EEP 524
Applied High-Performance GPU Computing

LECTURE 4 : Wednesday, January 27, 2021

Instructor: Dr. Colin Reinhardt

University of Washington - Professional Masters Program

Winter 2021

Lecture 3 : Outline

- Review and miscellaneous topics
  - GPI APIs
  - Nsight Visual Studio GPU debugger
  - Grid sizing
  - using nvcc
- AWS cloud GPU: Turing T4
- First look at GPU performance analysis
  - theory, practice, empirical
- Discussion

HW1 due Sun, 31 Jan by 11:59 PM (ought to be enough extensions!)

The GPU APIs

- CUDA
  - Runtime API
  - Driver API
  - CUDA C++ (kernel) language extensions
- OpenCL
  - Host API
  - C (kernel) language
- Comments
  - don't mix up datatypes or syntax between host and device
    - or between CUDA and OpenCL!
  - be sure to include the right headers and link libraries

Nsight Visual Studio: GPU Debugger

https://developer.nvidia.com/nsight-visual-studio-edition

- Stepping into CUDA kernel code on device is real!
- Powerful capabilities to use integrated Visual Studio debug with CUDA
  - view warps
  - view GPU registers
  - view per-thread kernel variables
  - more

Grid Sizing

- There are ways to automatically determine a reasonable (not always optimal) size for your execution configuration
  - blocks-per-grid (# workgroups)
  - threads-per-block (# workitems)
- Rules-of-Thumb (*often broken*)
  - threads-per-block even multiple of
    - warpsize (32)
    - total # SMs on device
  - Total grid size = nearest power-of-two

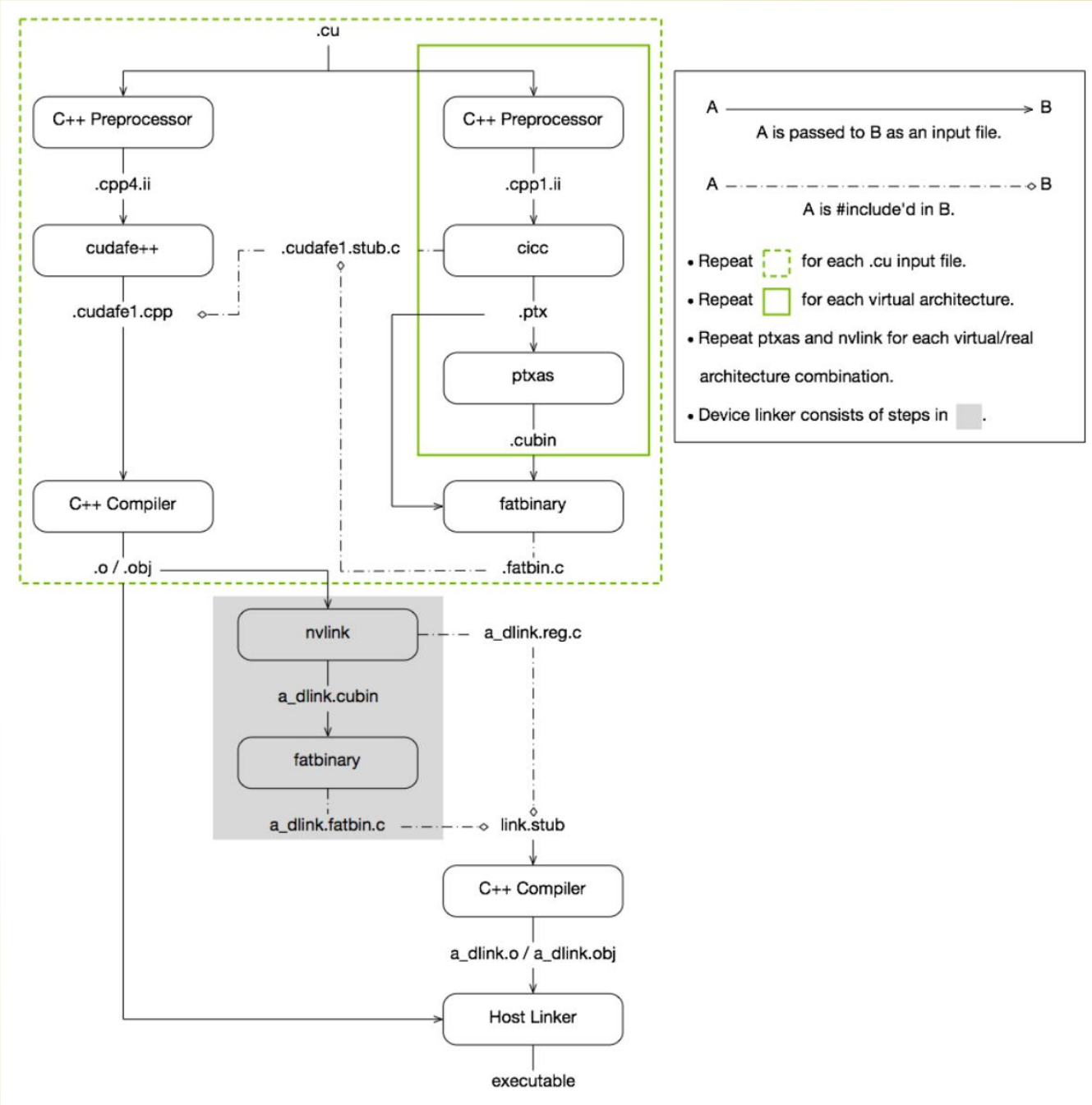
```
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
// OR
int blocksPerGrid = ceil(numElements/ threadsPerBlock;
my1Dkernel<<<blocksPerGrid,threadsPerBlock>>>(foo,bar);
```

Using nvcc : NVIDIA CUDA Compiler

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

- Simplest
  - `nvcc -o <output_file>.exe <input_path>/<input_file>.cu`
- A few more options
  - `nvcc -x cu -o <output_file>.exe -I <include_path> -l<libraries> <input_path>/<input_file>.cpp`
- Generating a PTX (intermediate assembly file)
  - `nvcc -o <output_file>.ptx <input_path>/<input_file>.cu -ptx`
- Compiling OpenCL
  - `nvcc -x cu -o oclquery.exe ocl_queryplatformdevice_hostapp.cpp -lOpenCL`
- Steering GPU Code Generation
  - `--gpu-architecture (-arch)  [-arch compute_75]`
    - controls preprocessing and compilation of input to PTX for a *virtual* architecture
    - at runtime, if no binary load image (real arch) is found, PTX is JIT-compiled
  - `--gpu-code (-code)  [-code sm_50]`
    - controls embedded binary load code image in resulting executable for specified real architecture
  - `--generate-code (-gencode)`
    - allows multiple PTX generations for different *virtual* architectures

CUDA compilation trajectory



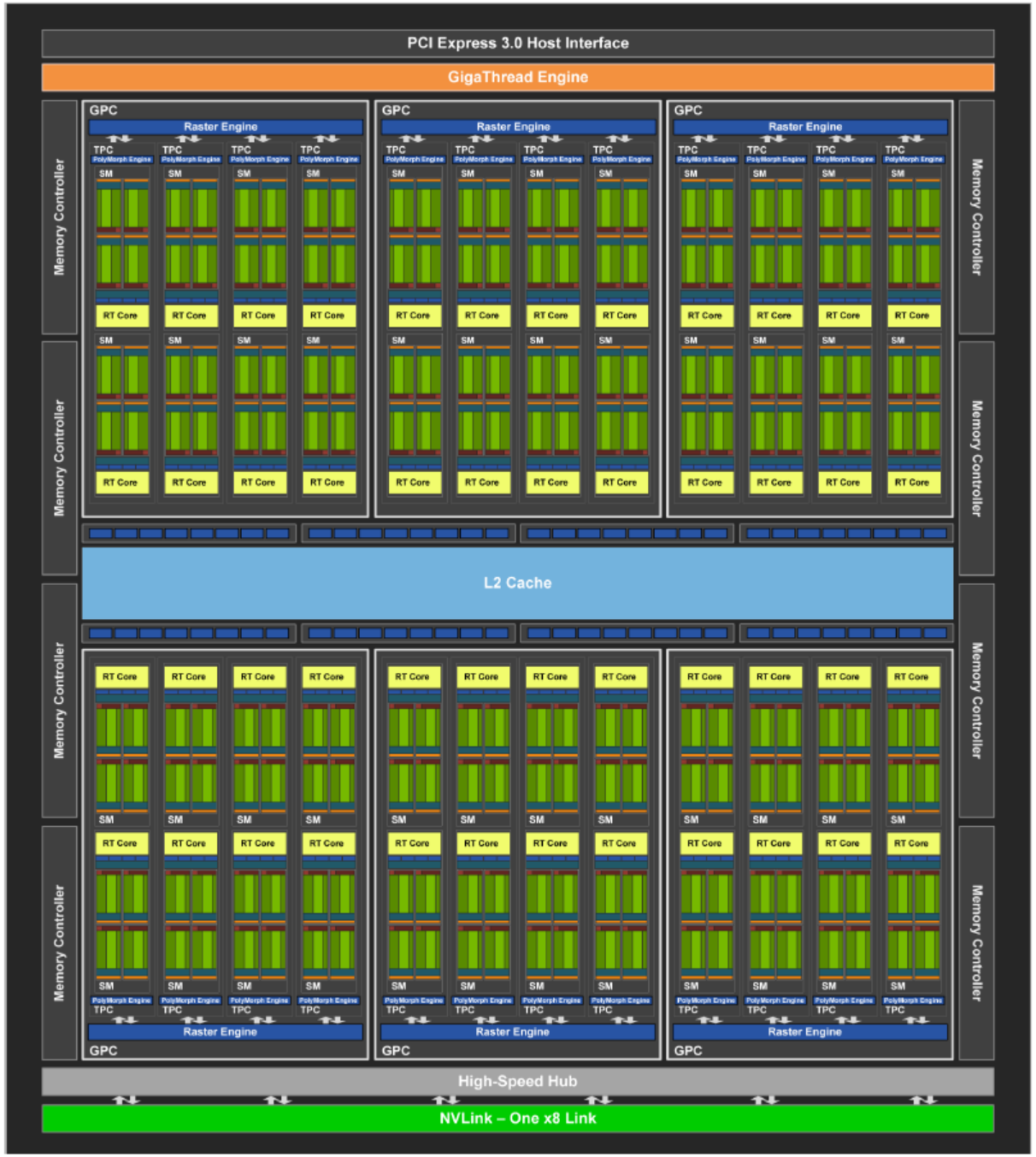
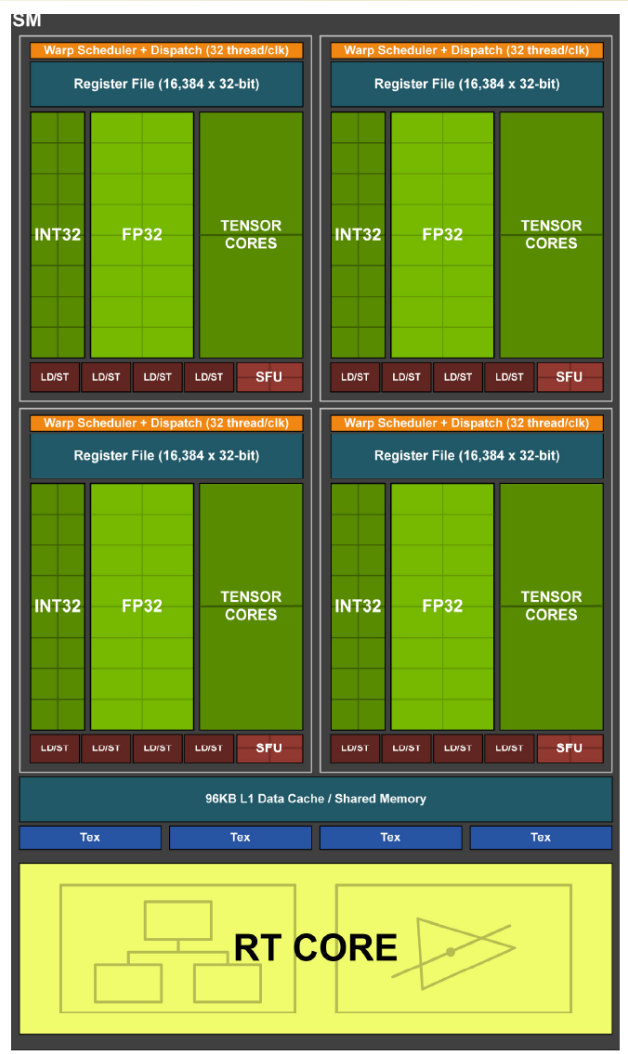


Figure 41. Turing TU104 Full Chip Diagram

Turing T4

- Unified shared memory/L1cache per SM = 96KB
- 64KB/32KB or 32KB/64KB
- L2 = 6MB
- Each SM
  - 64 FP32 + 64 INT32 cores
    - parallel execution pipes
  - 4 processing blocks
    - 16 FP32, 16 INT32 cores
    - 2 Tensor cores
    - 1 warp scheduler
    - 1 dispatch unit
    - L0 instruction cache
    - 64KB register file



Turing T4

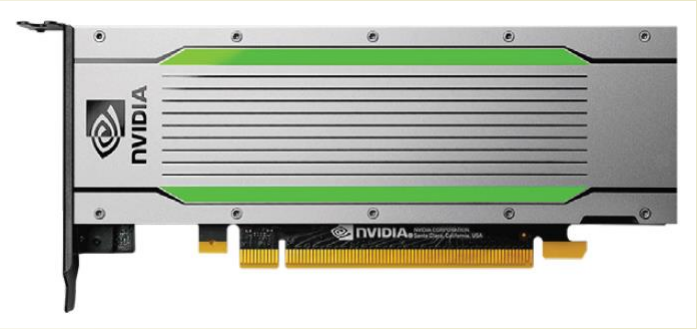


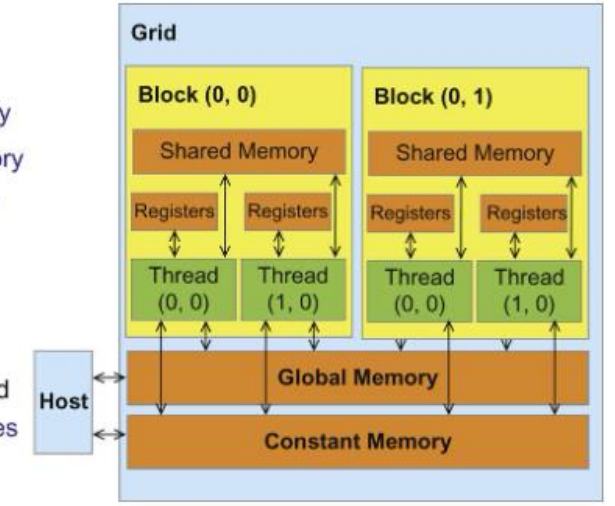
Table 5. Comparison of the Pascal Tesla P4 and the Turing Tesla T4

GPU	Tesla P4 (Pascal)	Tesla T4 (Turing)
GPCs	4	5
TPCs	20	20
SMs	20	40
CUDA Cores/SM	128	64
CUDA Cores/GPU	2,560	2,560
Tensor Cores/SM	NA	8
Tensor Cores/GPU	NA	320
RT Cores	NA	40
GPU Base Clock MHz	810	585*
GPU Boost Clock MHz	1,063	1,590
Peak FP32 TFLOPS	5.5	8.1
Peak INT32 TIPS	NA	8.1
Peak FP16 TFLOPS	NA	16.2
Peak FP16 Tensor TFLOPS with FP16 Accumulate*	NA	65
Peak FP16 Tensor TFLOPS with FP32 Accumulate*	NA	65
Peak INT8 Tensor TOPS*	22	130
Peak INT4 Tensor TOPS*	NA	260
Frame Buffer Memory Size and Type	8192 MB GDDR5X	16384 MB GDDR6
Memory Interface	256-bit	256-bit
Memory Clock (Data Rate)	6 Gbps	10 Gbps
Memory Bandwidth (GB/sec)	192	320
ROPs	64	64
TDP	75 Watts	70 Watts

- global mem: large & slow
- shared mem: small & fast

Device code can:
- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can
- Transfer data to/from per grid global and constant memories



- use shared mem when the overhead of moving data from global can be offset by repeated use of the data within the block

A first look at

GPU Performance Analysis

theory and practice

Growth of Functions

- Computational Complexity
  - Estimate growth of a function without worrying about (machine-dependent) constant multipliers and smaller order terms
    - Independent of hardware and software used to implement the algorithm!
  - *Time complexity*: time required to solve problem of particular size
  - *Space complexity*: memory required to solve problem of particular size
- Big-Omicron (big-O) Notation

$$O(g(n)) = \left\{ \begin{array}{c} f(n): \text{there exist positive constants } c, n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- Big-Omega (big-Ω) Notation

$$\Omega(g(n)) = \left\{ \begin{array}{c} f(n): \text{there exist positive constants } c, n_0 \text{ such that} \\ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- Big-Theta (big-Θ) Notation

$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n): \text{there exist positive constants } c_1, c_2, n_0 \text{ such that} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Growth of Functions

EXAMPLE Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$

$$0 \leq 8x \log x \leq 8x^2 \text{ for } x > 1$$

Thus it follows

$$3x^2 + 8x \log x \leq 11x^2 \text{ for } x > 1$$

So

$$3x^2 + 8x \log x \text{ is } O(x^2) \text{ for } c_2 = 11 \text{ and } n_0 = 1$$

Also

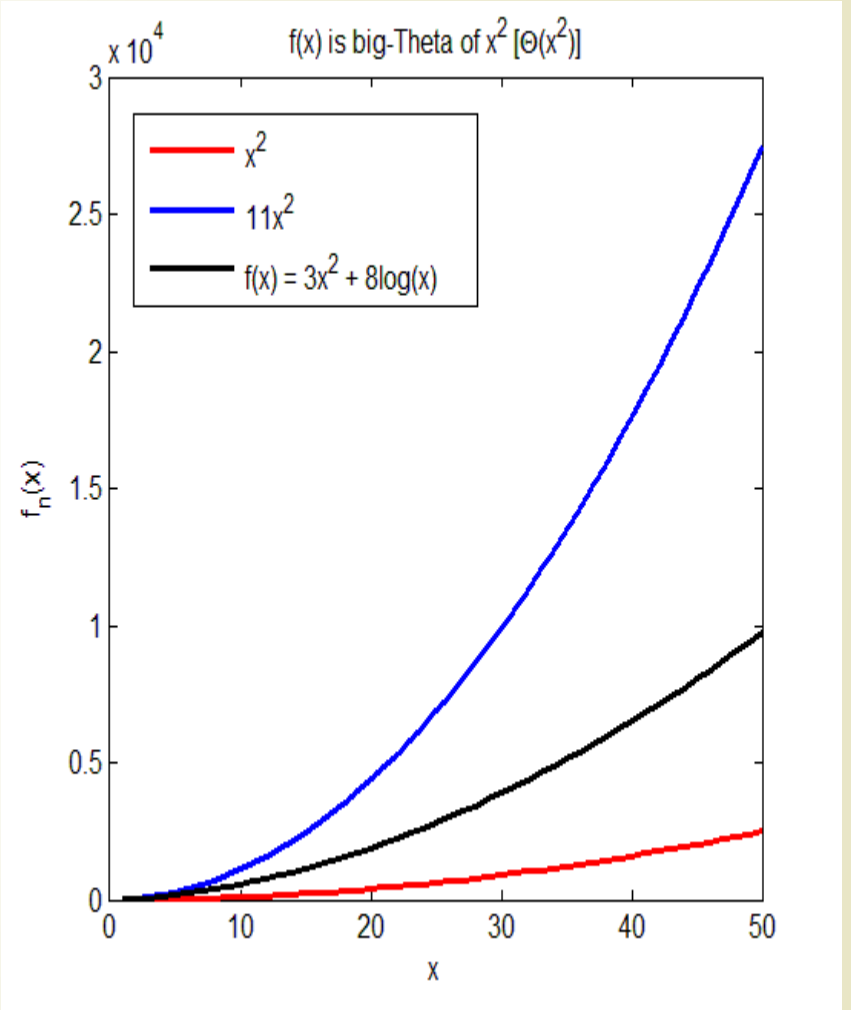
$$0 \leq x^2 \leq 3x^2 + 8x \log x \text{ for } x > 1$$

So

$$3x^2 + 8x \log x \text{ is } \Omega(x^2) \text{ for } c_1 = 1 \text{ and } n_0 = 1$$

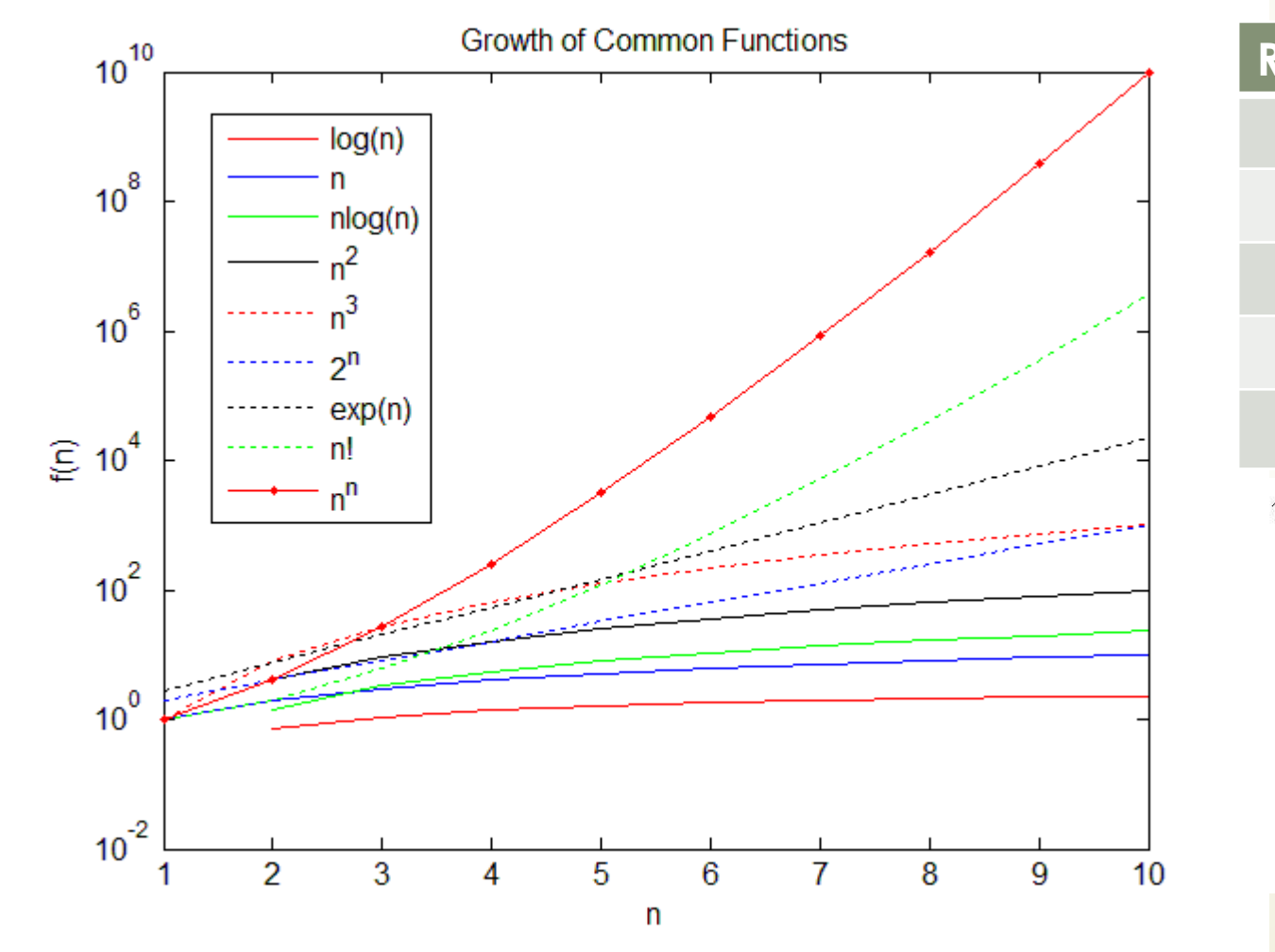
Therefore we have

$$0 \leq c_1 x^2 \leq f(x) \leq c_2 x^2 \text{ for } x > n_0 \text{ with } c_1 = 1, c_2 = 11, n_0 = 1 \text{ THUS } f(x) \text{is } \Theta(x^2) \blacksquare$$

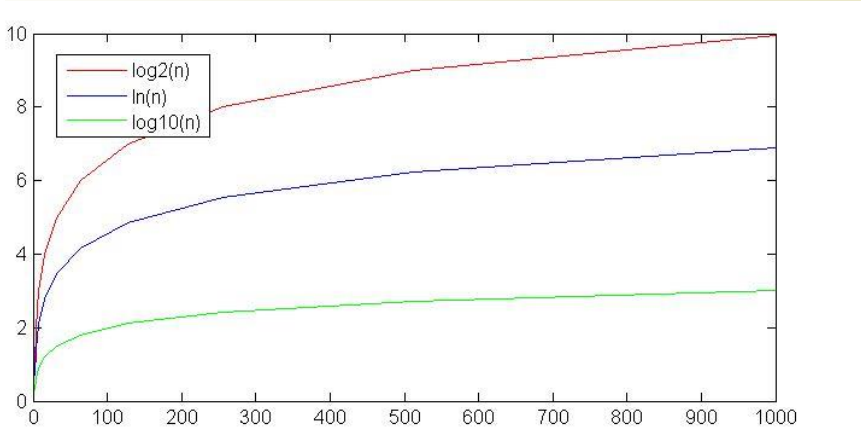


Growth of Functions

- Useful reference plots and terms for common function growth behaviors



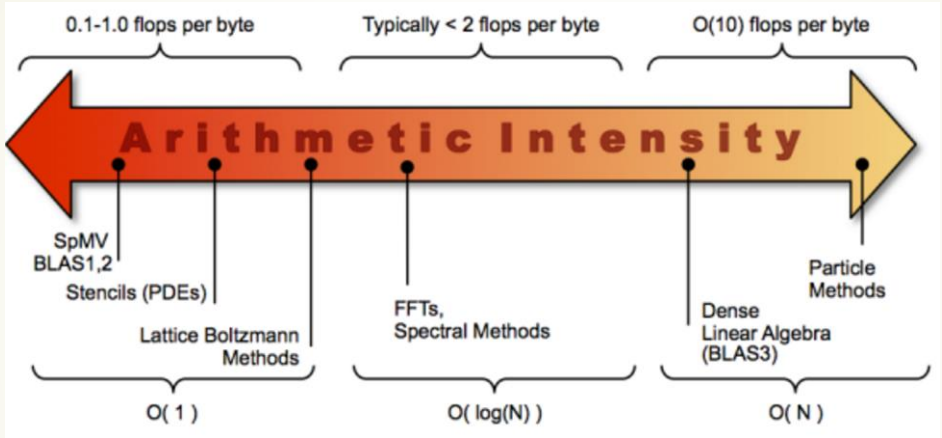
Running time	Is said to run in
$\Theta(1)$	constant time
$\Theta(\log n)$	logarithmic time
$\Theta(n)$	linear time
$\Theta(n^2)$	quadratic time
$O(f(n))$	polynomial time



Performance Theory and Metrics

Arithmetic Intensity : $AI = \frac{\text{computational work}}{\text{communication}}$

$$AI_{practical} = \frac{\text{FLOPS}}{\text{bytes memory traffic}}$$



Example:

Hypothetical device with **100 Gb/s** memory bandwidth (i.e. 25 Gfloats/s)

And **625 GFLOP/s** peak single-precision arithmetic throughput

What AI for kernel to reach both peak arithmetic and memory throughputs?

AI = 625/100 = 6.25 FLOP per byte memory transaction

(Or 625/25 = 25 FLOP/float memory transaction)

BLAS: Basic Linear Algebra Subroutines

- Categorized into three sets of routines
  - correspond to both chronological order of definition/publication and degree of the polynomial in complexity of algorithms

- Level 1 (1979): operations take *linear* time $O(n)$
  - vector operations on strided arrays: dot products, vector norms
  - generalized vector addition
    - S/DAXPY $\boldsymbol{y} = a\boldsymbol{x} + \boldsymbol{y}$
- Level 2 (1988): operations take *quadratic* time $O(n^2)$
  - matrix-vector operations, including generalized matrix-vector multiplication
    - GEMV: $\boldsymbol{y} = a\mathbf{A}\boldsymbol{x} + b\boldsymbol{y}$
- Level 3 (1990): operations take *cubic* time $O(n^3)$
  - matrix-matrix operations, including generalized matrix multiplication
    - GEMM: $\boldsymbol{y} = a\mathbf{AB} + b\mathbf{C}$

Arithmetic Intensity (AI): Examples

Evaluate AI for 3 standard algorithms from BLAS Levels 1-3

case	BLAS function	
1	DAXPY	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$
2	DGEMV	$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$
3	DGEMM	$\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$

1. Find computational work **W** (FLOPs)
   - Work is property of chosen algorithm and is independent of platform
   - In these (above) cases work depends only on input size: W=W(n)
2. Find communication = memory traffic **Q** (bytes)
   1. Unlike W, Q depends heavily on details of platform, such as cache hierarchy design. Initially, we'll make the "*perfect cache*" assumption
   2. $Q(n) = Q_r(n) + Q_w(n)$: read and write data traffic differs

Calculating FLOP/s

- typically FLOP = one floating-point operation.
  - multiplication, addition, subtraction, comparison of two **single**-precision floating point numbers
  - division, square-root, other complex operations must be considered separately, they are more costly in terms of clock-cycles
  - **double**-precision operations generally take twice as long
- Example: basic Matrix Multiplication
- **MIPS**: Million-of-Instructions-Per-Second: measure of integer performance

Arithmetic Intensity: Examples

case	BLAS function	
1	DAXPY	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$
2	DGEMV	$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$
3	DGEMM	$\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$

Empirical performance data capture

- Nsight Compute CLI (command line interface)

// NSight Compute Profiling CLI for AI: time, FLOPs, mem-bytes

```
ncu -k vectorAdd --metrics
smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum,smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,dram__bytes.sum,lts__t_bytes.sum,l1tex__t_bytes.sum,sm__cycles_elapsed.avg,sm__cycles_elapsed.avg.per_second ./vecadd_cudart.exe
```

// RESULTS: for simple vectorAdd kernel

```
[Vector addition of 33554432 float elements]
CUDA kernel launch with 131072 blocks of 256 threads
==PROF== Profiling "vectorAdd": 0%....50%....100% - 3 passes
Test PASSED. All elements agree between CPU and GPU within tolerance threshold ftol = 0.00001

[1500] vecadd_cudart.exe@127.0.0.1
  vectorAdd(float const*, float const*, float*, int), 2021-Jan-25 23:24:27, Context 1, Stream 7
    Section: Command line profiler metrics
    ------------------------------------------------------------------- ------------- ------------------------------
    dram__bytes.sum                                                     Mbyte                      439.45
    l1tex__t_bytes.sum                                                  Mbyte                      402.65
    lts__t_bytes.sum                                                    Mbyte                      402.71
    sm__cycles_elapsed.avg                                              cycle                      891,502.60
    sm__cycles_elapsed.avg.per_second                                   cycle/usecond              584.99
    smsp__sass_thread_inst_executed_op_fadd_pred_on.sum                 inst                       33,554,432
    smsp__sass_thread_inst_executed_op_ffma_pred_on.sum                 inst                       0
    smsp__sass_thread_inst_executed_op_fmul_pred_on.sum                 inst                       0
    ------------------------------------------------------------------- ------------- ------------------------------
```

Nsight Compute CLI Profiling

https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

SETS contain SECTIONS contain METRICS

- `ncu --list-sets`
- `ncu --list-sections`
- `ncu --query-metrics  (there are LOTS, 100s)`

- Profile using a set and a section for a kernel <kernel_name>
  - console output results
- `ncu -k vectorAdd --set default --section MemoryWorkloadAnalysis cudart_vecadd.exe`
  - output results to a report file
- `ncu -k vectorAdd -o vecadd_profile_rpt --set default --section MemoryWorkloadAnalysis cudart_vecadd.exe`