Parallel and Distributed Programming Introduction

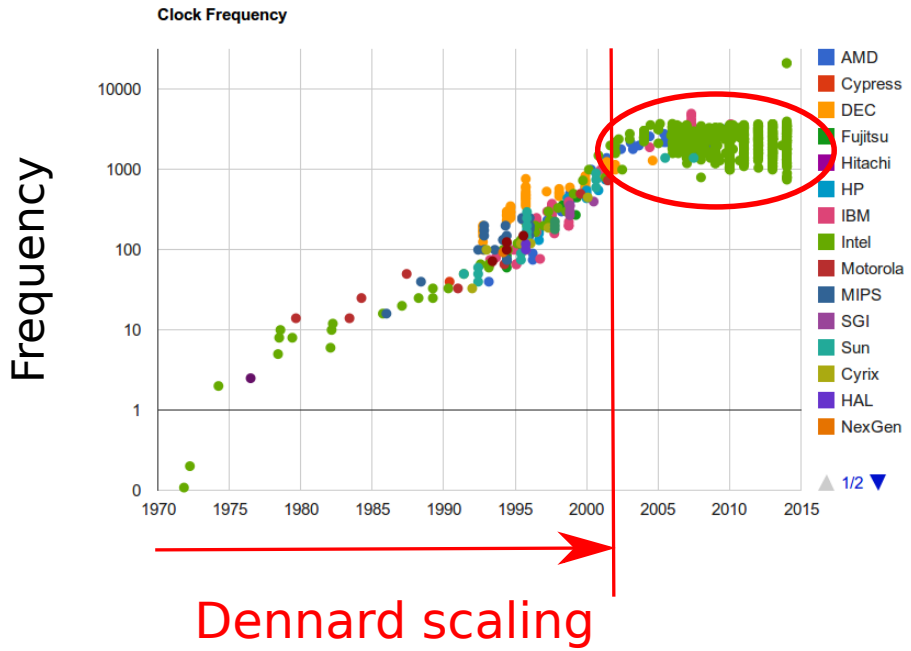
Kenjiro Taura

Contents

Contents

Why parallel?

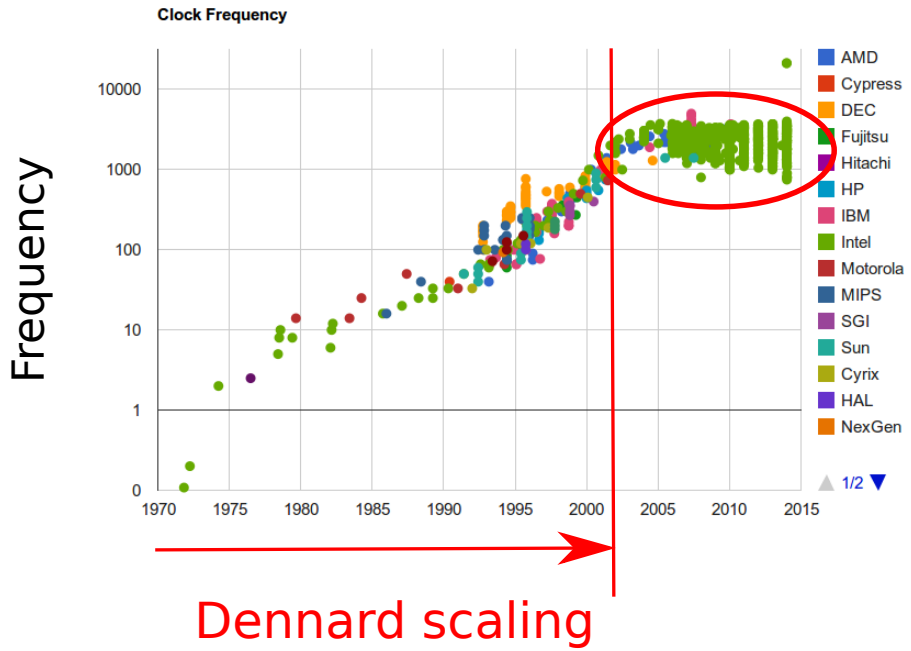
- frequencies no longer increase (end of Dennard scaling)



source: http://cpudb.stanford.edu/

Why parallel?

- frequencies no longer increase (end of Dennard scaling)
- techniques to increase performance (Instruction-Level Parallelism, or ILP) of serial programs are increasingly difficult to pay off (Pollack's law)

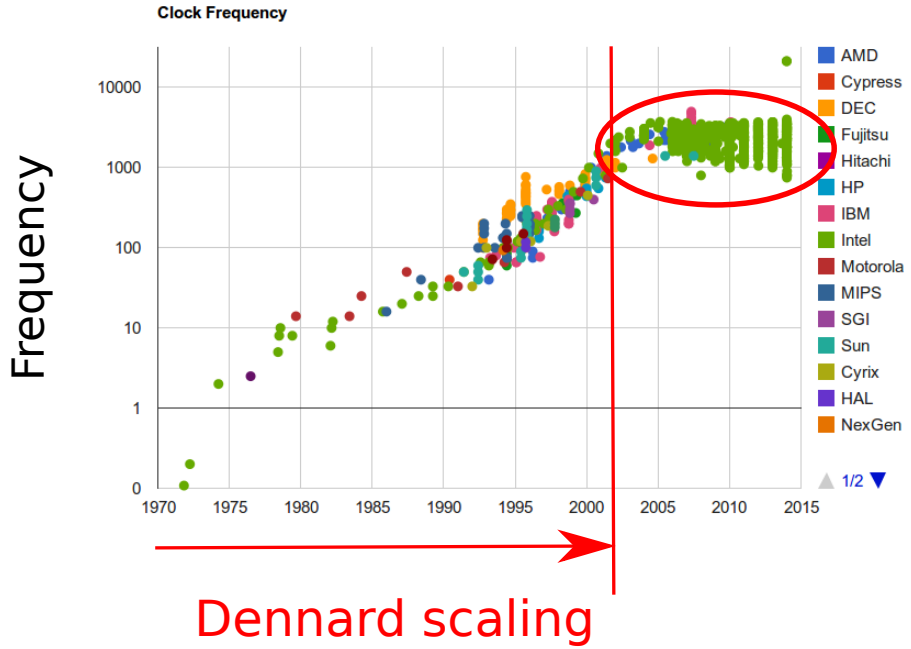


source: http://cpudb.stanford.edu/

Why parallel?

- frequencies no longer increase (end of Dennard scaling)
- techniques to increase performance (Instruction-Level Parallelism, or ILP) of serial programs are increasingly difficult to pay off (Pollack's law)
- multicore, manycore, and GPUs are in part response to it

have more transistors? ⇒ have more cores



source: http://cpudb.stanford.edu/

There are no serial machines any more

- virtually all CPUs are now *multicore*
- high performance accelerators (GPUs and Xeon Phi) run at even low frequencies and have many more cores *(manycore)*

Processors for supercomputers look ordinary, perhaps even more so

TOP 10 Sites for June 2020

For more information about the sites and systems in the list, click on the links or view the complete list.

1-100 | 101-200 | 201-300 | 301-400 | 401-500

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438

www.top500.org

More in the list

4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
7	**Selene** - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	272,800	27,580.0	34,568.6	1,344
8	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	

`www.top500.org`

- Fugaku ($\approx$ 2.2GHz)
- NVIDIA GPU ($\approx$ 1 - 1.5GHz)
- Sunway ($\approx$ 1.45GHz)
- Intel CPUs running at $\approx$ 2.0GHz

Implication to software

- existing serial SWs do not get (dramatically) faster on new CPUs

Implication to software

- existing serial SWs do not get (dramatically) faster on new CPUs
- just writing it in C/C++ goes nowhere close to machine's potential performance, unless you know how to exploit parallelism of the machine

Implication to software

- existing serial SWs do not get (dramatically) faster on new CPUs
- just writing it in C/C++ goes nowhere close to machine's potential performance, unless you know how to exploit parallelism of the machine
- you need to understand
  - does it use multiple cores?
  - if so, how work is distributed?
  - does it use SIMD instructions (covered later)?

Example: matrix multiply

Q: how much can we improve this on my laptop?

```
1  void gemm(long n, /* n = 2400 */
2            float A[n][n], float B[n][n], float C[n][n]) {
3    long i, j, k;
4    for (i = 0; i < n; i++)
5      for (j = 0; j < n; j++)
6        for (k = 0; k < n; k++)
7          C[i][j] += A[i][k] * B[k][j];
8  }
```

Example: matrix multiply

Q: how much can we improve this on my laptop?

```
1  void gemm(long n, /* n = 2400 */
2            float A[n][n], float B[n][n], float C[n][n]) {
3    long i, j, k;
4    for (i = 0; i < n; i++)
5      for (j = 0; j < n; j++)
6        for (k = 0; k < n; k++)
7          C[i][j] += A[i][k] * B[k][j];
8  }
```

```
1  $ ./simple_mm
2  C[1200][1200] = 3011.114014
3   in 56.382360 sec
4   2.451831 GFLOPS
```

Example: matrix multiply

Q: how much can we improve this on my laptop?

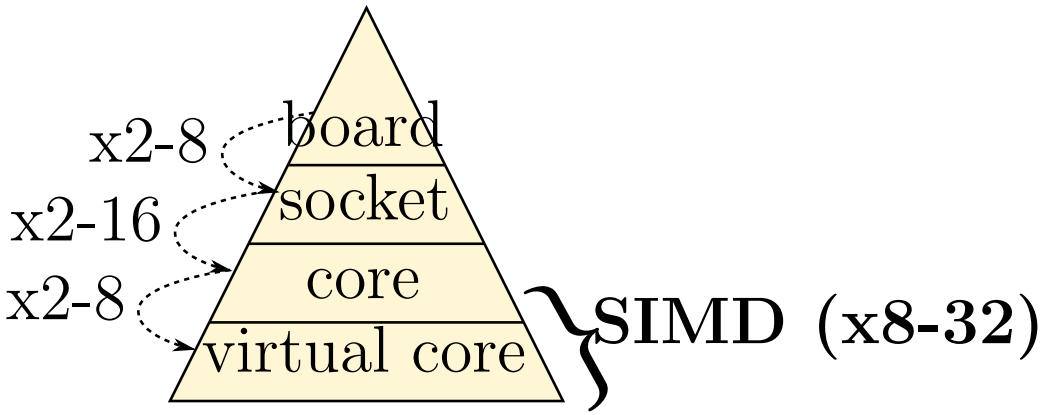
```
1  void gemm(long n, /* n = 2400 */
2            float A[n][n], float B[n][n], float C[n][n]) {
3    long i, j, k;
4    for (i = 0; i < n; i++)
5      for (j = 0; j < n; j++)
6        for (k = 0; k < n; k++)
7          C[i][j] += A[i][k] * B[k][j];
8  }
```

```
1  $ ./simple_mm
2  C[1200][1200] = 3011.114014
3   in 56.382360 sec
4   2.451831 GFLOPS
```

```
1  $ ./opt_mm
2  C[1200][1200] = 3011.108154
3   in 1.302980 sec
4   106.095263 GFLOPS
```

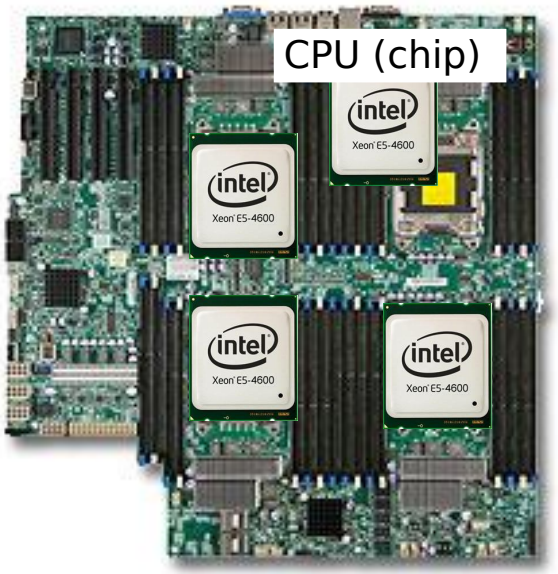
Contents

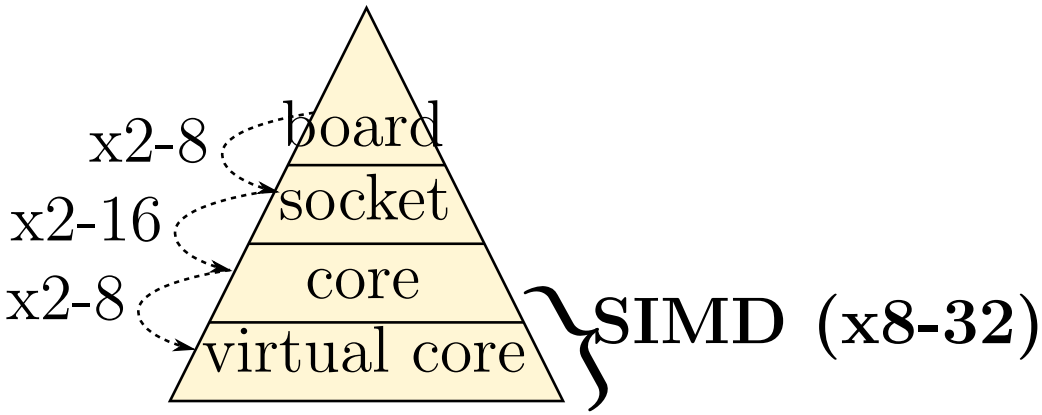
What a single parallel machine (node) looks like



- SIMD : Single Instruction Multiple Data
- a single SIMD register holds many values
- a single instruction applies the same operation (e.g., add, multiply, etc.) on all data in a SIMD register

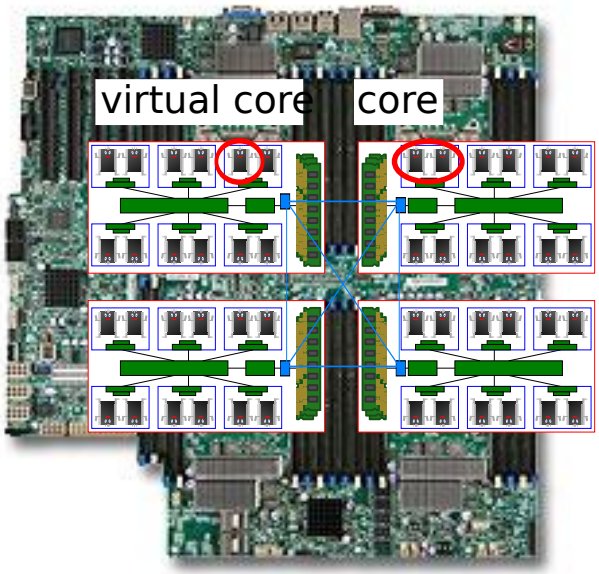
What a single parallel machine (node) looks like

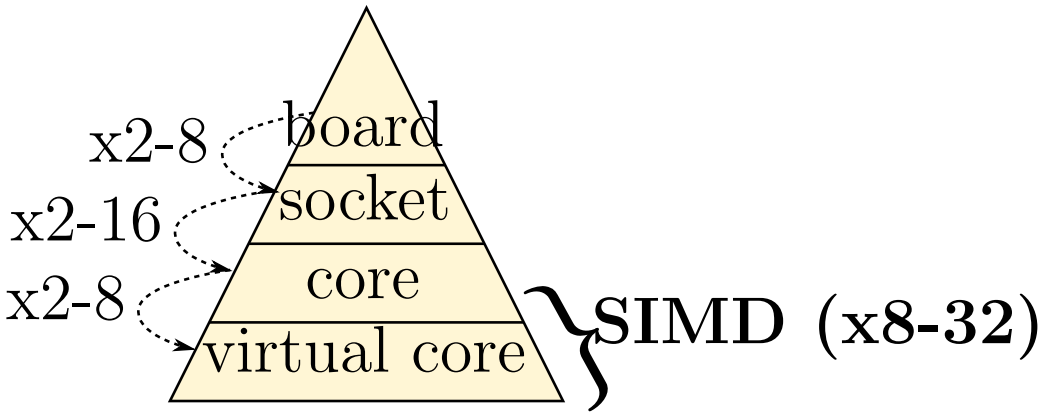




- SIMD : Single Instruction Multiple Data
- a single SIMD register holds many values
- a single instruction applies the same operation (e.g., add, multiply, etc.) on all data in a SIMD register

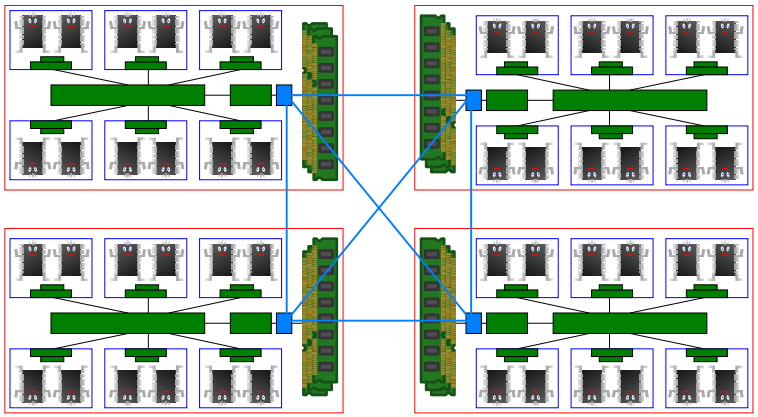
What a single parallel machine (node) looks like

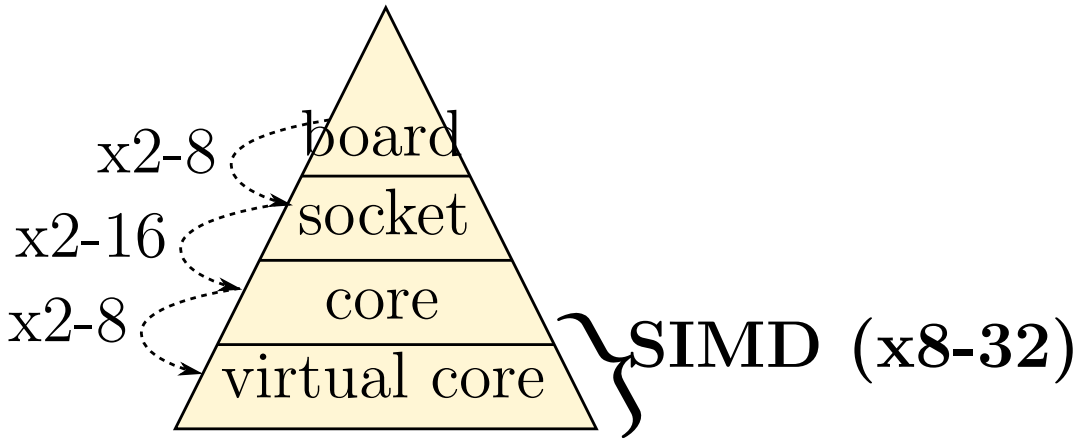




- SIMD : Single Instruction Multiple Data
- a single SIMD register holds many values
- a single instruction applies the same operation (e.g., add, multiply, etc.) on all data in a SIMD register

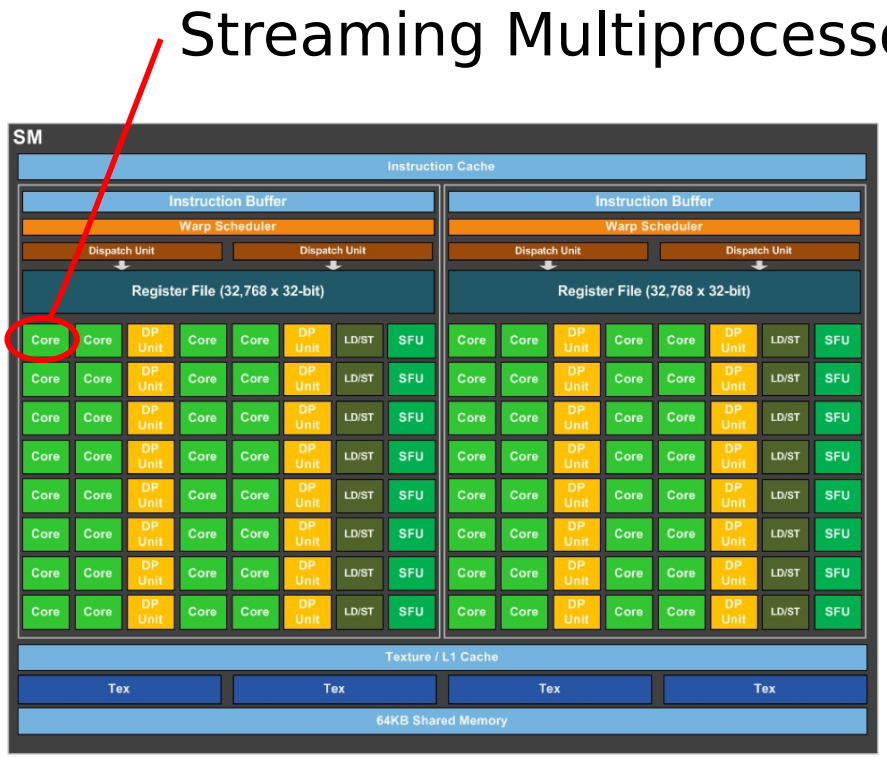
What a machine looks like





- performance comes from *multiplying* parallelism of many levels
- parallelism (per CPU)
  $$= \text{SIMD width} \times \text{instructions/cycle} \times \text{cores}$$
- in particular, peak FLOPS (per CPU)
  $= (2 \times \text{SIMD width}) \times \text{FMA insts/cycle/core} \times \text{freq} \times \text{cores}$
- FMA: Fused Multiply Add ($d = a * b + c$)
- the first factor of 2 : multiply and add (each counted as a flop)

What a GPU looks like?



- a GPU consists of many *Streaming Multiprocessors (SM)*
- each SM is highly multithreaded and can interleave many *warps*
- each warp consists of 32 *CUDA threads*; in a single cycle, threads in a warp can execute the same single instruction

What a GPU looks like?

- despite very different terminologies, there are more commonalities than differnces

GPU	CPU
SM	core
multithreading in an SM	simultaneous multithreading
a warp	a thread executing SIMD instructions
CUDA thread	each lane of a SIMD instruction

- there are significant differeces too, which we'll cover later

How much parallelism?

- Intel CPUs

arch model	SIMD width SP/DP	FMAs /cycle /core	freq GHz	core	peak GFLOPS SP/DP	TDP W
Haswell E78880Lv3	8/4	2	2.0	18	1152/576	115
Broadwell 2699v4	8/4	2	2.2	22	1548/604	145
Skylake 6130	16/8	2	2.1	16	2150/1075	125
KNM 7285	16/8	2	1.4	68	6092/3046	250

- NVIDIA GPUs

acrh model	threads /warp	FMAs /cycle /SM SP/DP	freq GHz	SM	paek GFLOPS SP/DP	TDP W W
Pascal P100	32	2/1	1.328	56	9519/4760	300
Volta V100	32	2/1	1.530	80	15667/7833	300
Ampere A100	32	2/1	1.410	108	19353/9676	400

Note: numbers are without Tensor Cores

Peak (SP) FLOPS

	Skylake 6130		Volta V100
=	(2×16) [flops/FMA insn]	=	(2×32) [flops/FMA insn]
$\times$	2 [FMA insns/cycle/core]	$\times$	2 [FMA insns/cycle/SM]
$\times$	2.1G [cycles/sec]	$\times$	1.53G [cycles/sec]
$\times$	28 [cores]	$\times$	80 [SMs]
=	2150 GFLOPS	=	15667 GFLOPS

Tensor Cores

- performance shown so far is limited by the fact that a single (FMA) instruction can perform 2 flops (1 multiply + 1 add)
- Tensor Core, a special execution unit for a small matrix-multiply-add, changes that
- V100's each Tensor Core can $C = A \times B + C$ for 4×4 matrices

$$2 \times 4 \times 4 \times 4 = 128\text{flops}$$

per cycle (A and B are HP (Half Precision; 16 bit) and C and D are either HP or SP)

- V100 GPU has 640 Tensor Cores

$$\begin{aligned} & \text{Volta V100} \\ = \; & (2 \times 4 \times 4 \times 4)\ [\text{flops/MMA insn}] \\ \times \; & 1.53\text{G}\ [\text{cycles/sec}] \\ \times \; & 640\ [\text{Tensor Cores}] \\ = \; & 125337.6\ \text{GFLOPS} \end{aligned}$$

Trends

- processors' performance improvement is getting less and less "transparent"
  - frequencey + instruction level parallelism → explicit parallelism (multicore) → special execution unit for macro operations (e.g., MMA) → application-specific instructions (?)
- performance is getting more and more dependent on programming

Contents

So how to program it?

- no matter how you program it, you want to maximally utilize multiple cores and SIMD instructions
- "how" depends on programming languages

Language constructs for multiple cores

from low level to high levels

- OS-level threads
- SPMD $\approx$ the entire program runs with N threads
- parallel loops
- dynamically created tasks
- internally parallelized libraries (e.g., matrix operations)
- high-level languages executing pre-determined operations (e.g., matrix operations and map & reduce-like patterns) in parallel (Torch7, Chainer, Spark, etc.)

Language constructs for SIMD

from low level to high levels

- assembly
- intrinsics
- vector types
- vectorized loops
- internally vectorized libraries (e.g., matrix operations)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, TBB, SIMD, . . .)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, TBB, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, TBB, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)
- understand when you can get "close-to-peak" CPU/GPU performance and how to get it (SIMD and instruction level parallelism)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, TBB, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)
- understand when you can get "close-to-peak" CPU/GPU performance and how to get it (SIMD and instruction level parallelism)
- learn many reasons why you don't get good parallel performance

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, TBB, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)
- understand when you can get "close-to-peak" CPU/GPU performance and how to get it (SIMD and instruction level parallelism)
- learn many reasons why you don't get good parallel performance
- have a good understanding about caches and memory and why they matter so much for performance