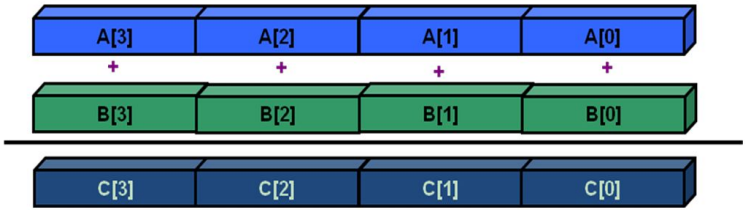
- A larger size of registers allow the computation on multiple data at one time
  - Single Instruction multiple data (**SIMD**)
- If the data is aligned properly, vectorization can be done automatically with modern CPUs

```
For (i =0; i < 4; i++){
    C[i] = A[i] + B[i];
}
```

Auto-vectorization example of four-vector addition



Auto-vectorization OFF



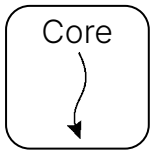
Auto-vectorization ON

Image credit

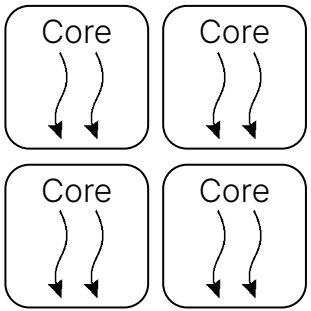
Parallelization

- Ideally, performance will linearly increase with the number of logical cores running in parallel
- In the programming side, a core can be operated with multiple threads
  - **Thread:** A virtual sequence of instructions
  - Single thread per core may not be enough to draw full performance of the core

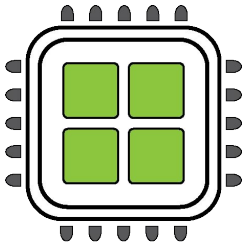
Single core with single thread



multiple cores with multiple threads

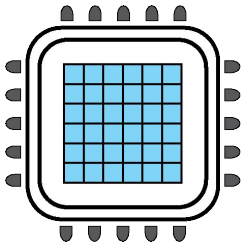


Hardware Accelerators beyond CPU



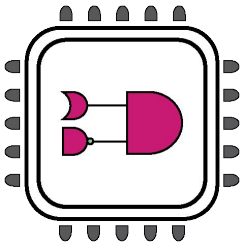
CPU

Central
Processing Unit



GPU

Graphics
Processing Unit



FPGA

Field Programmable
Gate Array

- GPU with extreme **parallelization**
  - Lots of cores
- FPGA with extreme **customizability**
  - Chipset design tailored for a specific algorithm
  - Highest potential but with a high learning curve

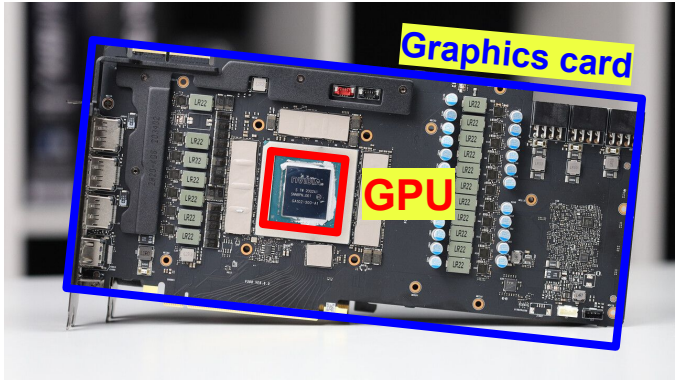
In the today's talk, **GPUs** will be highlighted but FPGA-based acceleration is also actively investigated in HEP computing

Introduction to GPU

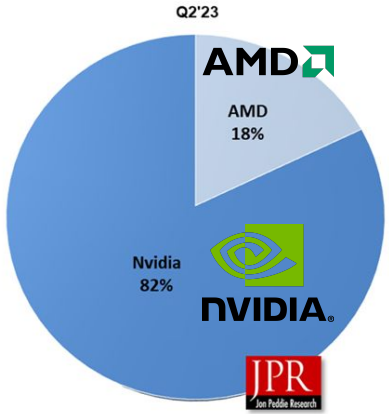
- Graphical Processing Unit
  - a.k.a graphics card
  - Major Vendors: NVIDIA, AMD, and Intel

- Invented for the visualization on PC screen
  - Now also used as a hardware accelerator

- Two types of GPU
  - CPU-integrated
  - Discrete GPU



Discrete GPU



Discrete GPU Market Share

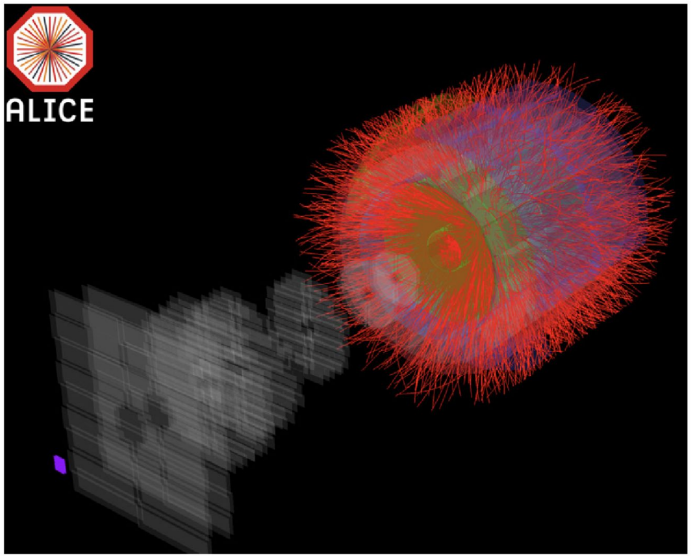
General Purpose GPU

- Use GPUs also for general purpose computation
- Machine learning in GPUs
  - The 1st generation of AlphaGo
  - Physics analysis of HEP experiments
- Any parallelizable algorithm in HEP computing
  - Online track reconstruction of the ALICE and LHCb experiments
  - Many ongoing R&D projects for simulation and analysis

4.1.2 General-Purpose Computation on GPUs

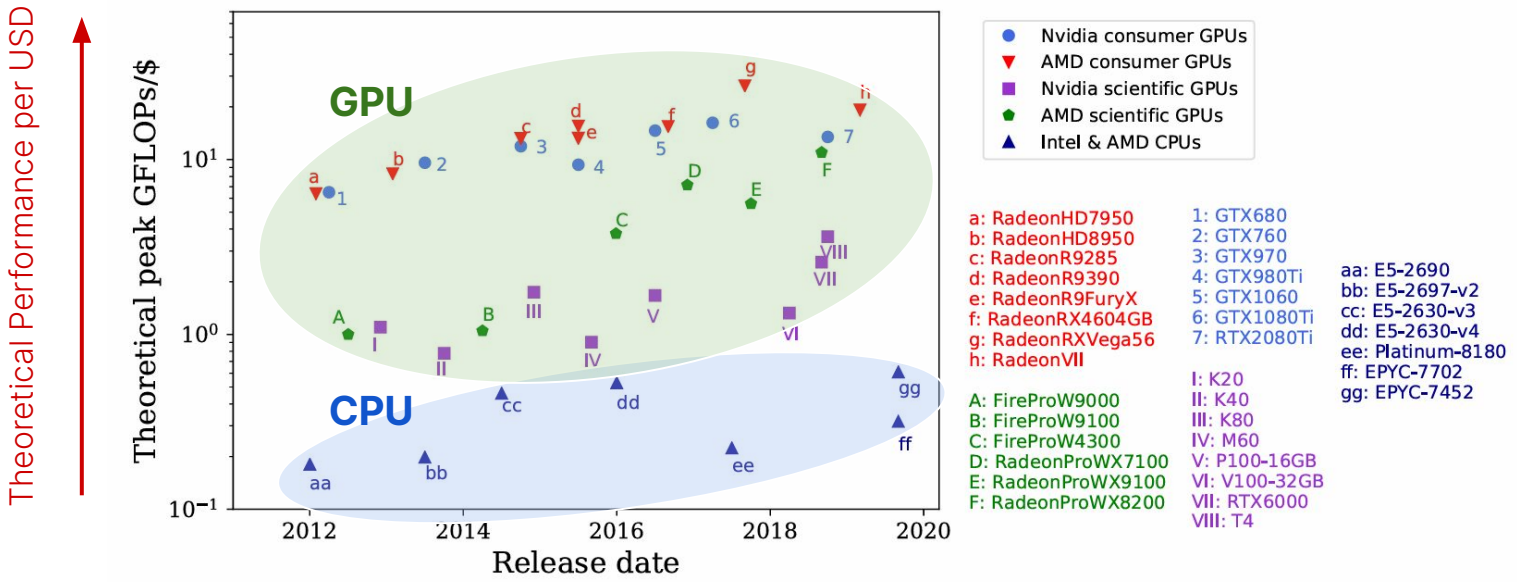
The use of computer graphics hardware for general-purpose computation has been an area of active research for many years, beginning on machines like the Ikonas (England, 1978), the Pixel Machine (Potmesil and Hoffert, 1989), and Pixel-Planes 5 (Rhoades et al., 1992). The wide deployment of GPUs in the last several years has resulted in an increase in experimental research with graphics hardware. Trendall and Steward give a detailed summary of the types of computation available on modern GPUs (Trendall and Steward, 2000).

GPGPU Coined by M. Harris (2003, PhD thesis)



ALICE Event Reconstructed by GPU

CPU vs GPU: Performance per **USD**



arXiv:2003.11491

Theoretically GPU is faster and more economical than CPU.
(Of course, there is a trade-off to achieve this)

CPU vs GPU: Performance per **Watt**

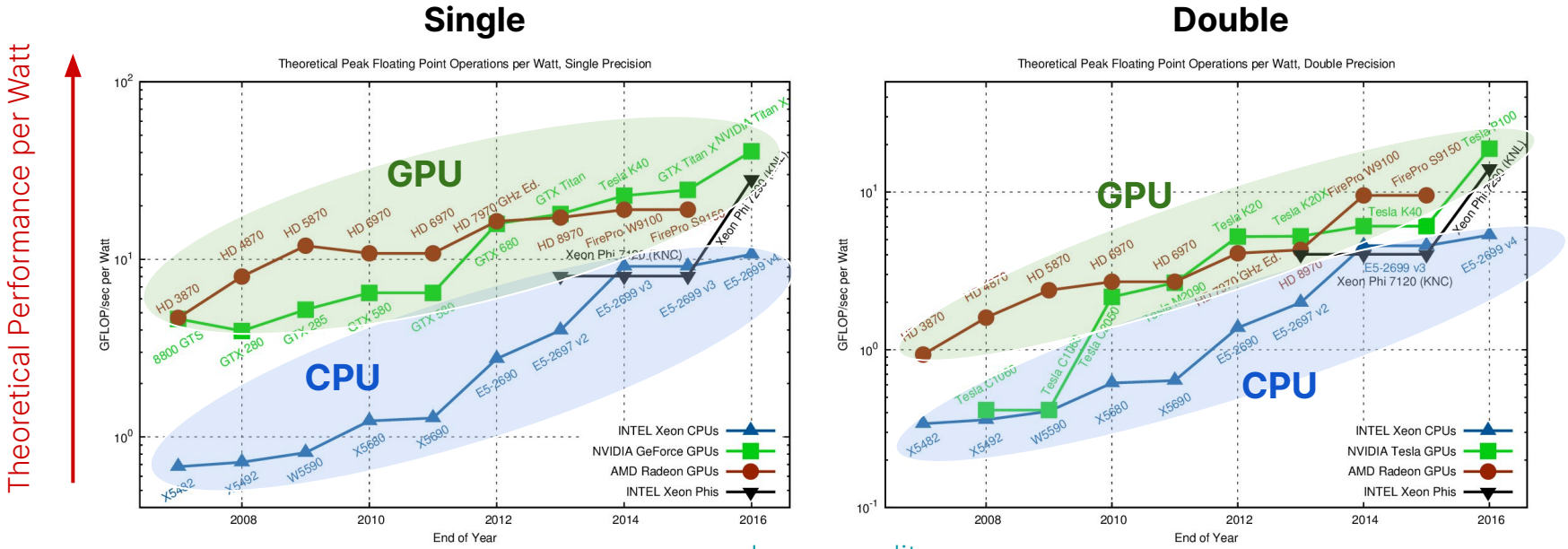


Image credit

Theoretically GPU is faster and more economical than CPU.
(Of course, there is a trade-off to achieve this)

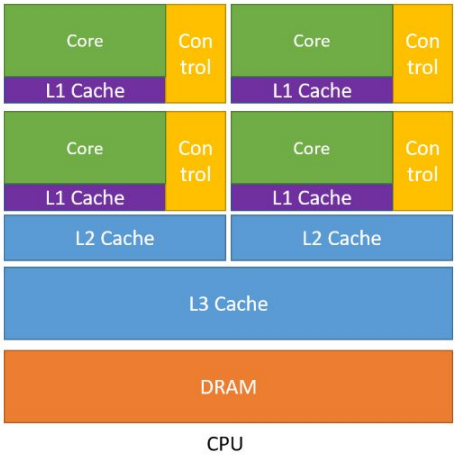
CPU vs GPU: Chip Structure

CPU

- Small number of powerful cores (~10)
  - Branch prediction, out-of-order execution, etc.
- Large caches
- Single instruction on multiple data (**SIMD**)

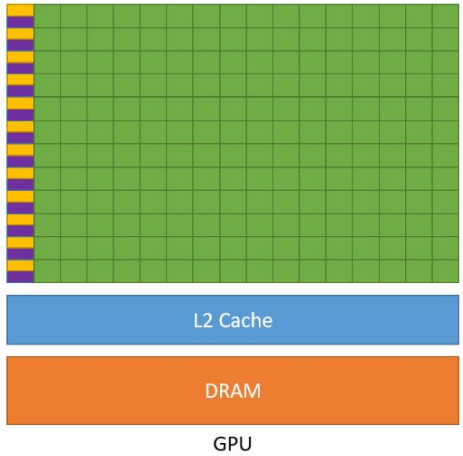
GPU

- Many number of cores (≳1000)
  - *A lot* simpler
- Small caches
- Single instruction on multiple threads (**SIMT**)



CPU

Image credit



GPU

Practically GPU can outperform CPU with parallelizable and relatively simple algorithms

Example of Hardware Specifications

	CPU	**GPU**
Chipset Model	**AMD** Ryzen Threadripper PRO 7995WX	**NVidia** GH100
Launch Date	Oct. 2023	Mar. 2022
Transistor Size	5 nm	5 nm
Number of Transistors	78,840 million	80,000 million
Clock Frequency	2.5 GHz	1.095 GHz
Cores	**96**	**14592**
Price	~10000 $	~30000 $ (Graphics card price)
TDP	350 W	350 W
Peak Performance (float)	**12.2 TFLOPS**	**51.22 TFLOPS**

SIMD and SIMT

```
For (i =0; i < 4; i++){
    C[i] = A[i] + B[i];
}
```

Register			
Data Slot 0	Data Slot 1	Data Slot 2	Data Slot 3
A[0] + B[0]	A[1] + B[1]	A[2] + B[2]	A[3] + B[3]
C[0]	C[1]	C[2]	C[3]

CPU SIMD (single thread, auto-vectorization) example of four-vector addition

```
C[threadIdx.x]
  = A[threadIdx.x] + B[threadIdx.x];
```

Thread			
Thread 0	Thread 1	Thread 2	Thread 3
A[0] + B[0]	A[1] + B[1]	A[2] + B[2]	A[3] + B[3]
C[0]	C[1]	C[2]	C[3]

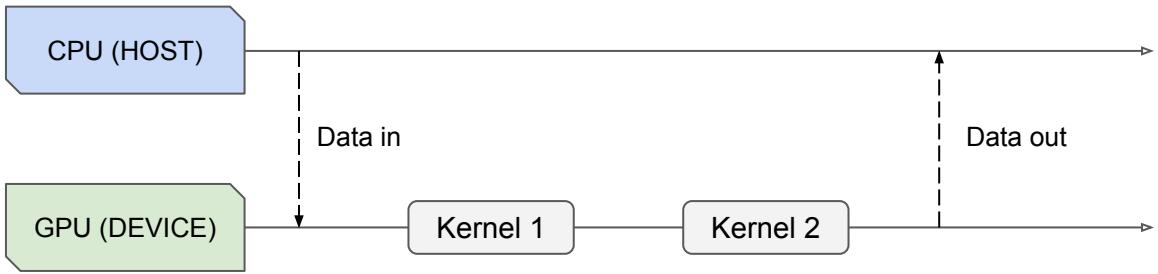
GPU CUDA SIMT (multiple threads) example of four-vector addition

GPU Programming Workflow

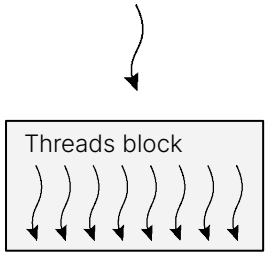
The code is always executed from the CPU (HOST) side

1. Transfer data from **CPU** to **GPU**
2. Launch a **GPU** kernel function with a dimension of thread block
3. Transfer data from **GPU** to **CPU**

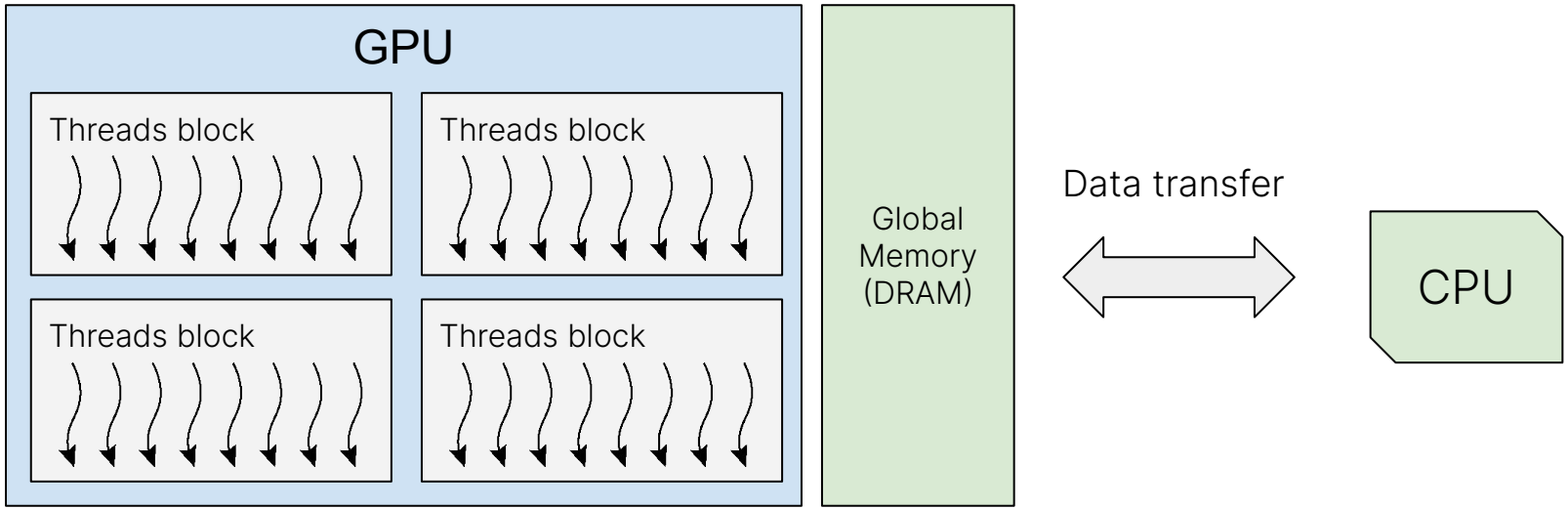
N.B. The kernel launches are not free and can take 5 to 100 μs depending on hardware, driver, etc.



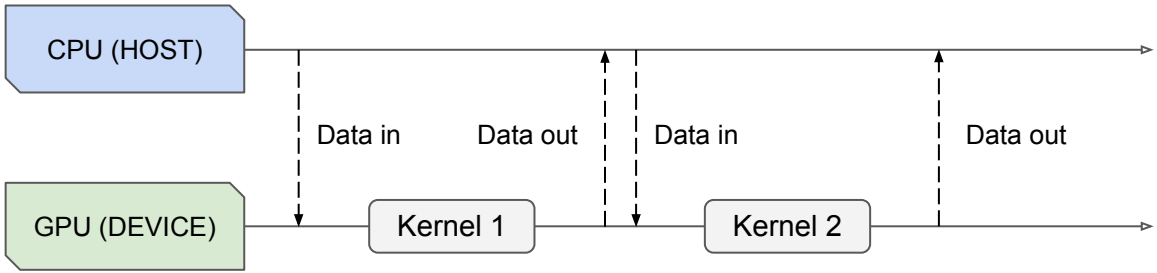
GPU Parallelization Structure

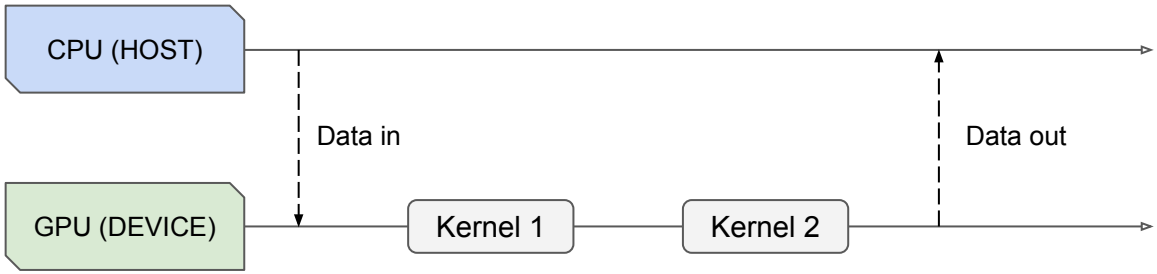


- **Thread:** A virtual sequence of instructions, similar to the CPU thread
- **Thread block**: A cooperative set of threads



GPU Optimization – Minimization of Data Transfer

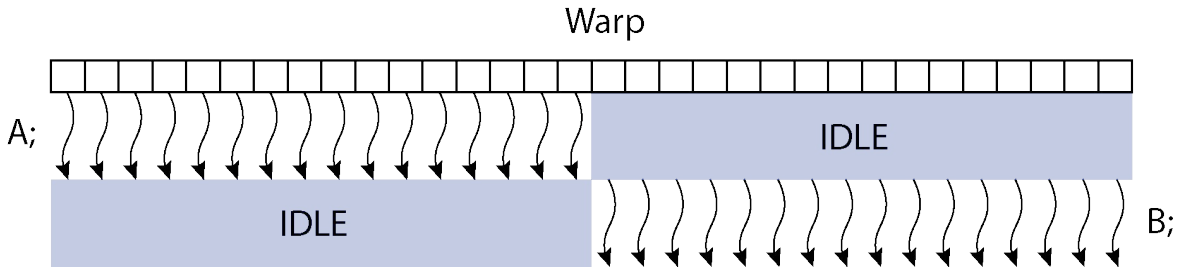




GPU Optimization – Avoid Thread Divergence

- In GPUs, threads in a set can only do the same thing at one time.
  - An important feature of Single instruction Multiple Thread (**SIMT**)
  - NVIDIA **Warp**: 32 synchronized threads
  - AMD **Wavefront**: 64 synchronized threads

- Following diagram shows the half threads in a warp trying to do a different task resulting into **loss of 50% efficiency**
  - The efficiency loss is minimized by avoiding the thread divergence




```
if (threadIdx.x < 16){
        A;
}
else {
        B;
}
```

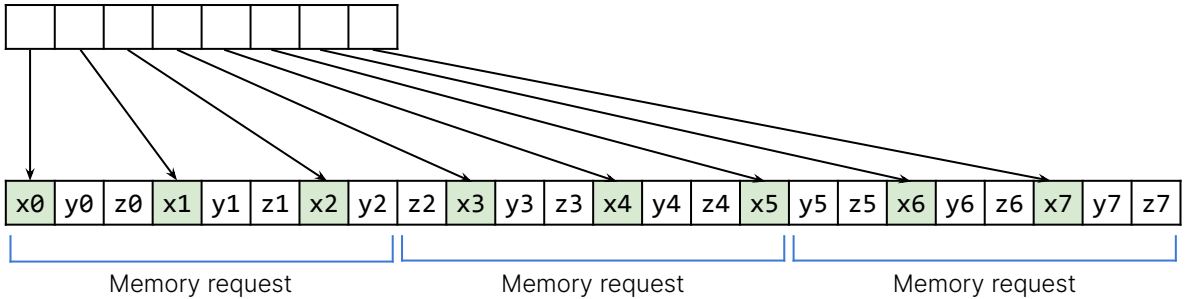
GPU Optimization – Vectorized Memory Access

- Threads in a warp can access global memory with the least amount of transactions when the data is vectorized (coalesced, precisely speaking)
  - **Struct-of-Array (SoA)** memory layout is more GPU-friendly in many cases

Array-of-Struct (AoS)

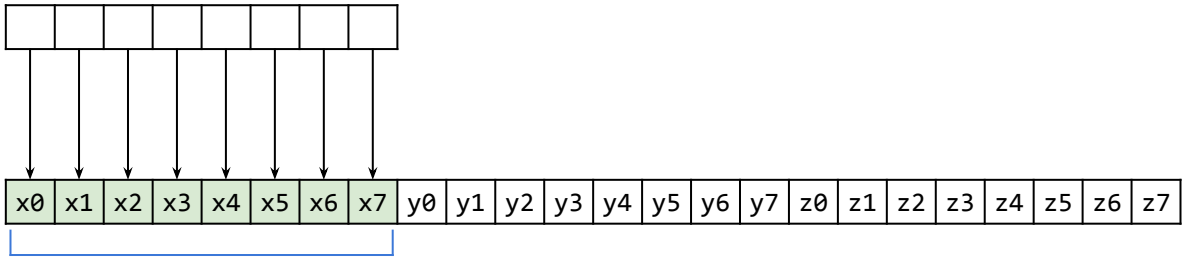
```
struct float3{
 float x,y,z;
};

float3 r[8];
```

8 threads in a warp trying to access **x** data



Struct-of-Array (SoA)

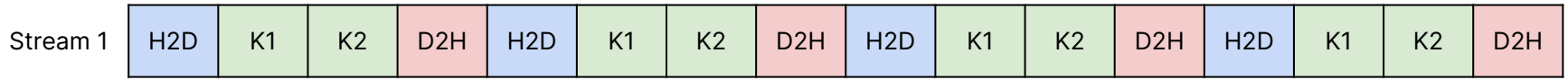
```
float x[8];
float y[8];
float z[8];
```



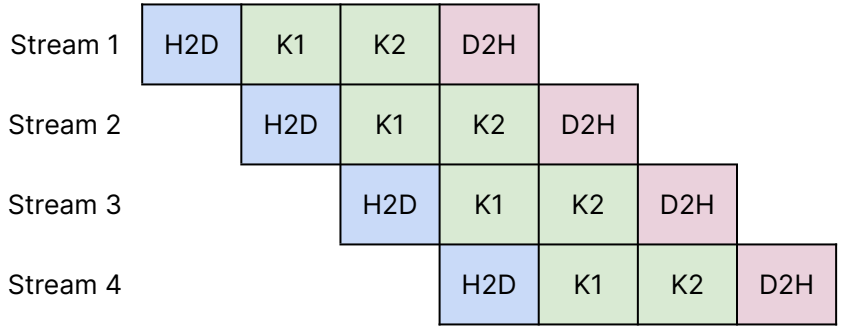
GPU Optimization – Multiple (Asynchronous) Kernels

- A single GPU kernel may not utilize the entire resource of GPUs
  - Running multiple GPU kernels with `streams` (multiple CPU threads) can be beneficial
  - Kernels and memory transfers can be overlapped

Single stream (single CPU thread)

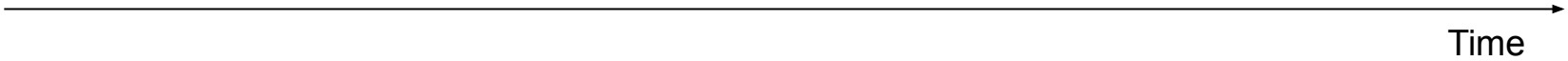


Four streams (four CPU threads)



Definitions

H2D	Host to Device transfer
K1	Kernel 1
K2	Kernel 2
D2H	Device to Host transfer



(Non-Exhaustive) List of Optimization Strategies

- Do everything in GPU and never come back to CPU
  - To minimize the data transfer between CPU and GPU
- Keep all threads of a warp busy (high warp efficiency)
  - Balancing the workloads for each thread is important
- Make threads of a warp access global memory in a coalesced manner
  - Memory layout of Struct-of-Array (SoA) can be beneficial
- Keep entire device busy
  - Launching multiple kernels concurrently can be helpful
- Put large data into a kernel to reduce the number of kernel launches
- No dynamic allocations (e.g. `malloc` or `new`) in kernels

GPU Programming Models

- **C++**
  - CUDA: Low-level language for GPU, *exclusive to NVIDIA*
  - SYCL: Stemmed from OpenCL, compatible with all vendors
  - HIP
- **Python**
  - Numba: CUDA Implementation in Python

	C++			**Python**
	CUDA	**SYCL**	**HIP**	**Numba**
NVIDIA				
AMD				Deprecated
Intel			Prototype	?

Support

No Support

Portability Layers for GPU

Programming Models for **single source-code** for both CPU and GPU

1. **Alpaka** and **Kokkos**: Abstraction layers on (Native) languages
2. **`std::par`**: c++ standard library for parallel execution
3. **OpenMP**: directive (e.g., c++ macro) based GPU execution

	CUDA	Kokkos	SYCL	HIP	OpenMP OpenACC	alpaka	std::par std::exec
NVIDIA GPU				*hipcc*	*nvc++ LLVM, Cray GCC, XL*		*nvc++*
AMD GPU	*ZLUDA prototype*		*openSYCL intel/llvm*	*hipcc*	*AOMP LLVM Cray*		*AdaptiveCpp ROCm stdpar oneapi::dpl*
Intel GPU	~~*ZLUDA prototype*~~		*oneAPI intel/llvm*	*CHIP-SPV: early prototype*	*Intel OneAPI compiler*	*advanced prototype*	*oneapi::dpl*
x86 CPU	*PGI CUDA for x86 (2010)*		*oneAPI intel/llvm openSYCL*	*via HIP-CPU Runtime*	*nvc++ LLVM, CCE, GCC, XL*		
FPGA				*via Xilinx Runtime*	*prototype compilers (OpenArc, Intel, etc.)*	*protytype via SYCL*	

C. Leggett (ACAT 2024)

How to Get Started with GPU Programming

- C++ programming models are recommended
- If you are a beginner, start with **CUDA**
  - Most standard GPU language
  - Useful libraries (e.g. **Thrust**)
  - Many tutorials in the web
  - Best profiling and debugging tools
- The optimization strategies are universal so the transition to different languages will be relatively easy if the CUDA version is available

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

image credit

Sorting a vector in the device using CUDA Thrust

Summary

- Microprocessor has evolved to improve the performance with the limited power budget
  - In the past, Moore's law guaranteed the free scaling of performance
  - The free lunch is over
  - The chipset architects need to reconsider the chipset design to maximize the throughput

- The GPU is the result of the maximization of the parallelization in the microprocessor design
  - The mechanism of GPU SIMT is very different from the CPU SIMD
  - Hence quite different optimization strategy

- There are many programming models or APIs (e.g. CUDA, SYCL, etc) for different purposes and motivations
  - In general, CUDA is recommended for beginners

Further Reads

- NVIDIA CUDA Guide: CUDA Tutorial
- C. K. Koraka, Introduction to GPU programming, HSF-India HEP Software Workshop (2023)
- H. Gray, Novel Computing Hardware in HEP, CHACAL (2024)
- J. Lebar, Bringing Clang and C++ to GPUs, CppCon (2016)
- S. Borkar and A. A. Chien, The future of microprocessors (2011)