HPC Methodology

Yusux @ ZJUSCT

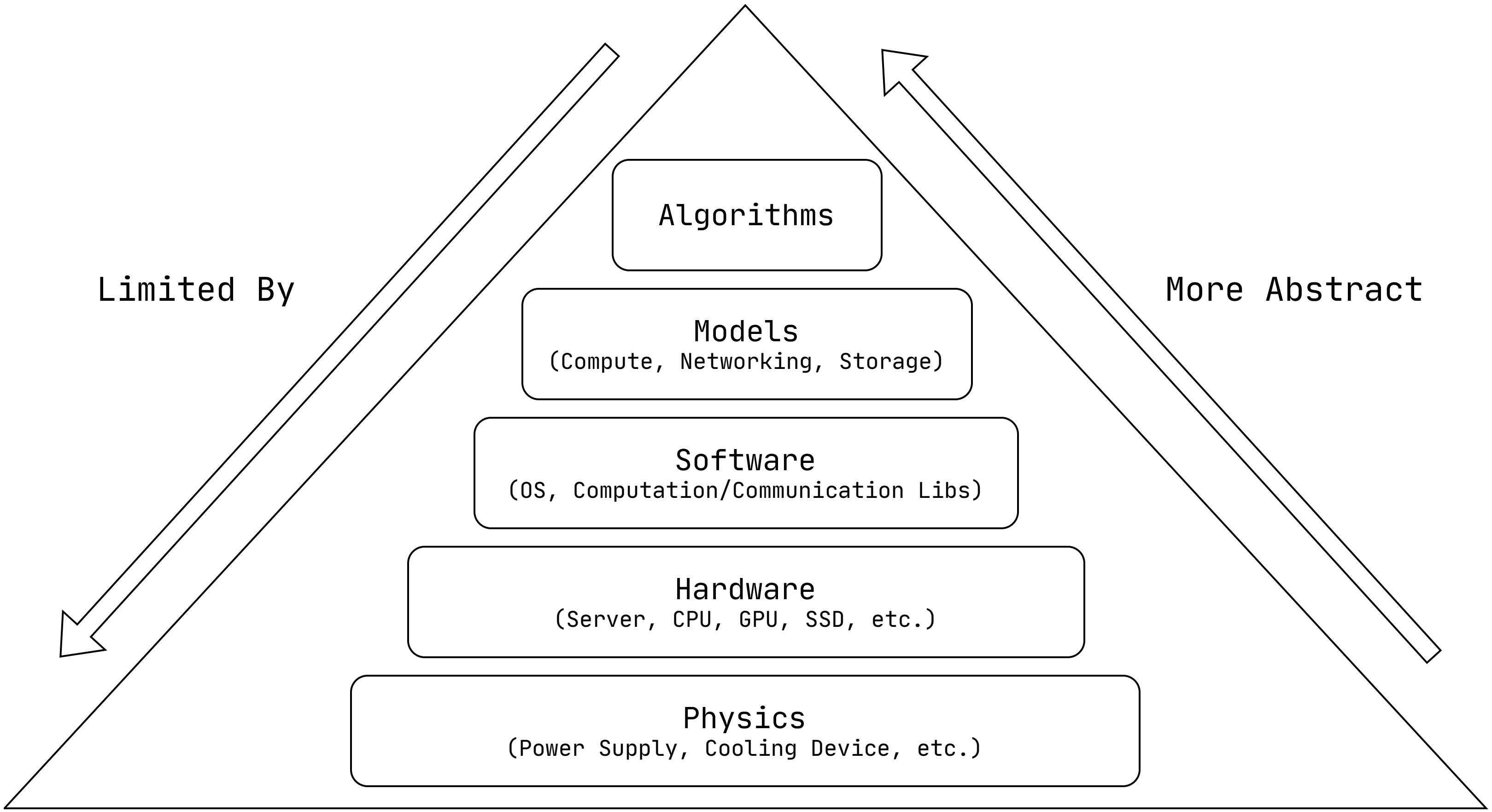
2024/7/3

Today's Content

- Basic Theories for HPC
- Performance Analysis and Optimization Methodology
- Practical Optimization Stretagies
- HPC Skill Tree

1 Basic Theories for HPC

Factors Affecting Performance



High-Level Models

- Compute
  - Program, Function, Programming language, Computation Graph
  - lynn Models (SISD, SIMD, MISD, MIMD), SIMT
  - ...
- Storage
  - Database: Relational, KV, Graph
  - Storage System: Block, Object, File
  - ...
- Networking
  - I/O: Blocking, Signal-Driven, Asynchronous
  - Communication Mode: P2P, Collective Communication
  - ...

Software: Implementation of Models

- Host OS
  - Compute Library
  - BLAS, FFT
  - OpenMP, pthreads, TBB, Intel MKL, Nvidia CUDA
  - ...
- Storage
  - File System: Local, Remote, Distributed
  - ...
- Communication Library
  - MPI, Gloo, NCCL
  - ...

Software: Implementation of Models

- Host OS
  - Compute Library
  - BLAS, FFT
  - OpenMP, pthreads, TBB, Intel MKL, Nvidia CUDA
  - ...
- Storage
  - File System: Local, Remote, Distributed
  - ...
- Communication Library
  - MPI, Gloo, NCCL
  - ...

Hardware: Operated by Software

- Server
- Processing Units
  - CPU, GPU, NPU, FPGA
  - Related: Cache, Memory
  - ...
- Storage Hardware
  - HDD, SSD, NVMe
  - RAID
  - ...
- Networking
  - Ethernet, IB
  - Smart NIC, DPU, IPU
  - ...

Example: Matrix Multiplication - Algorithm

Consider $Y = A \cdot B$, where A, B are huge matrices

$$AB_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

1 x 5 + 2 x 7 = 19

1 x 6 + 2 x 8 = 22

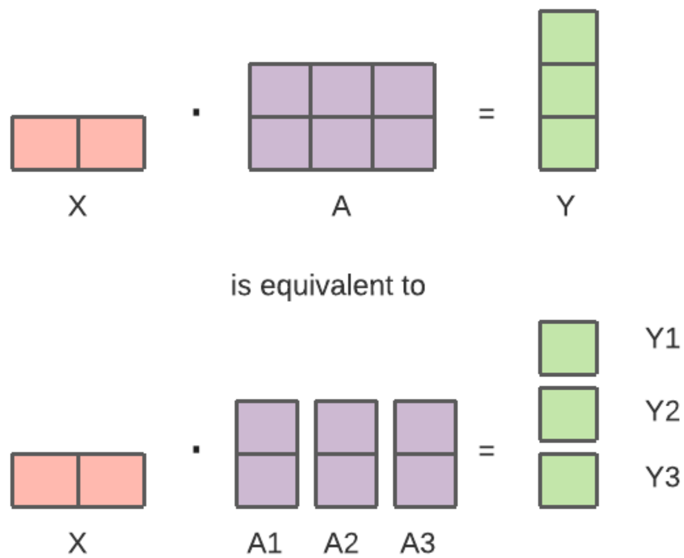
3 x 5 + 4 x 7 = 43

3 x 6 + 4 x 8 = 50

Example: Matrix Multiplication - Models

We decide to run it in parallel

- Assuming we divide it into 3 small matrix multiplication tasks
- Compute on 3 different processing units
- Distribute workload & gather results via network



Example: Matrix Multiplication - Software

- For each small matrix multiplication
  - We use BLAS for efficient computing
- For workload distributing & result gathering
  - We use MPI for communication

Example: Matrix Multiplication - Hardware

- We use BLAS on CPU/GPU
  - More efficient/powerful CPU/GPU → higher performance
  - For computation, GPUs are usually faster
- We use MPI on InfiniBand
  - Larger throughput & lower latency

Example: Matrix Multiplication - Physics

- All these hardware may be subject to physical limitations
- Do not let them overheat or run out of power

2 Performance Analysis and Optimization Methodology

2.1 What is optimization?

What is optimization?

Before Manual Optimization:

```cpp
#include <cstdio>

long long fibonacci(int i) {
    if(i <= 2) {
        return 1;
    } else {
        return fibonacci(i - 1)
             + fibonacci(i - 2);
    }
}
int main() {
    int k = 5;
    printf("fib(%d)=%lld\n", k,
           fibonacci(k));
    return 0;
}
```

After Manual Optimization:

```cpp
#include <cstdio>

static constexpr long long
fibonacci(int i) {
    return i <= 2
        ? 1
        : fibonacci(i - 1) +
          fibonacci(i - 2);
}
int main() {
    const int k = 5;
    printf("fib(%d)=%lld\n", k,
           fibonacci(k));
    return 0;
}
```

What is optimization?

Before Manual Optimization:

```cpp
#include <cstdio>

long long fibonacci(int i) {
    if(i <= 2) {
        return 1;
    } else {
        return fibonacci(i - 1)
             + fibonacci(i - 2);
    }
}
int main() {
    int k = 5;
    printf("fib(%d)=%lld\n", k,
           fibonacci(k));
    return 0;
}
```

Compilation Result (gcc 13.2.0, -O2):

```asm
... ;(omitted)
main:
        sub     rsp, 8
        mov     edi, 5
        call    _Z9fibonaccii
        mov     esi, 5
        mov     edi, OFFSET FLAT:.LC0
        mov     rdx, rax
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

https://godbolt.org/z/rax365P1P

What is optimization?

After Manual Optimization:

```cpp
#include <cstdio>

static constexpr long long
fibonacci(int i) {
    return i <= 2
        ? 1
        : fibonacci(i - 1) +
          fibonacci(i - 2);
}
int main() {
    const int k = 5;
    printf("fib(%d)=%lld\n", k,
           fibonacci(k));
    return 0;
}
```

Compilation Result (gcc 13.2.0, -O2):

```asm
.LC0:
        .string "fib(%d)=%lld\n"
main:
        sub     rsp, 8
        mov     edx, 5
        mov     esi, 5
        xor     eax, eax
        mov     edi, OFFSET FLAT:.LC0
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

https://godbolt.org/z/3KKcKEjWa

What is optimization?

After Manual Optimization (another way):

```
#include <cstdio>

int main() {
    puts("fib(5)=5");
    return 0;
}
```

Failed O2 Optimization (k = 93, since $fib(93) > 2^{63}$):

https://godbolt.org/z/YrYx3eKbz

Another example (Collatz Conjecture):

```
bool collatz(int x) {
    while (true) {
        if (x <= 1) return true;
        if (x % 2) x >>= 1;
        else x = 3*x + 1;
    }
}
```

```
_Z7collatzi:
        mov     eax, 1
        ret
```

https://godbolt.org/z/exfEjdshj

What is optimization?

> *Mathematical optimization or mathematical programming is the selection of a best element, with regard to some criteria, from some set of available alternatives.* [Wikipedia]

For example, maximize/minimize $f(x)$ subject to $x \in \Omega$

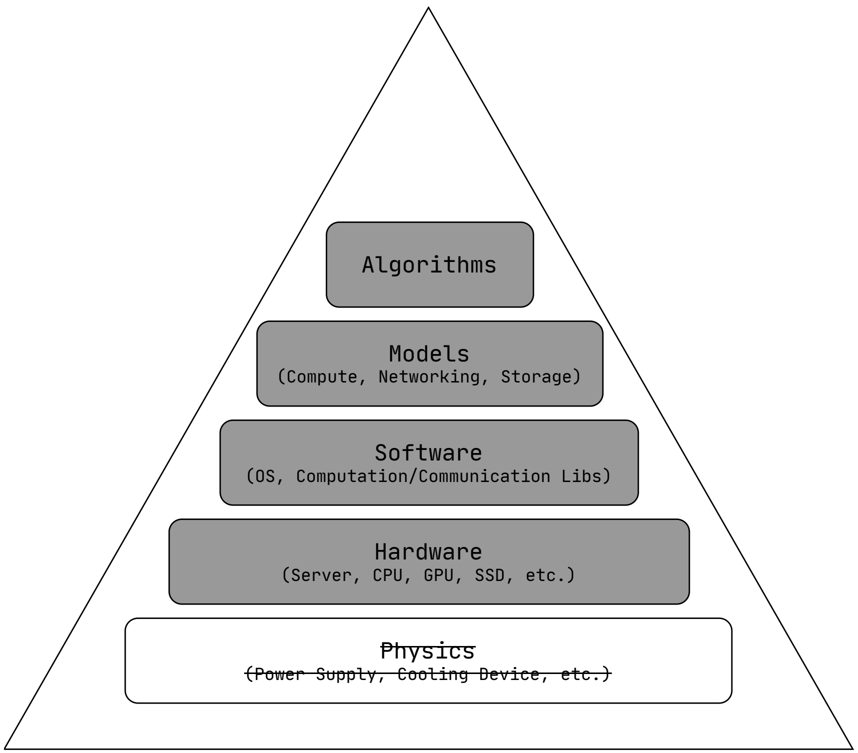
- x: decision (selection within available alternatives)
- $f(x)$: objective function
- Ω: constraints (criteria)

What is optimization? (Back to HPC)

In our case:

- Goal: Maximize performance
  - Speed
  - Throughput
  - Latency
  - ...
- Criteria: Limited resources
  - Restricted hardware
  - Limited Power
  - Limited Quota
  - ...

- Alternatives



2.2 Should I optimize?

Should I optimize?

- Is performance critical to my program?
  - One-time small programs, just run them slowly
  - I can wait till tomorrow to see the results, just play and wait for it
- Is there room for optimization?
  - Performance Test
  - Optimization Space Analysis

Performance Test

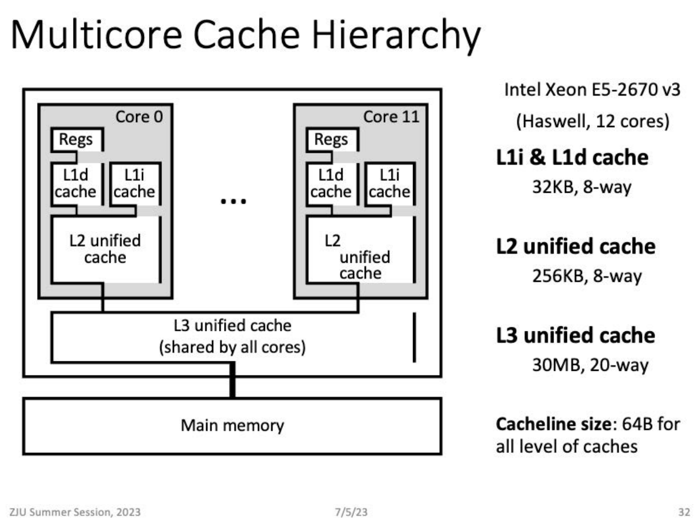
Just directly run the program and see how long it takes (measured by wall time)

```
time_t start = get_time_hires();
// loop 100 times to get a more accurate result
// by averaging the time
for(int i = 0; i < 100; i++) {
    do_something();
}
time_t stop = get_time_hires();
time_t res = (stop - start)/100;
```

Optimization Space Analysis

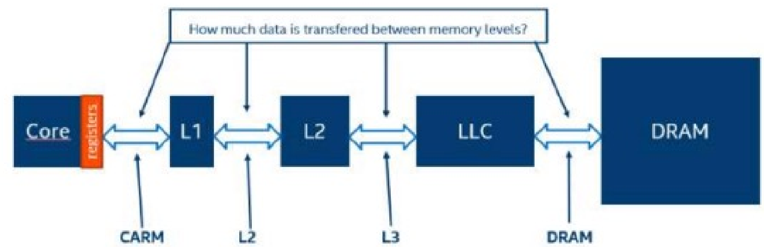
We can find the theoretical upper bounds

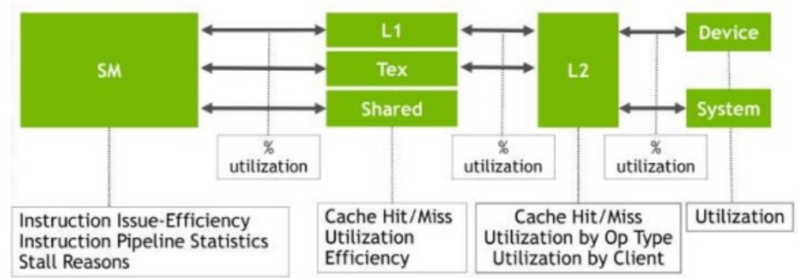
- CPU/GPU Flops
- Memory Accessing Speed
- PCIe Bandwidth
- Disk/Net IO Speed
- ...



However, it is somehow still an open question

- Modern/Real-World architectures are **complicated**
- Turn to use **black-box models**





Opt. Space Analysis - Roofline Model

> *The roofline model is an intuitive visual performance model used to provide performance estimates of a given kernel or application, by showing inherent hardware limitations, potential benefit and priority of optimizations.* *[Wikipedia]*

- Work
  - The work W denotes the number of operations performed, and in most cases, W is expressed as FLOPs
- Memory traffic
  - The memory traffic Q denotes the number of bytes of memory transfers incurred during the execution
- Arithmetic intensity
  - The arithmetic intensity I is the ratio of the work W to the memory traffic Q

Opt. Space Analysis - Roofline Model

Roofline model only focus on 1~2 dominant components

Example: CPU DRAM Roofline

$$\text{Arithmetic intensity}(I) = \frac{\text{Floating point operaraons (W)}}{\text{Total data movement (Q)}} (\text{FLOPs}/\text{Byte})$$

For Matrix Multiplication of two $n \times n$ matrices

- Floating point operaraons = $2n^3$ = $\text{O}(n^3)$
- Total data movement = $3n^2$ = $\text{O}(n^2)$
- Arithmetic intensity $I = \frac{2n^3}{3n^2} = \frac{2}{3}n = \text{O}(n)$

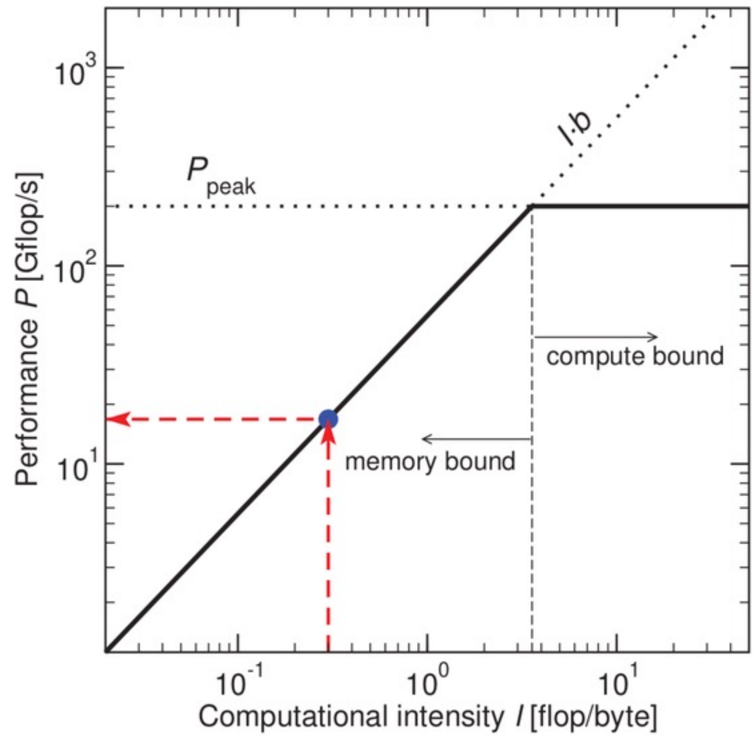
Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM 52.4 (2009): 65-76.

Opt. Space Analysis - Roofline Model

Sustainable performance is bound by

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

- P: Attainable performance
- π: Peak performance
- β: Peak bandwidth
- I: Arithmetic intensity



Opt. Space Analysis - Roofline Model

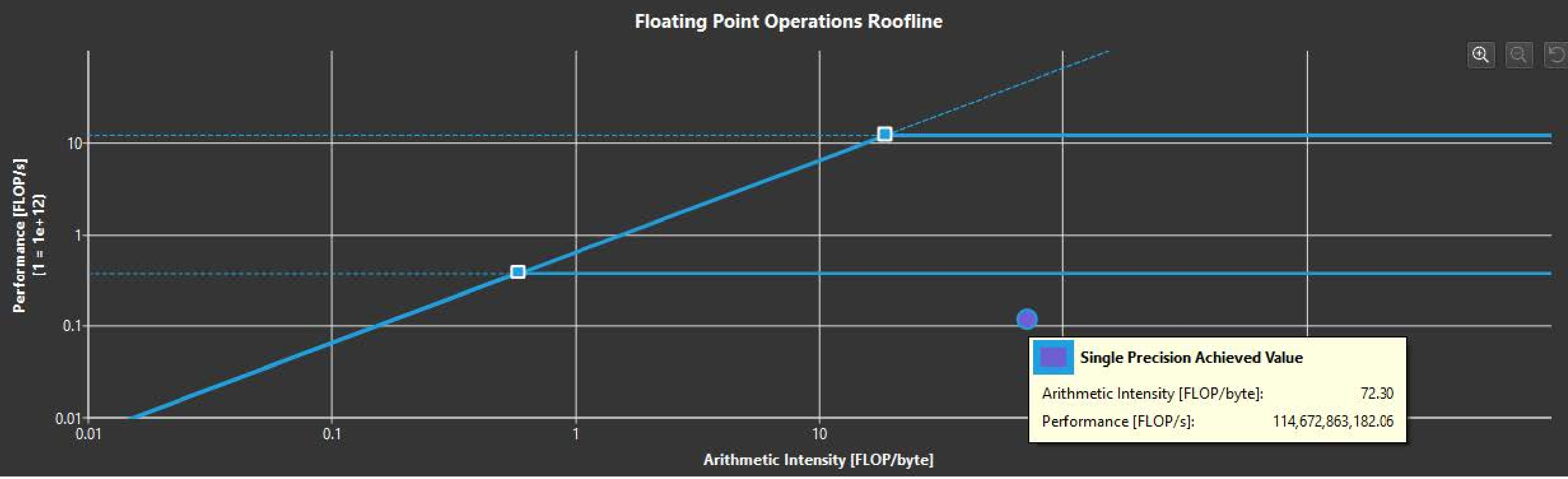
GPU Speed Of Light Throughput | GPU Throughput Rooflines

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	86.42	Duration [msecond]	1.14
Memory Throughput [%]	8.04	Elapsed Cycles [cycle]	1,557,809
L1/TEX Cache Throughput [%]	8.45	SM Active Cycles [cycle]	1,477,479.37
L2 Cache Throughput [%]	0.97	SM Frequency [cycle/nsecond]	1.36
DRAM Throughput [%]	0.24	DRAM Frequency [cycle/nsecond]	6.78

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

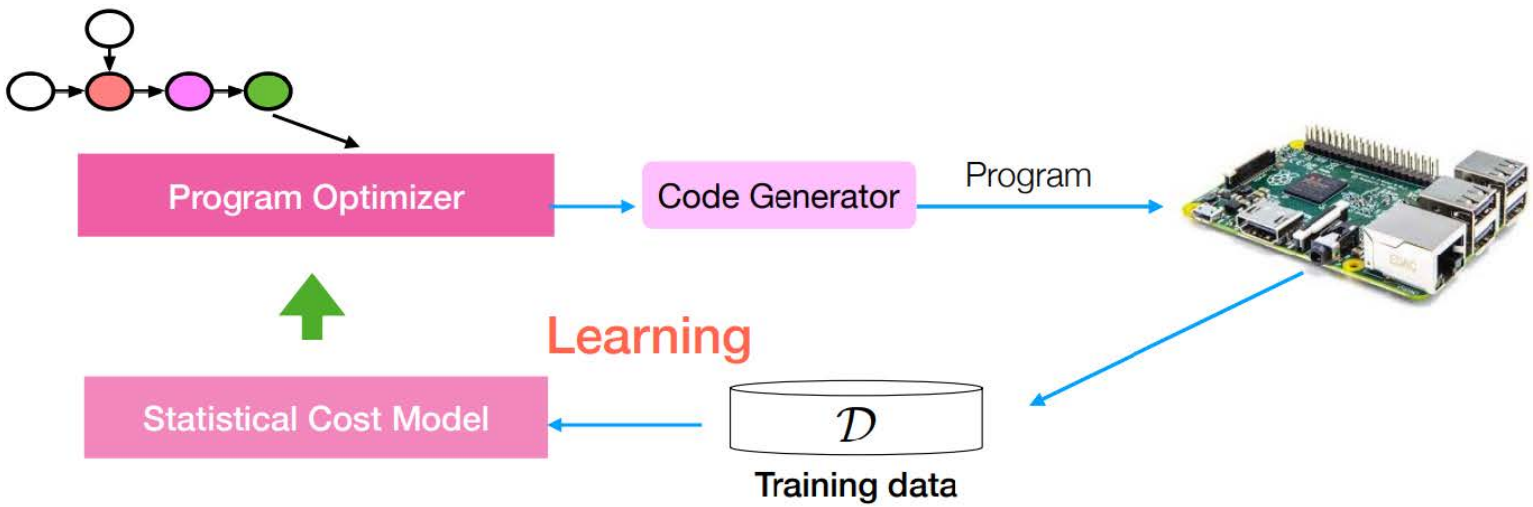
Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved close to 1% of this device's fp32 peak performance and 0% of its fp64 peak performance.



Opt. Space Analysis - Other Models

Learning-based Statistical Cost Model

- Adapt to different hardware type by learning



Chen, Tianqi, et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning." 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).2018.

2.3 Where to optimize?

Where to optimize? - Amdahl's law

The slowest part → bottleneck/hotspot

> *Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.* *[Wikipedia]*

Amdahl's law can be formulated in the following way:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \leq \frac{1}{1-p}$$

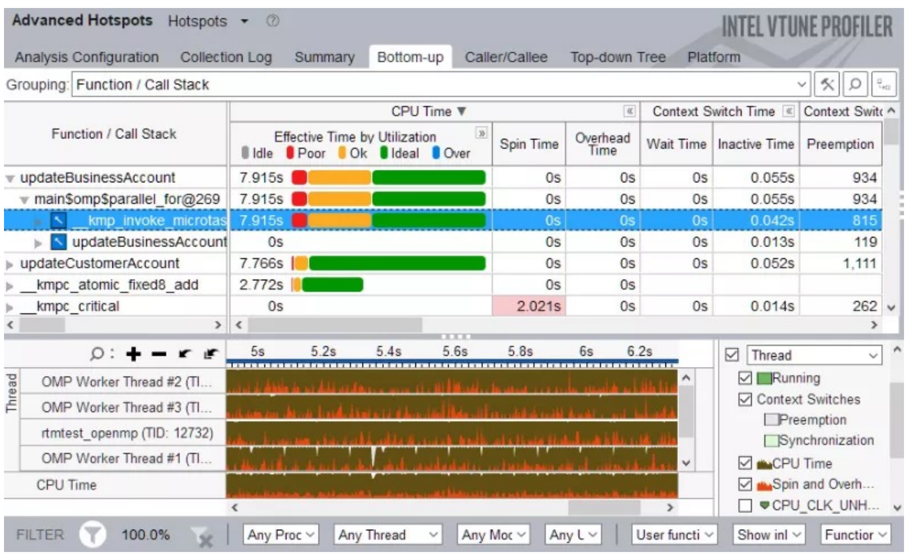
- $S_{latency}(s)$: Theoretical speedup of the whole task
- s: Speedup of the part of the task that benefits from improved system resources
- p: Proportion of execution time that the part benefiting from improved resources originally occupied

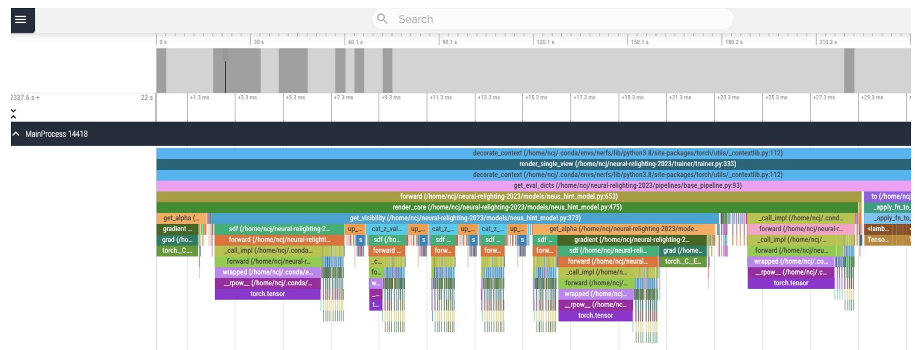
Hotspot Analysis

Use hotspots analysis to find the bottleneck of the program

Methods:

- Analytical
- Hardware simulator
- Profile: sampling some usage of a resource by a program
- Trace: collecting highly detailed data about the execution





2.4 General Optimization Pipeline

General Optimization Pipeline

1. Determine your baseline code
2. Run performance test
3. Is optimization target reached? (Optimization Space Analysis)
4. Find bottleneck (Hotspot Analysis)
5. Optimize the bottleneck
6. Go to 2.

3 Practical Optimization Stretagies

Optimization Strategies

- Algorithm optimization
  - reduce complexity
  - space for time
  - ...
- Code optimization
  - remove redundancy
  - reduce precision
  - ...
- Compile/running parameter optimization
- Hardware optimization

3.1 Algorithm Optimization

Alg. Optimization - Reduce Complexity

The following code is the fast inverse square root implementation from Quake III Arena, and the 2nd Newton iteration can be removed to reduce complexity with cost of precision

```
float Q_rsqrt(float number) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;             // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );   // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
    // y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

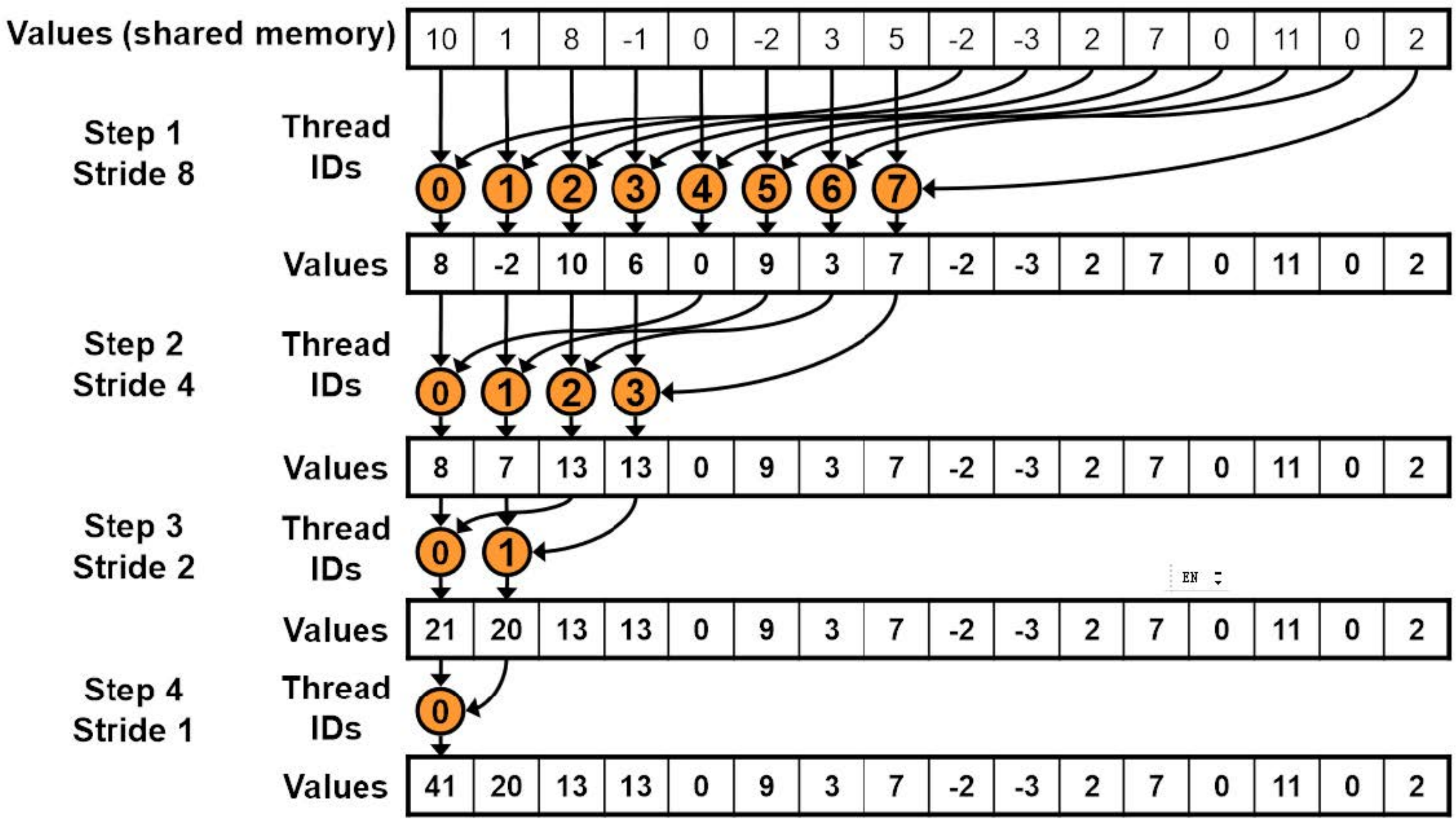
Alg. Optimization - Trade space for time

Lookup tables are used to accelerate CRC32 computation.

```
uint32_t poly8_lookup[256] = {
    0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA,
    0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,
    0x0EDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988,
    0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,
    // ...
}
```

Alg. Optimization - Parallelization

Sum a large array



Alg. Optimization – Prefetch & Prediction

- Locality of high-level logic
  - Web page prefetching/Data preloading
  - Contributes to locality at lower-levels
- Instruction level
  - Branch prediction

Alg. Optimization – Caching

- Stores results from previous executions
  - Directly returns stored results
  - Requirement: pure function
    - Return values are identical for identical arguments
  - Add cache invalidation for non-pure ones
- Limited cache size
  - LRU/Set Associativity

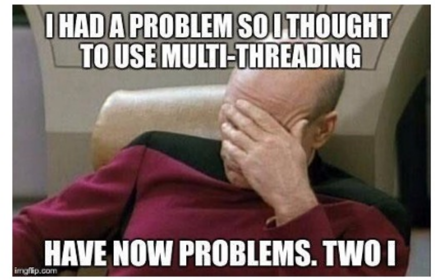
Alg. Optimization - Lock-Free

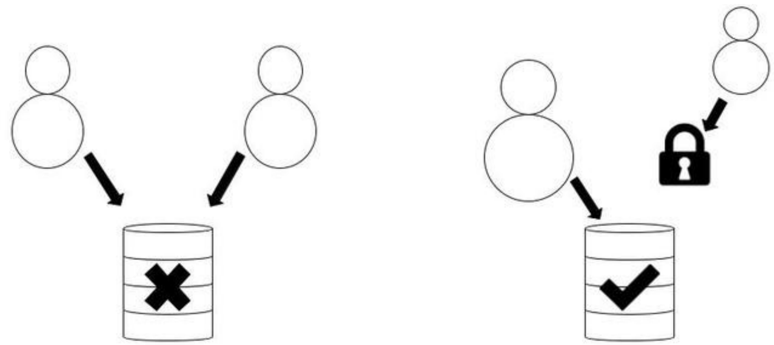
- Locks are needed for concurrency
- However, Lock → Waiting (e.g. spinlock)
  - Waste CPU resources
- Use atomic primitives
  - CAS (Compare and Swap)
  - Atomic_Add
- Negative example
  - GIL in Python

Critical Section

- We are expecting 20,000 (2 times 10,000)
- However we are getting weird results:

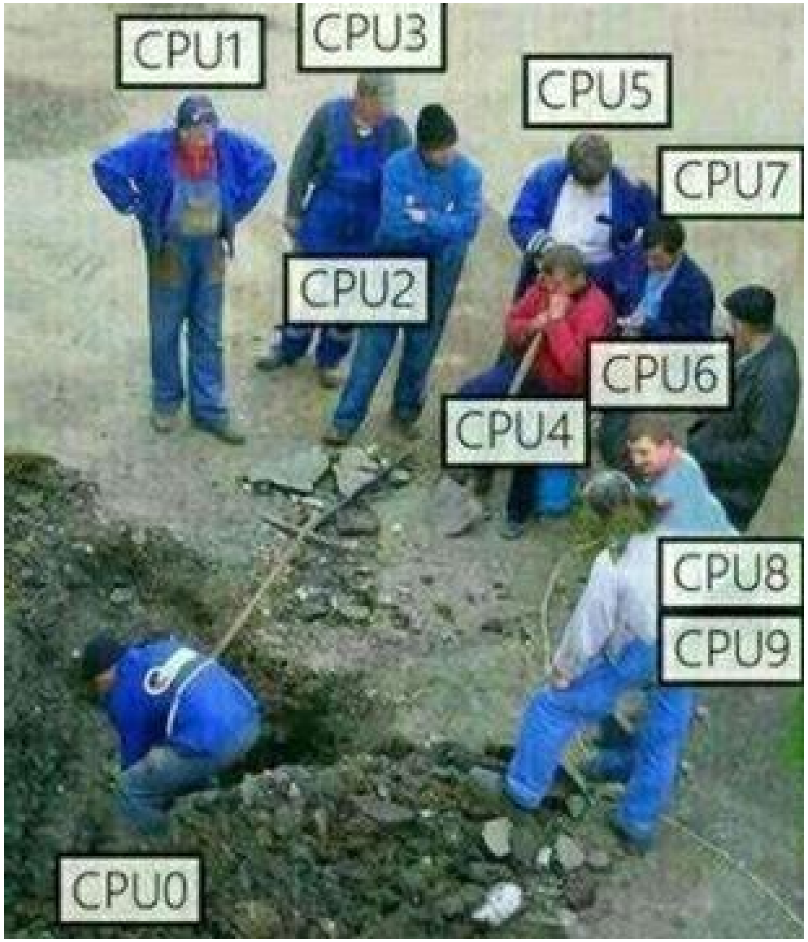
17636
17930
18185
19362
...





Alg. Optimization - Load Balancing

Avoid load imbalance (所谓“一核有难，七核围观”)



3.2 Code Optimization

Code Optimization - Remove Redundancy

Before:

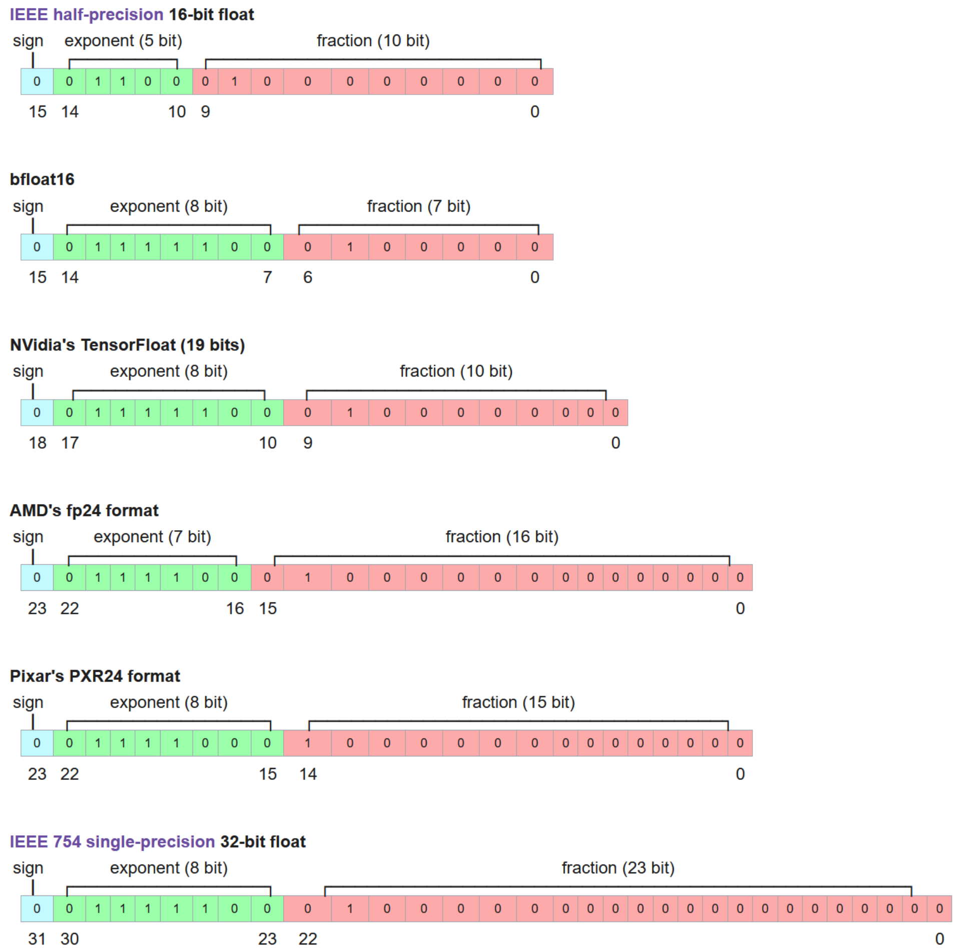
```
if (fn(1) >= 0 && fn(1) < 10) {
    do_fn1_between_0_10();
} else if(fn(1) >= 10 && fn(1) < 1000) {
    do_fn1_between_10_1000();
} else {
    do_fn1_unknown_state();
}
```

After:

```
auto fn1val = fn(1);
if (fn1val >= 0 && fn1val < 10) {
    do_fn1_between_0_10();
} else if(fn1val >= 10 && fn1val < 1000) {
    do_fn1_between_10_1000();
} else {
    do_fn1_unknown_state();
}
```

Code Optimization – Reduce Precision

- High-precision data
  - Take up lots of space
  - Large computation cost
  - Consume lots of resources
- Save both memory & computation by reducing precision
  - FP8, FP16, FP32, FP64, FP128
  - INT1, INT4, INT8, INT16, INT32, INT64, INT128
- Mix-precision
  - PyTorch AMP
  - Transformer Engine



Code Optimization - Reduce Branching

- Range comparison w/ binary decomposition
  - Binary Search Tree
- Skip List

Before:

```
assert(v ≥ 0 && v < 100);
if (v ≥ 75) {
    // 75..99
} else if (v ≥ 50) {
    // 50..74
} else if (v ≥ 25) {
    // 25..49
} else {
    // 0..24
}
```

After:

```
assert(v ≥ 0 && v < 100);
if (v ≥ 50) {
    if (v ≥ 75) {
        // 75..99
    } else {
        // 50..74
    }
} else {
    if (v ≥ 25) {
        // 25..49
    } else {
        // 0..24
    }
}
```

Code Optimization - Vectorization

What is vectorization?

- Scaler computation: $a = 2 \cdot a$
- Vector computation: $\vec{a} = 2 \cdot \vec{a}$
  - [a, b, c, d] ⇒ [2a, 2b, 2c, 2d]

Methods:

- High-level: vectorized computation graph
- Instruction-level: SIMD instructions

Enjoy your lab2~

Code Opt. - Optimize Memory Access Locality

For GEMM

- Blocking
- Loop Permutation/Unrolling
- Array Packing
- ...

Enjoy your following labs~

Code Optimization - Adjusting Modifiers

Just as the `fibonacci` function, we can use `constexpr` and `const` to hint the compiler to optimize the code

```cpp
#include <cstdio>

static constexpr long long fibonacci(int i) {
    return i <= 2
        ? 1
        : fibonacci(i - 1) +
          fibonacci(i - 2);
}
int main() {
    const int k = 5;
    printf("fib(%d)=%lld\n", k, fibonacci(k));
    return 0;
}
```

Code Opt. - Instruction/Data Alignment

- Optimize CPU memory access
- Usually done automatically by compilers

```
  bank 1    bank 2
+--------+--------+
|  8 bit | 8 bit  |
+--------+--------+
|        |        |
+--------+--------+
| 4      | 5      | <-- the CPU can fetch only values on the same "row"
+--------+--------+
| 2      | 3      |
+--------+--------+
| 0      | 1      |
+--------+--------+
 \      / \      /
  |    |   |    |
  |    |   |    |

 data bus  (to uP)
```

Discussion: Man v.s. Compiler

When do we need manual optimization?

Domain Specific Language

(b) Fast C++ (for x86) : 0.90 ms per megapixel

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

(c) Halide : 0.90 ms per megapixel

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```

Ragan-Kelley, Jonathan, et al. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." ACM Transactions on Graphics (TOG) 31.4 (2012): 1-12.

3.3 Compile/Running Parameter Optimization

Compile/Running Parameter Tuning

- Adjust Running Scale
- Adjust Cache Size
- Adjust Core Affinity
  - NUMA

Discussion: Is Parameter-tuning Optimization?

- Adapts general code to local machine
- Auto-tuning
  - Black-box method: TVM (learning-based), etc.
  - Analytical: Alpa (Dynamic programming, etc.)

3.4 Hardware Optimization

Different Hardware

CPU → GPU → ASIC/DSA/FPGA

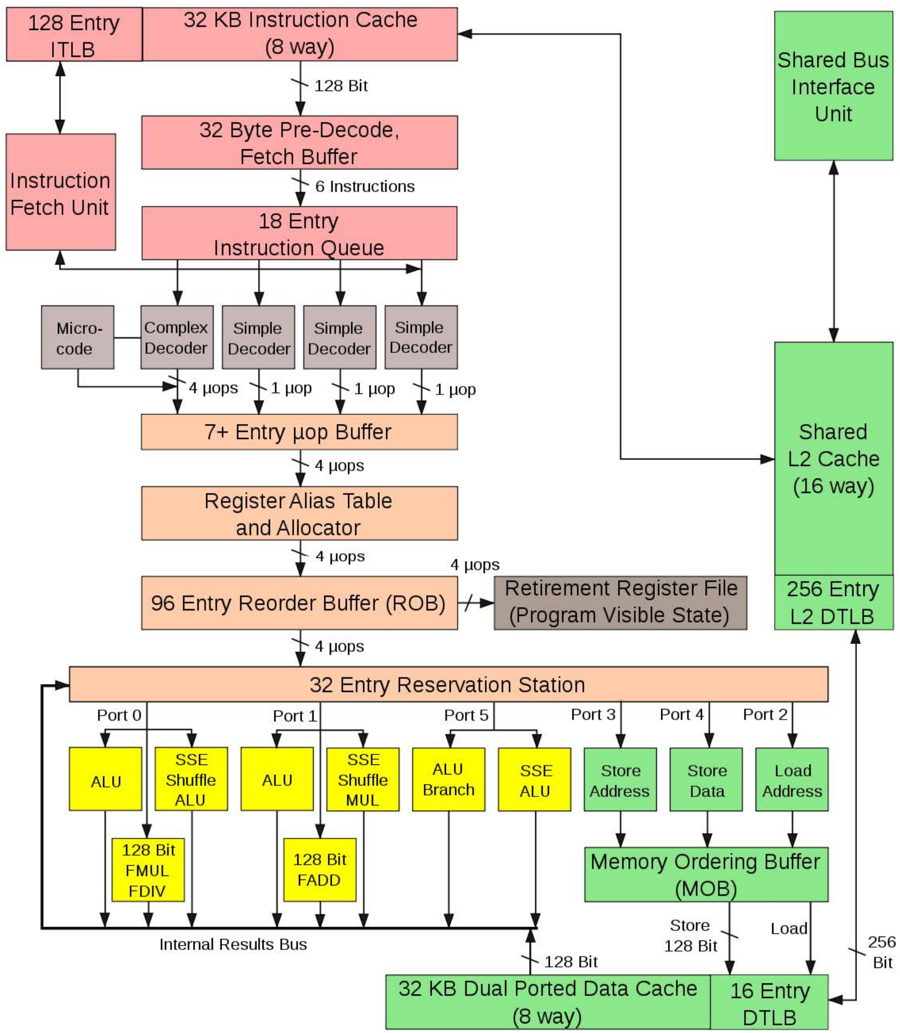
	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	10	4.3 GHz	System RAM	$385	~640 **G**FLOPs FP32
GPU (NVIDIA RTX 3090)	10496	1.6 GHz	24 GB GDDR 6X	$1499	~35.6 **T**FLOPs FP32
GPU (Data Center) NVIDIA A100	6912 CUDA, 432 Tensor	1.5 GHz	40/80 GB HBM2	$3/hr (GCP)	~9.7 TFLOPs FP64 ~20 TFLOPs FP32 ~312 TFLOPs FP16
TPU Google Cloud TPUv3	2 Matrix Units (MXUs) per core, 4 cores	?	128 GB HBM	$8/hr (GCP)	~420 TFLOPs (non-standard FP)

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

TPU: Specialized hardware for deep learning

Different Hardware (Cont.)

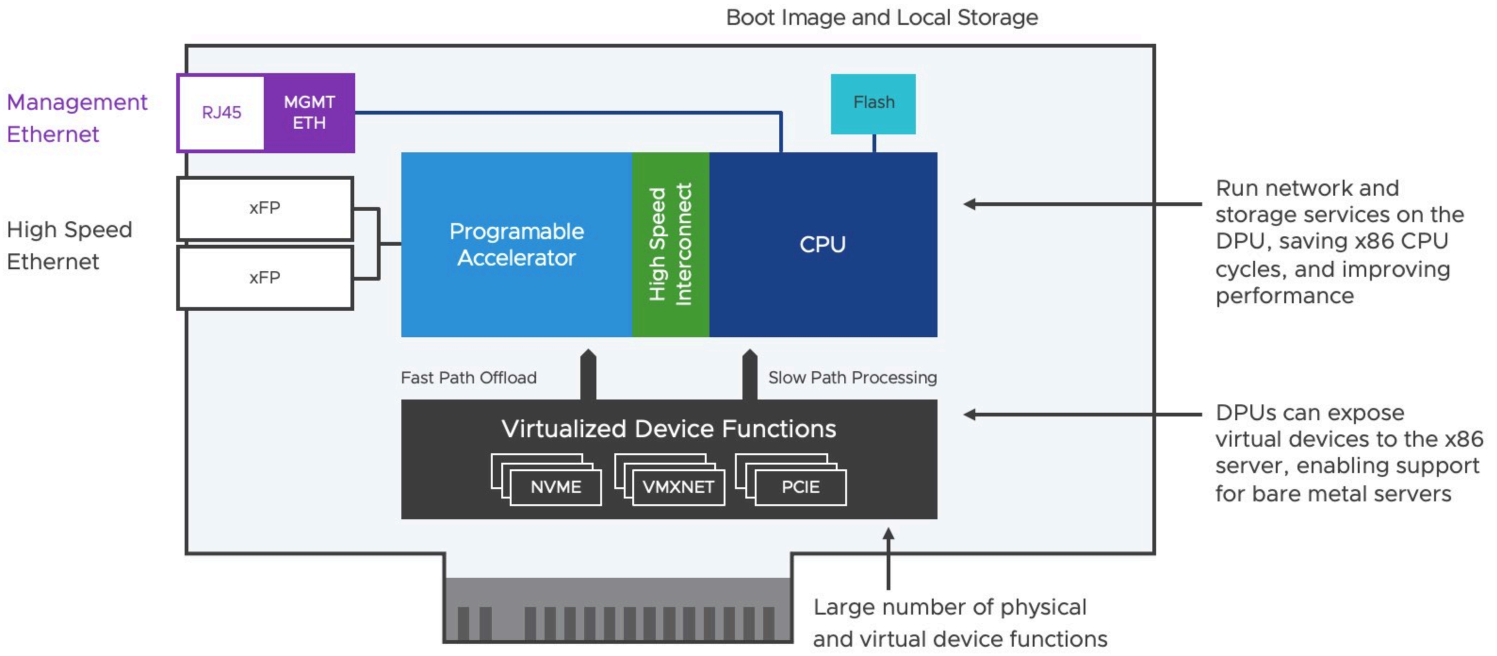


Intel Core 2 Architecture

Hardware Optimization - DPU

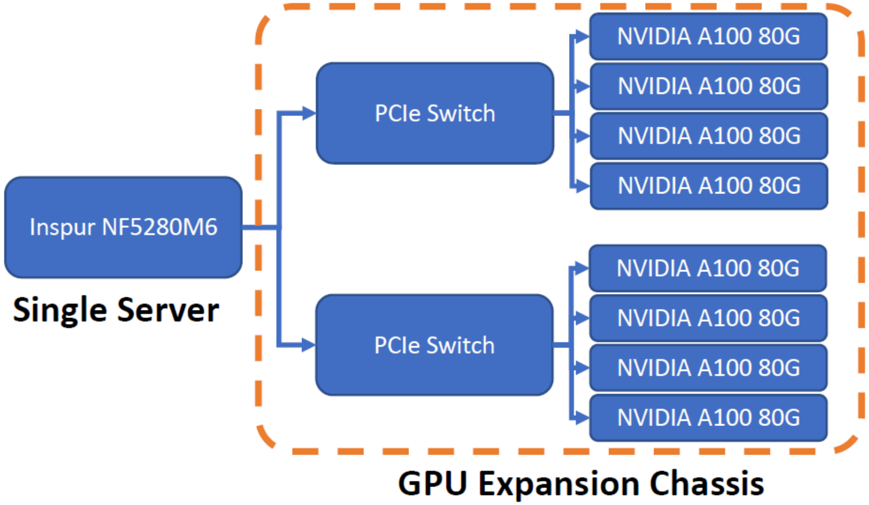
DPU - System on a chip that combines

- Industry-standard, high-performance, software-programmable multi-core CPU
- High-performance network interface
- Flexible and programmable acceleration engines



Hardware Optimization - GPU Chassis

Optimize communication & power usage



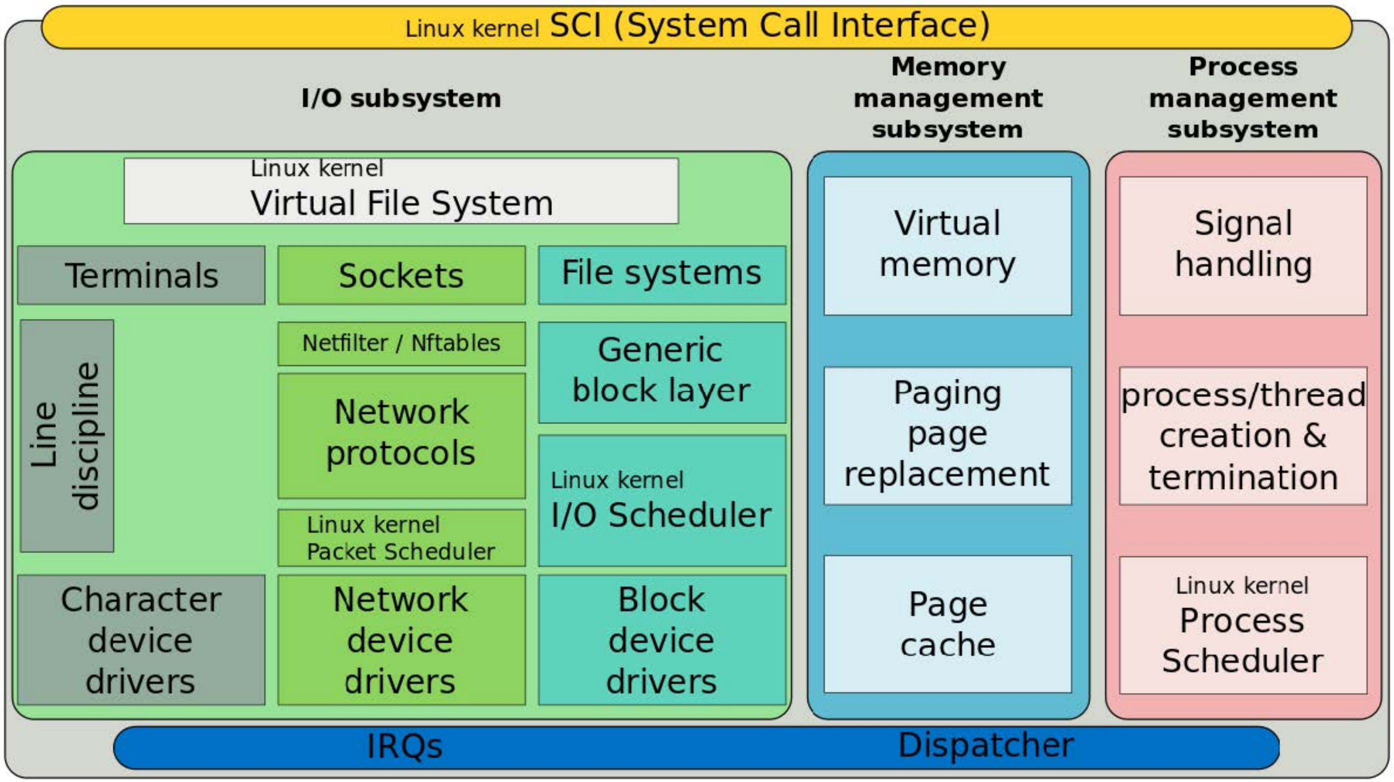
4 HPC Skill Tree

HPC Skills

- Linux system and common commands
- Cluster maintainment and network management
- Collaborative development and version control
- Script automation
- Complex data analysis and processing
- Manual compilating and linking of dependent programs
- Parallel program design, testing and optimization
- Power control and parameter adjustment

Linux System and Common Commands

- Compared with Windows Server, Linux occupies less resources
  - Generally the operating system used by servers are Linux
- Ecosystem: Many scientific computing software only have Linux versions, or have no official support on other systems



Linux System and Common Commands (Cont.)

Shells that are commonly used in Linux:

- Ash
- Bash
- Zsh
- ...

Why we need shells?

- In many cases, there is no GUI on servers
  - To save resources and reduce maintenance costs
- Remote GUI access is not provided

Cluster Maintanence and Network Management

Including

- (Un)Installation of various software and environment
- Software and hardware troubleshooting
  - Network problem troubleshooting
- Job submission & scheduling system (slurm, LSF, Spark...)
- Cluster status monitoring
- IaaS, PaaS
- ...

Provide a stable and efficient computing environment

Characteristics: troublesome

Collaborative Dev. and Version Control

- Cooperation awareness, team spirit, ...
- Use version control software (mainly Git)
- Software engineering
- Documentation, annotation
- Communication skills

Script Automation

When running a script that takes a long time, running it manually requires a lot of effort

To ~~be lazy~~ improve efficiency, use automated scripts instead of manual operations

- Linux shell scripts
  - Usually used for simple/general tasks
- Scheduling system scripts
  - Used for job submission and monitoring

Complex Data Analysis and Processing

- Statistics, preprocessing, feature engineering
- Graphing
- Papers related to specific fields
- Big/Massive Data processing capabilities
- Understanding of data structures
- Data loading skills (e.g., how to load data larger than RAM)

Almost every science in modern times is data science

Need to read relevant literature to understand the efforts of predecessors

Manual Compilating and Linking of Dependent Programs

- Understand the compilation process
- Configure the compilation and running environment
- Process the dependencies of the program

Parallel Program Design, Testing and Optimization

- Design
- Test
- Optimize

Just as we discussed before, enjoy your following labs

Power Control and Parameter Adjustment

Why do we need power control?

- Mainly to meet the needs of competitions and practical applications
  - In some competitions we participated in, the total power of the cluster cannot exceed 3kW
  - In some practical applications, the power consumption of the cluster is expected to be as low as possible
- Need to obtain the optimal operating parameters

Methods:

- Adjust the frequency (CPU, GPU, Memory, etc.)
- Adjust the fan speed
- Adjust the operating scale (such as the batch size)
- ...

That's all for today

Thank you for your attention