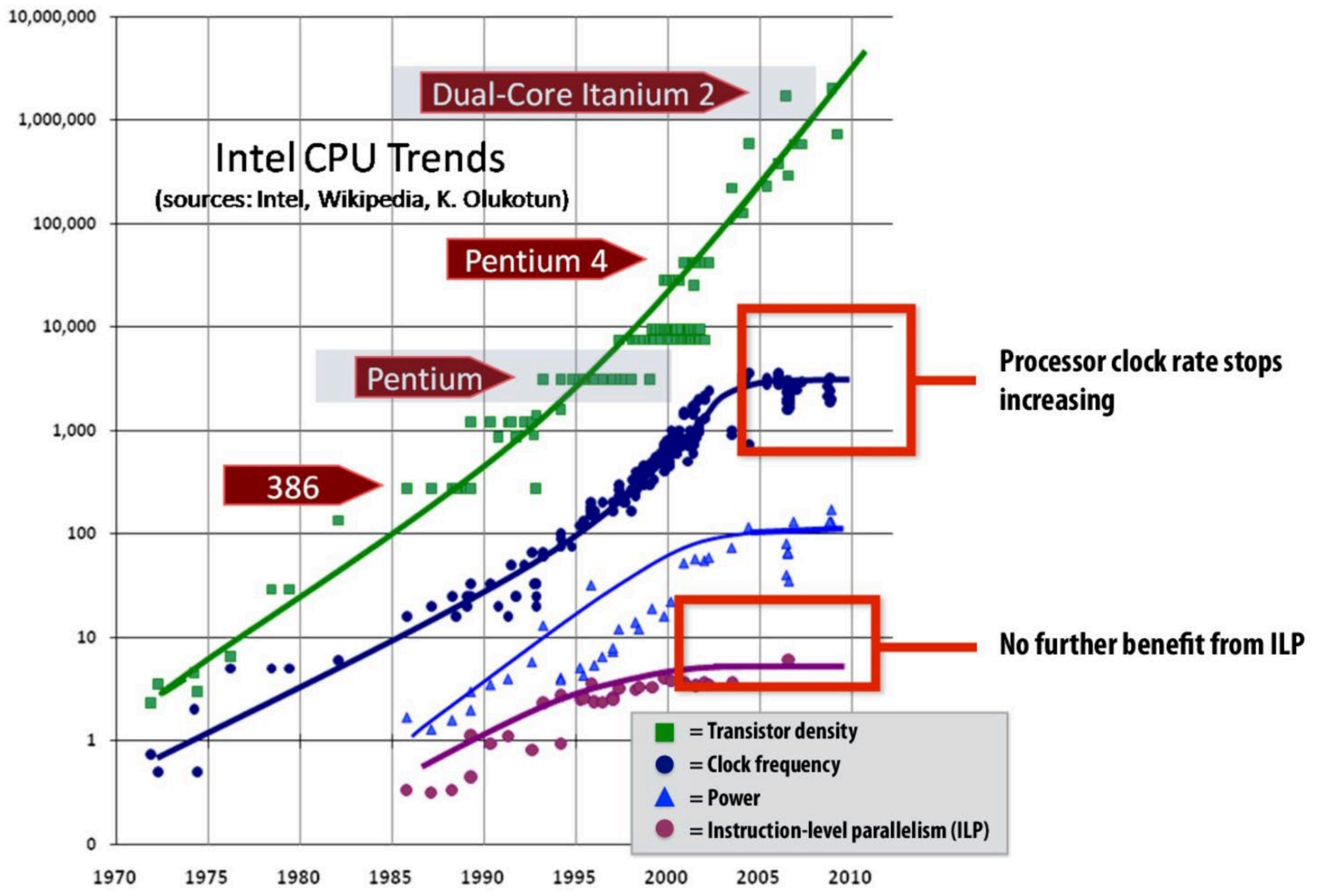
Lecture 1

Why Parallel Programming?

GPU Computing

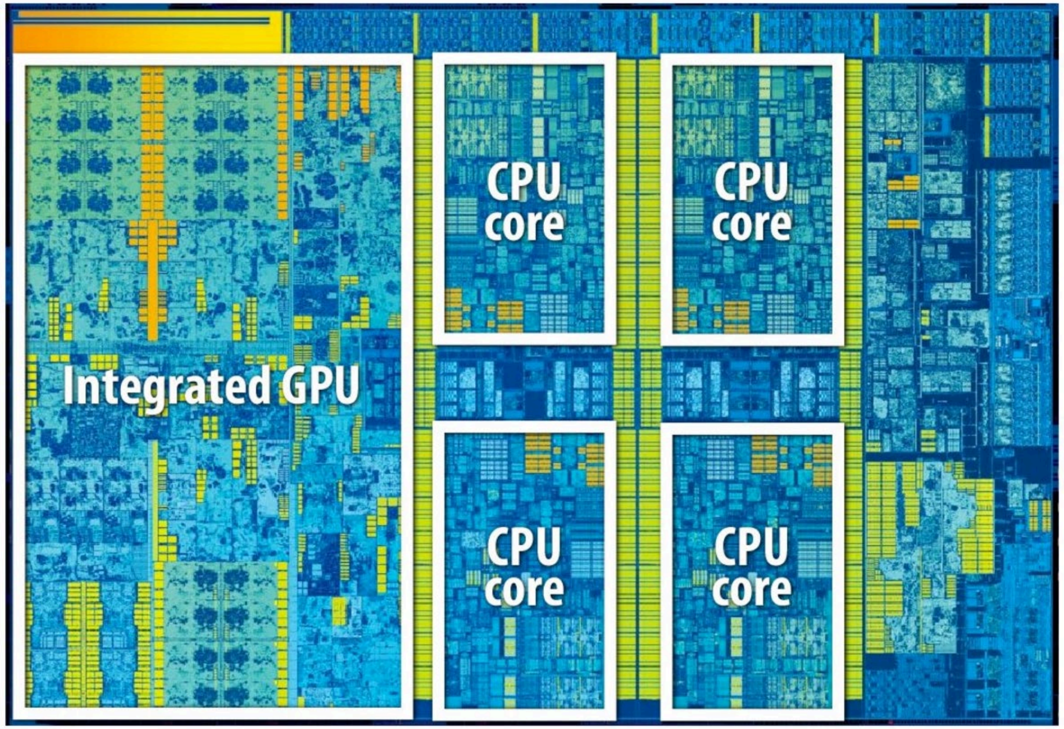
The Free Lunch is Over

- The rate of single-instruction stream performance scaling has decreased (almost to zero)
  - Frequency scaling limited by power
  - ILP scaling tapped out
- Architects are now building faster processors by adding more execution units that run in parallel
- Software must be written to be **parallel** to see performance gains. No more free lunch for software developers!



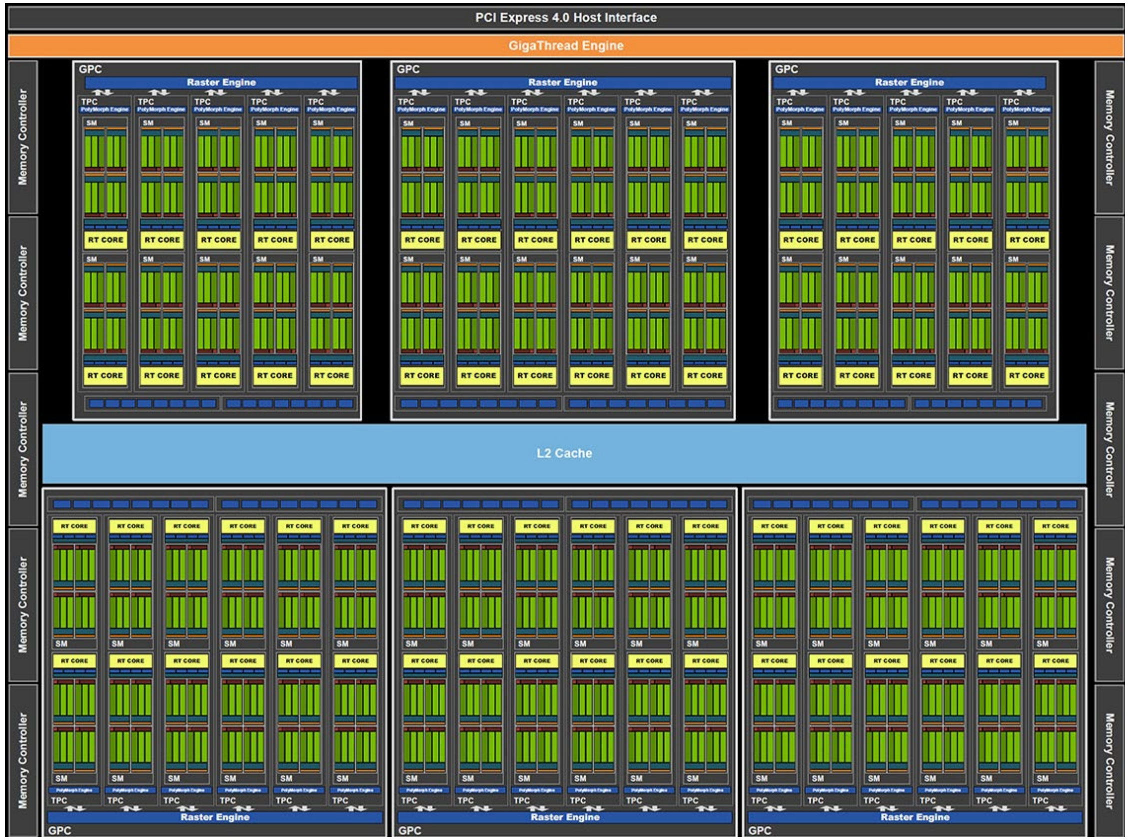
Intel Skylake (2015) —— 6th generation Core i7

- Quad-core CPU + multicore GPU integrated on one chip



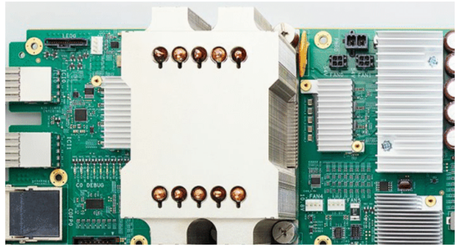
NVIDIA RTX 3080 (2020)

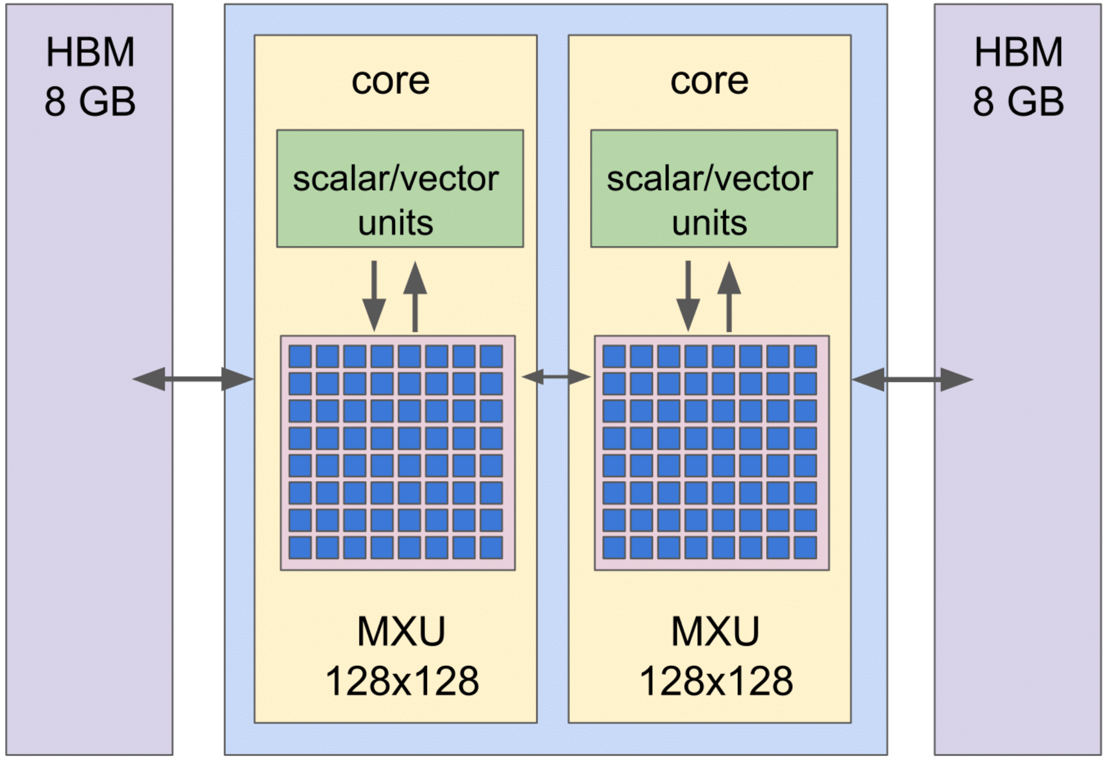
- 68 SMs, 8704 CUDA Cores, 30TFLOPS, 760 GB/s



Google Tensor Processing Unit (TPU)

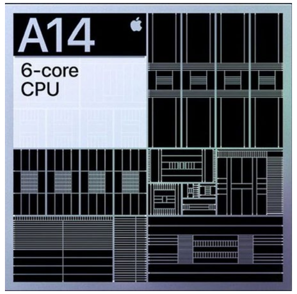
TPUv2 Chip



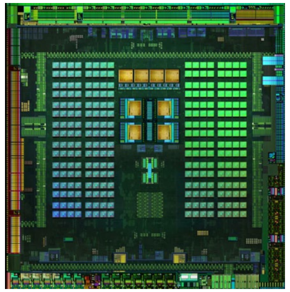


- 16 GB of HBM
- 600 GB/s mem BW
- Scalar/vector units: 32b float
- MXU: 32b float accumulation but reduced precision for multipliers
- 45 TFLOPS

Mobile Computing & Supercomputers



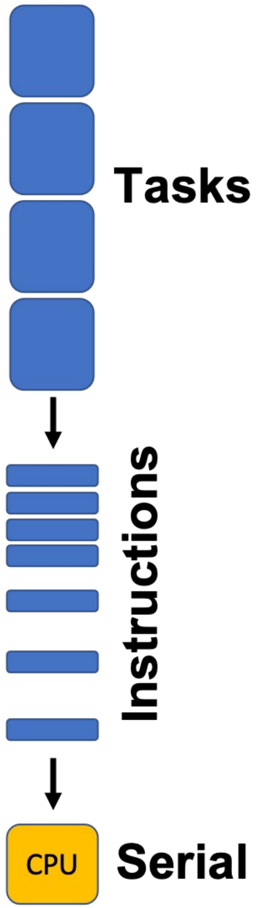
Apple A14



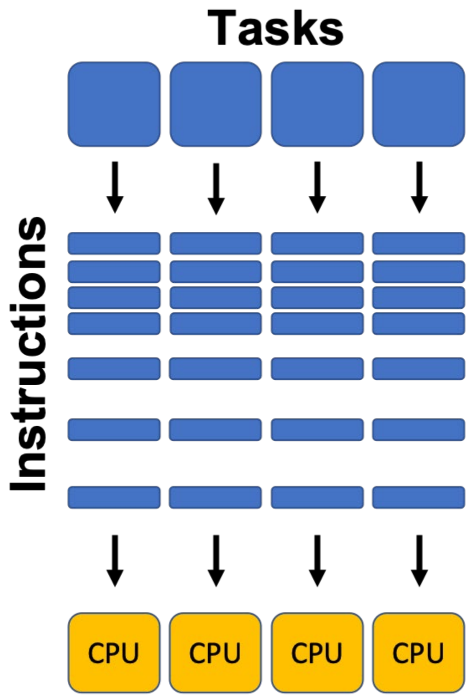
NVIDIA Tegra

Summit Supercomputer

Serial Computing vs. Parallel Computing



Serial computing



Parallel computing

Parallel Programming on CPU

- **EXAMPLE:** Generate 10,000,000 random numbers between 0 and 10, and square the number. Store the results in a list

Serial version

```python
import numpy as np
import time

def random_square(seed):
    np.random.seed(seed)
    random_num = np.random.randint(0, 10)
    return random_num**2
```

```python
t0 = time.time()
results = []
for i in range(10000000):
    results.append(random_square(i))
t1 = time.time()
print(f'Execution time {t1 - t0} s')
```

```
Execution time 32.9464430809021 s
```

Parallel version

```python
import multiprocessing as mp
```

```python
print(f"Number of cpu: {mp.cpu_count()}")
```

```
Number of cpu: 12
```

```python
t0 = time.time()
n_cpu = mp.cpu_count()

pool = mp.Pool(processes=n_cpu)
results = [pool.map(random_square, range(10000000))]
t1 = time.time()
print(f'Execution time {t1 - t0} s')
```

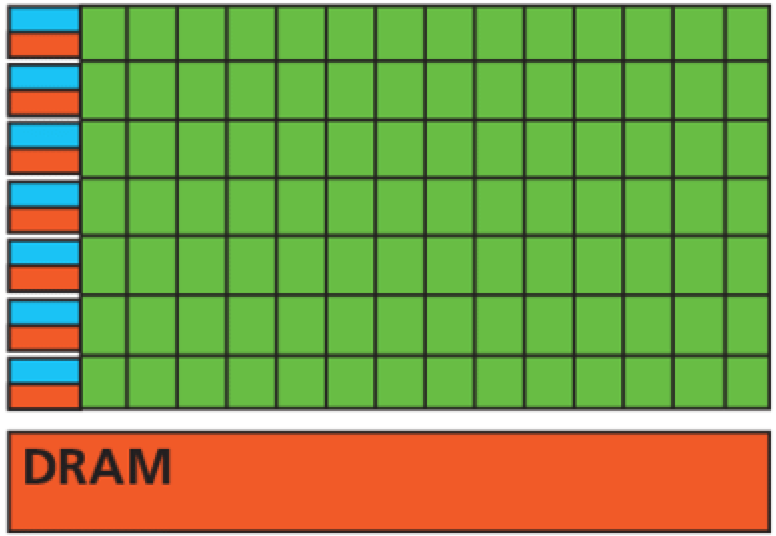
```
Execution time 6.066138744354248 s
```

CPU vs. GPU (1/2)

Latency Oriented

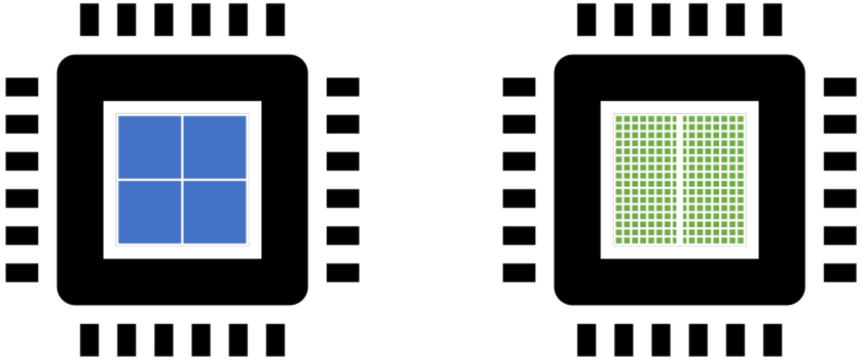
CPU

Throughput Oriented



GPU

CPU vs. GPU (2/2)



CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Python	PyCUDA, Numba
Fortran	CUDA Fortran, OpenACC
C	CUDA C, OpenACC
C++	CUDA C++, Thrust
C#	Hybridizer

Thank You

Next: Introduction to CUDA C Programming