Introduction to CUDA

Jongsoo Kim
Korea Astronomy and Space Science Institute
@The 13th KIAS CAC School

June 2019, www.top500.org

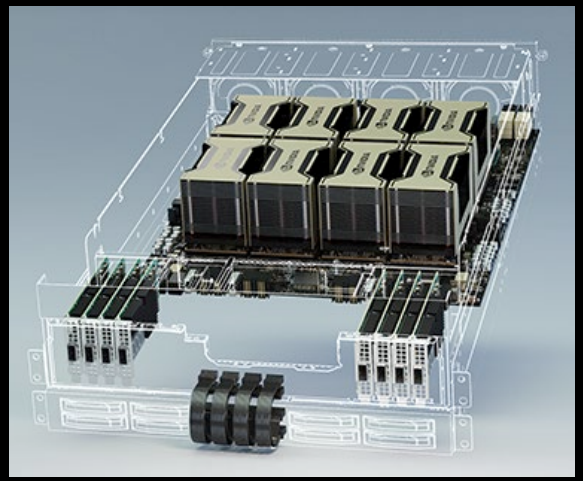
Summit #4, 148.6 Pflos/s, 10.1MW
two 22-core IBM Power 9 CPU
+six NVIDA GV100 GPUs

June 2022, www.top500.org

Selene #8, 63.4 Pflos/s, 2.646MW
based on NVIDIA DGX SUPERPOD

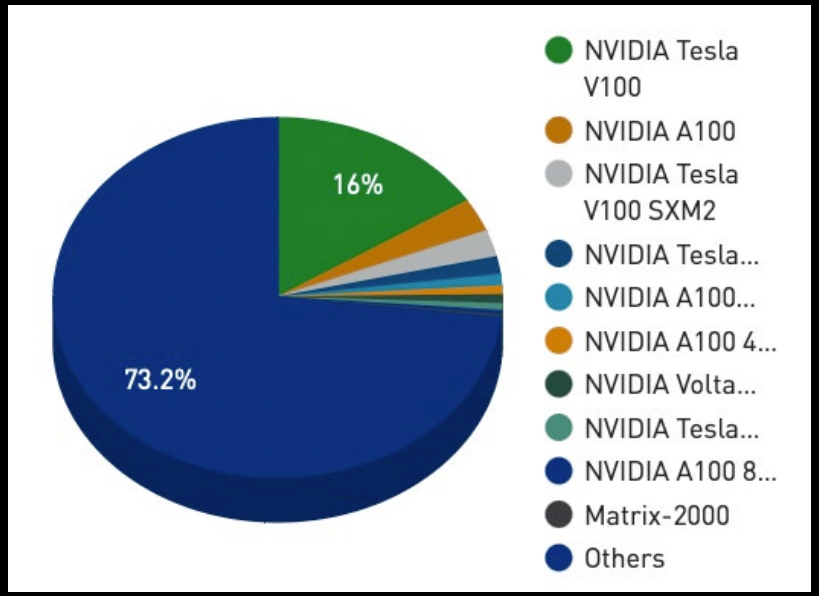
AMD EPYC 7742 64C 2.25 GHz
NVIDA A100 GPUs
Mellanox HDR Infiniband

SUPERPOD #17 GREEN500
26.2 Gflops/watt

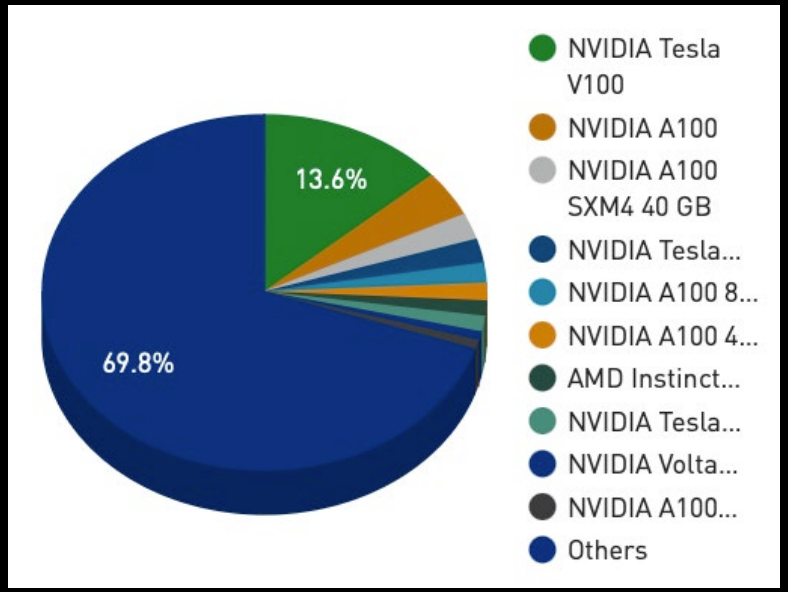


Accelerator/Co-Processor Performance Share

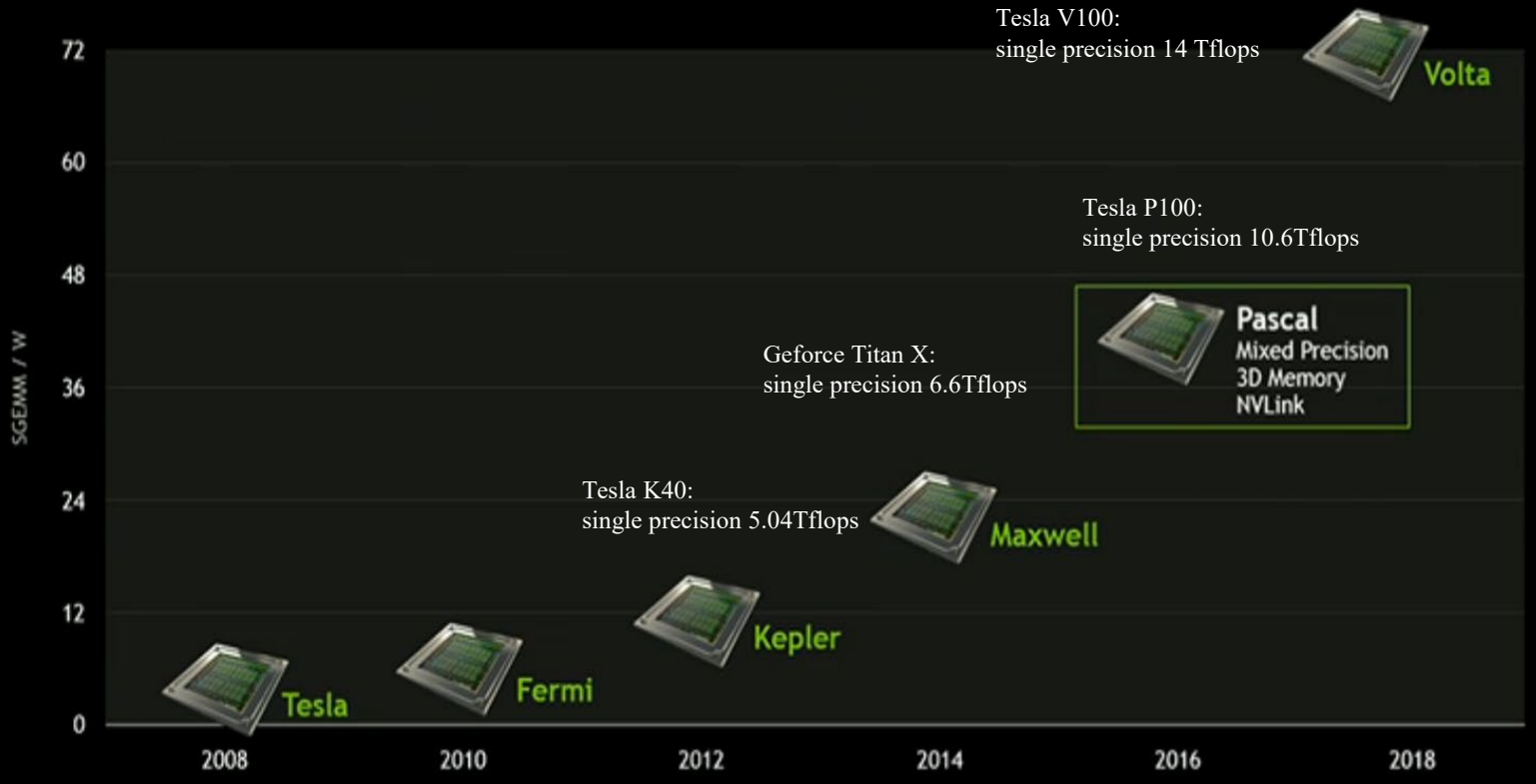
November 2021



June 2022



NVIDIA GPU Roadmap



Astronomy Applications of GPU

- N-body simulations
- Fluid HD and MHD simulations
- Radiative Transfer
- Data processing, e.g., radio astronomy
- AI applications
- # of papers searched in ADS with a keyword “GPU” in abstract

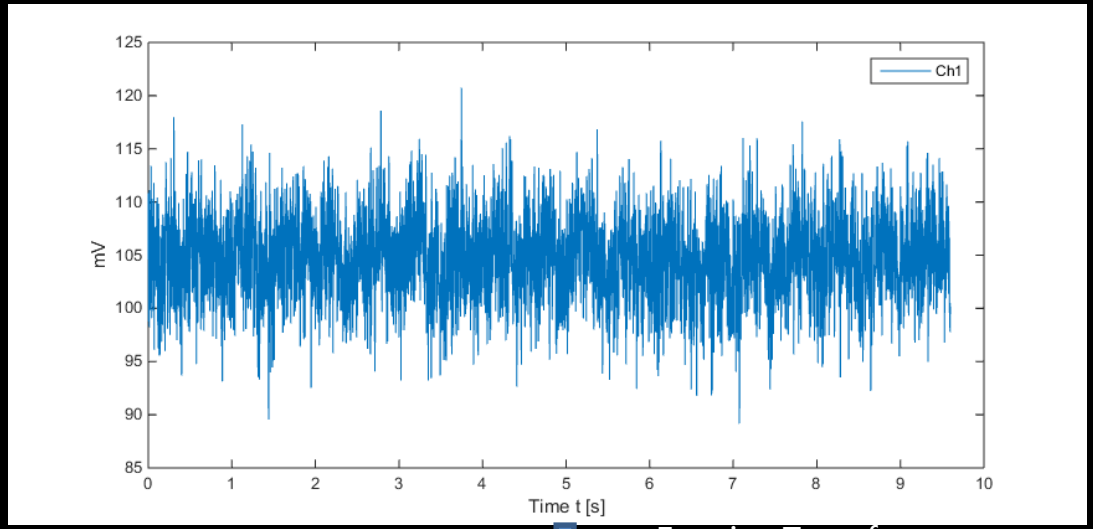
yr	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21
#	6	8	11	37	41	70	57	64	61	74	65	91	76	160	161

Atacama Large Millimeter/ Submimllimter Array

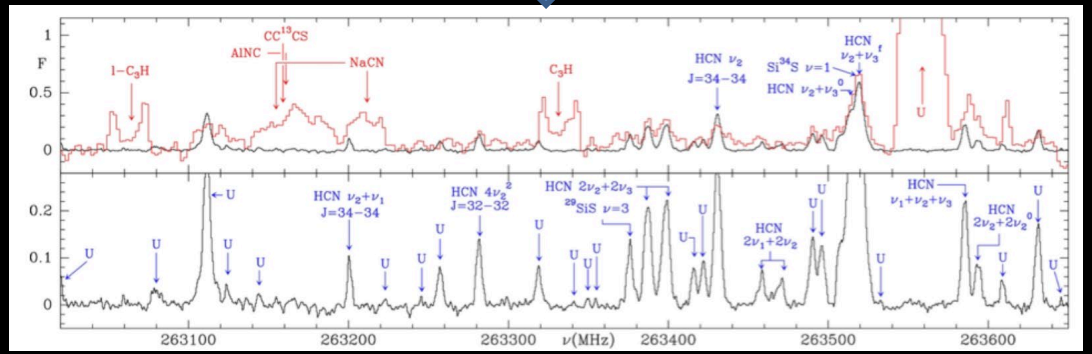
- 12m array: 50 x 12m antennas
- ACA array: 12 x 7m antennas + 4 x 12m antenna
- Longest distance between two antennas: 16 km

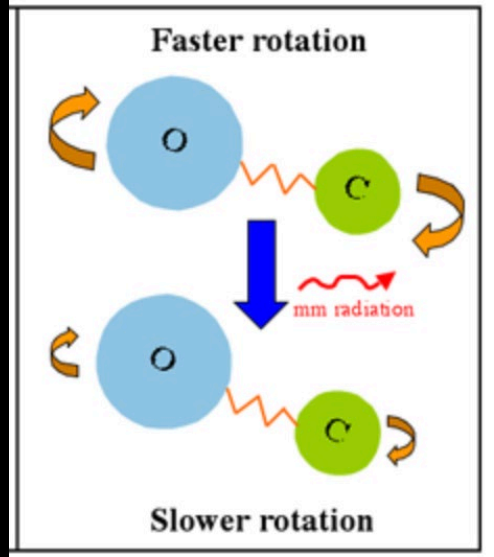
The ALMA Site

How to get a radio spectrum



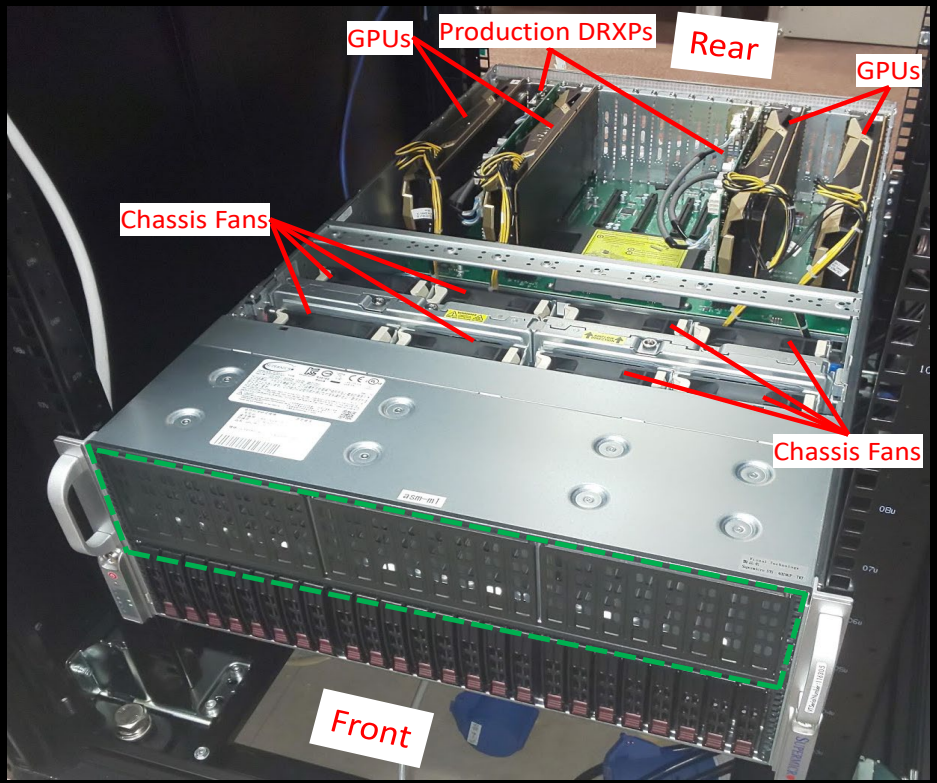
Fourier Transform

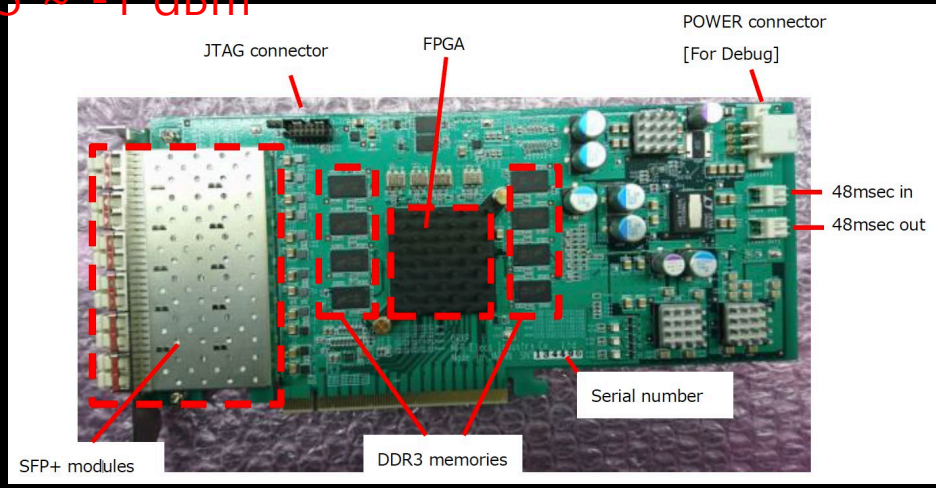




ASM; DRXP; GPU

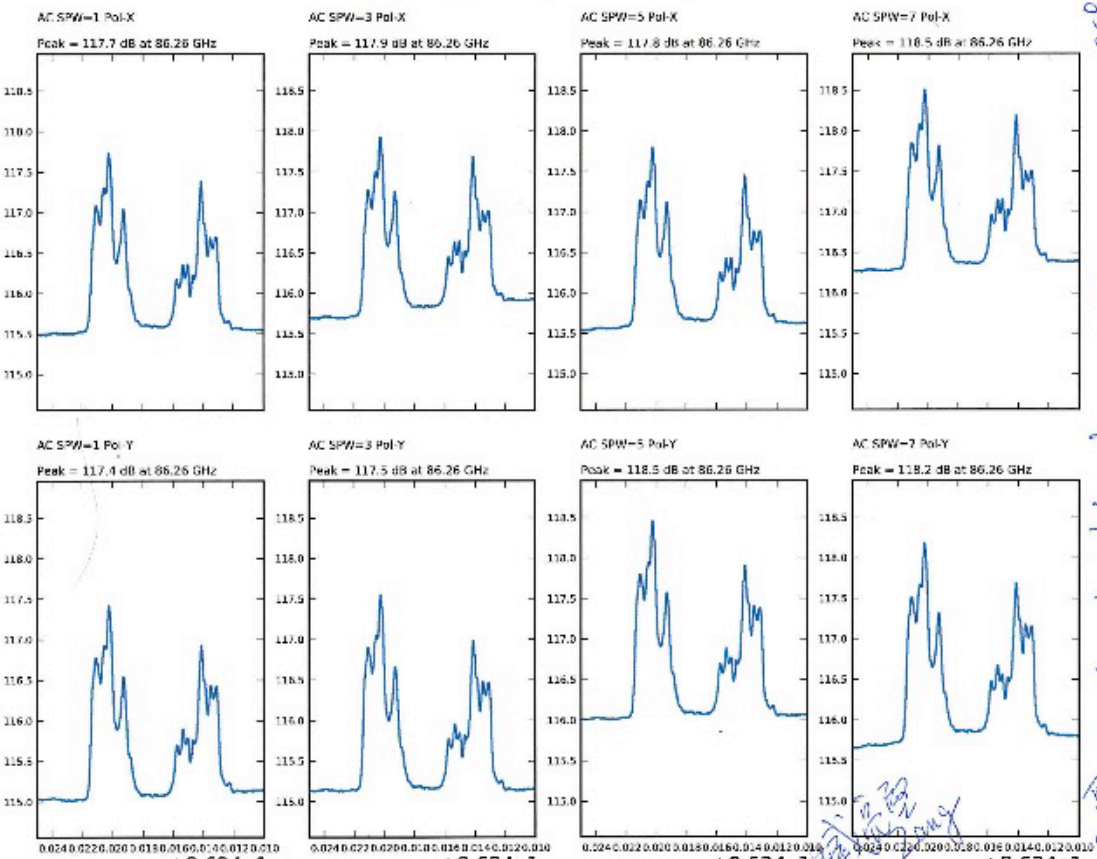
Acceptable signal level
of a transceiver
-15.5 ~ -1 dBm





86
Ori
202

uid___A002_Xf59d27_X42 Power Spectra



Jongsoo Kim 2022-04-22
I am happy to get these spectra. I appreciate many people to help this (PI)
Manoby Watanabe
Unexpected by great achievement.
Congratulation to the Team!
Kawamura
Scott
Jongsuk
Shuri Ishii
Congratulations on your achievement! :D
Carlo Goldoni

Serial Programming

- Languages
  - Fortran, C/C++, python, …
- Optimization of Serial Codes
  - Compilers
  - opts flags
  - math kernel, IPP (integrated Performance Primitives), ….
- Intel AVX-512 (Advanced Vector Extensions)
  - vector operations capabilities

Parallel Programming

- OpenMP, OpenACC
  - SMP, incremental parallelization
- CUDA (openCL)
  - Coprocessor, incremental parallelization
- MPI
  - SMP or cluster, needs sometimes many changes of serial codes

Hybrid Programming

- OpenMP (MPI) + CUDA
  - SMP, multiple GPUs
- MPI + OpenMP
  - Cluster of SMP nodes
- MPI + CUDA
  - Cluster, multiple GPUs
- MPI + OpenMP + CUDA

CUDA from Wikipedia

- CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing platform and application programming interface(API) model created by Nvidia.
- It allows software developers and software engineers to use a CUDA-enabled graphics processing unit(GPU) for general purpose processing – an approach termed GPGPU(general-purpose computing on graphics processing units).
- The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

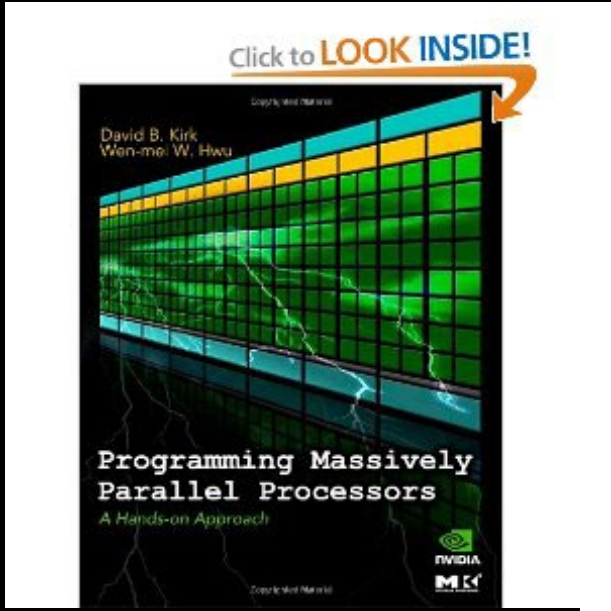
CUDA Ecosystem

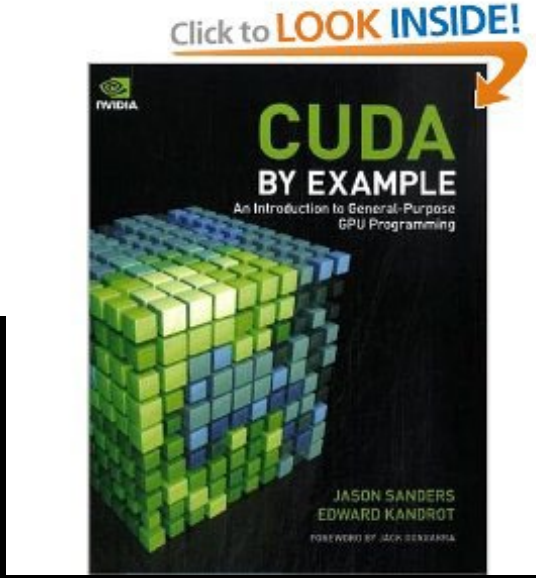
- Programming languages and APIs
  - CUDA: C, C++, Fortran, Python, etc.
- Libraries
  - Mathematical libraries: cuBLAS, cuRAND, cuFFT, cuSPARSE, cuTENSOR, cuSOLVER
  - Parallel algorithm libraries: nvGRAPH, Thrust
  - Image and video libraries: nvJPEG, NPP, Optical Flow SDK
  - Communication libraries: NVSHMEM, NCCL
  - Deep learning libraries: cuDNN, TensorRT, Jarvis, DALI
  - Partner libraries: OpenCV, FFmpeg, ArrayFire, MAGMA
- Profiling and debugging tools
  - NVIDIA Nsight: profiling, tracing, and debugging tool.
  - CUDA GDB: extension of Linux GDB, a console-based debugging interface
  - CUDA-Memcheck: memory access issues by examining the thousands of threads running concurrently.

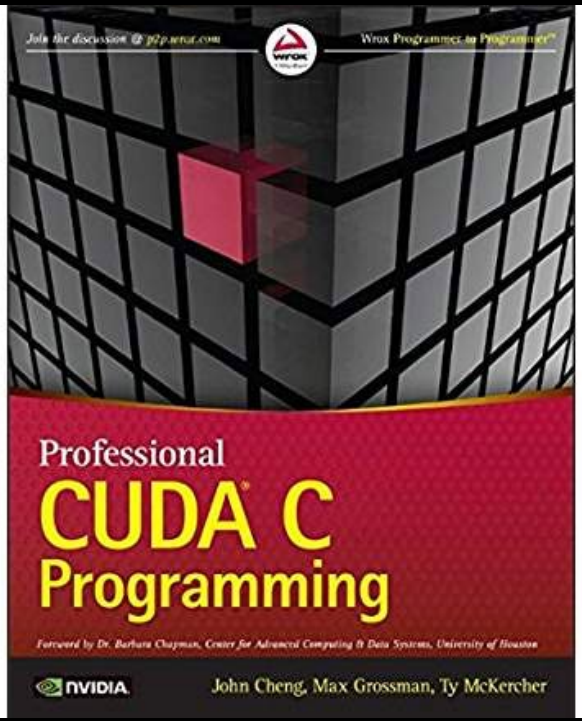
http://developer.nvidia.com

- Document center
  - CUDA Toolkit
  - CUDA Libraries
  - …
- Developer Blog
  - AI/Deep Learning
  - Autonomous Vehicles
  - HPC
  - Data Science
  - Graphics / Simulation
  - Networking
  - …

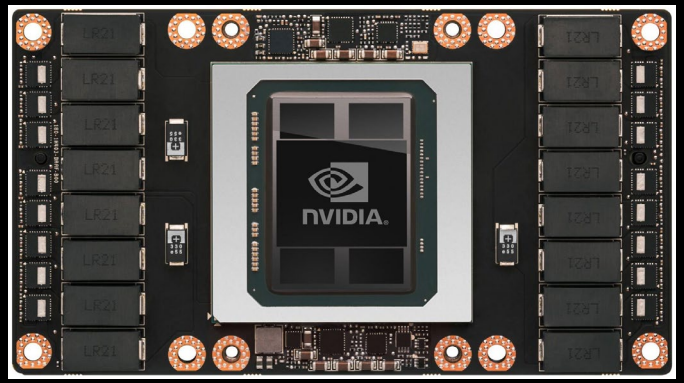
Books for CUDA

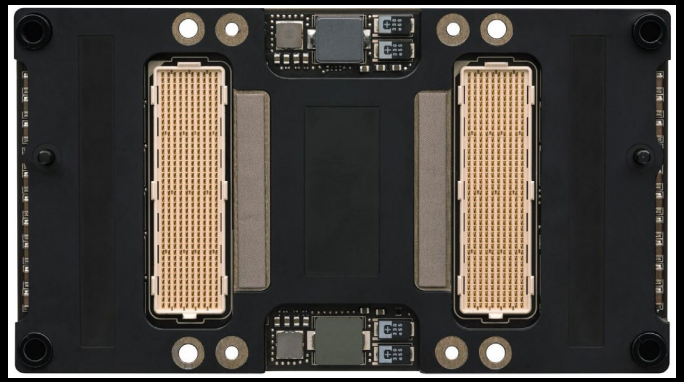






NVIDA TESLA V100 SXM2-32GB

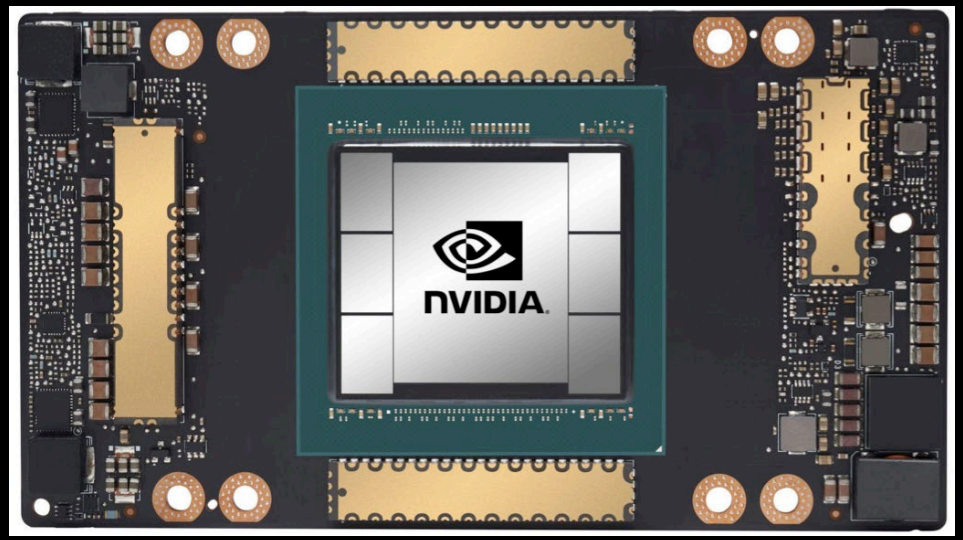




- Volta Architecture
- Cores: 5140 FP32+640 Tensor
- Memory: : 32 GB HBM2, 900GB/s ban dwidth
- 2nd Gen NVLink: 300 GB/s
- PCIe gen3: 32 GB/s bi-direction
- Max Power: 300 W

- 7.8 TFLOPS of FP64
- 15.7 TFLOPS of FP32
- 125 TFLOPS of Tensor

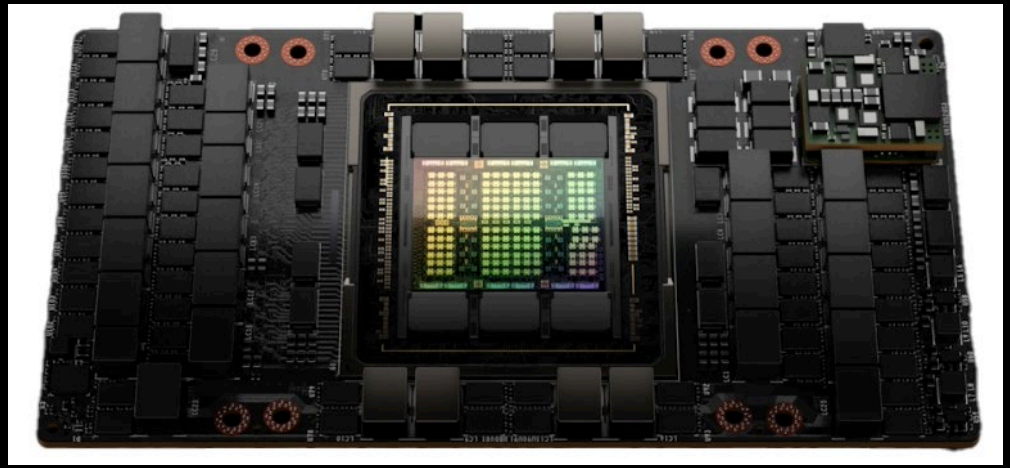
NVIDA TESLA A100 SXM4 80GB



- Ampere Architecture
- Cores: 6912 FP32+432 Tensor
- Memory: 80 GB, 2039GB/s bandwidth
- 3rd Gen NVLink: 600 GB/s
- PCIe gen4: 64 GB/s
- Max Power: 400 W

- 9.7 TFLOPS using FP64
- 19.5 TFLOPS using FP32
- 78 TFLOPS using FP16
- 156 TFLOPS using FP32 Tensor Core

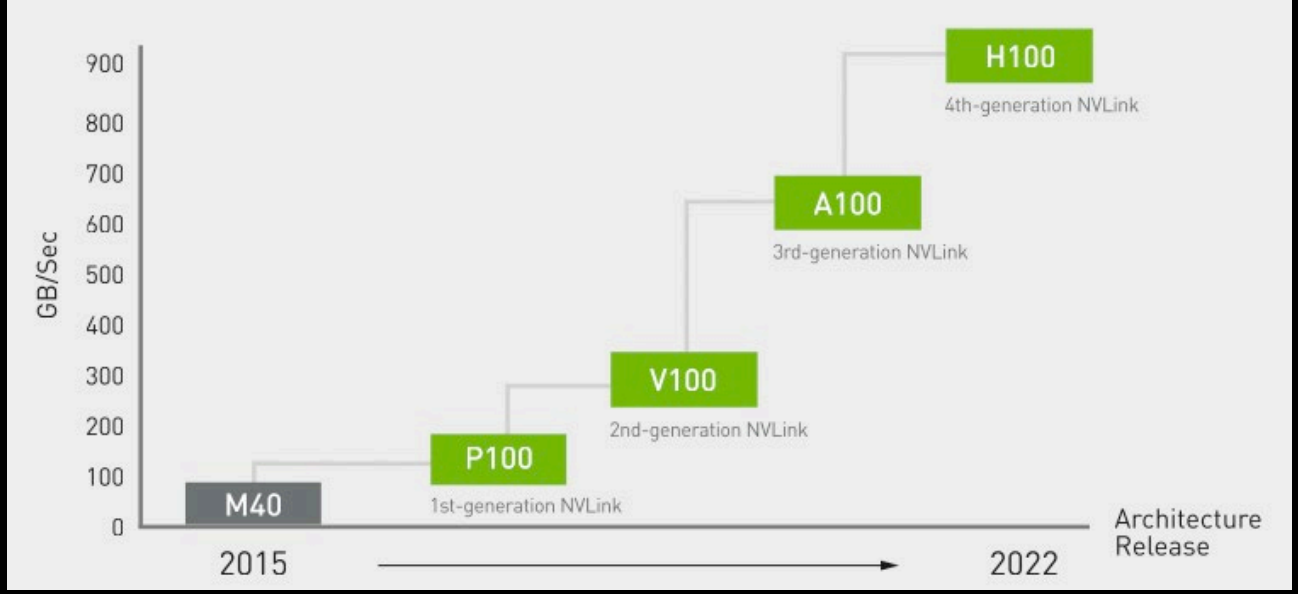
NVIDA TESLA H100 SXM5



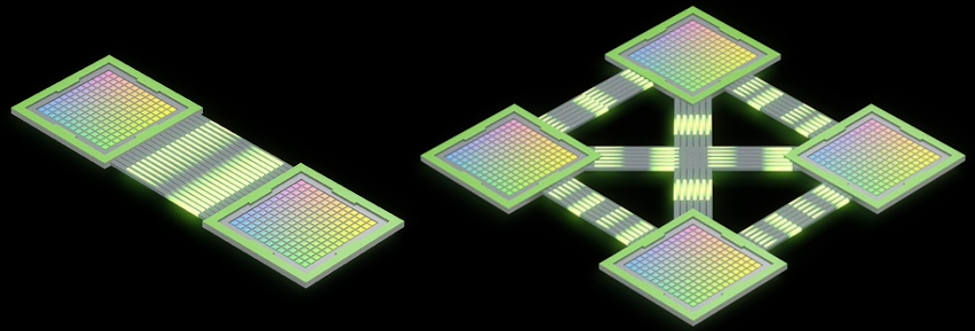
- Hopper Architecture
- Cores: 16896 FP32+432 Tensor
- Memory: 80 GB HBM3
- 4th Gen NVLink: 900 GB/s
- PCIe gen5: 128 GB/s
- Max Power: 700 W

- 30 TFLOPS using FP64
- 60 TFLOPS using FP32
- 120 TFLOPS using FP16
- 500 TFLOPS using FP32 Tensor Core

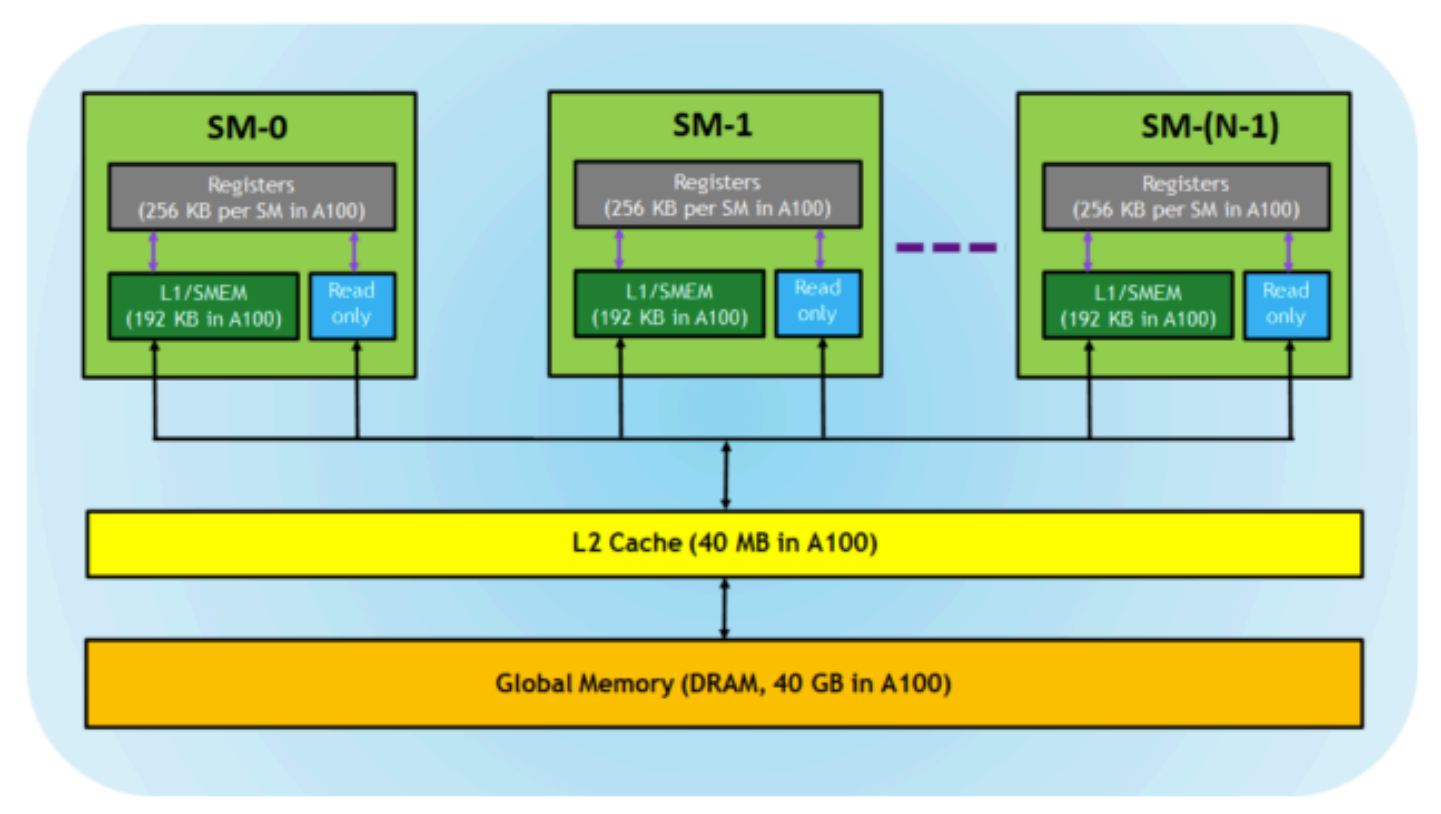
NVLINK



- Total bandwidth of 900 GB/s
- 50GB/s bi-direction per link
- 18 NVLINK connections



GPU memory hierarchy



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

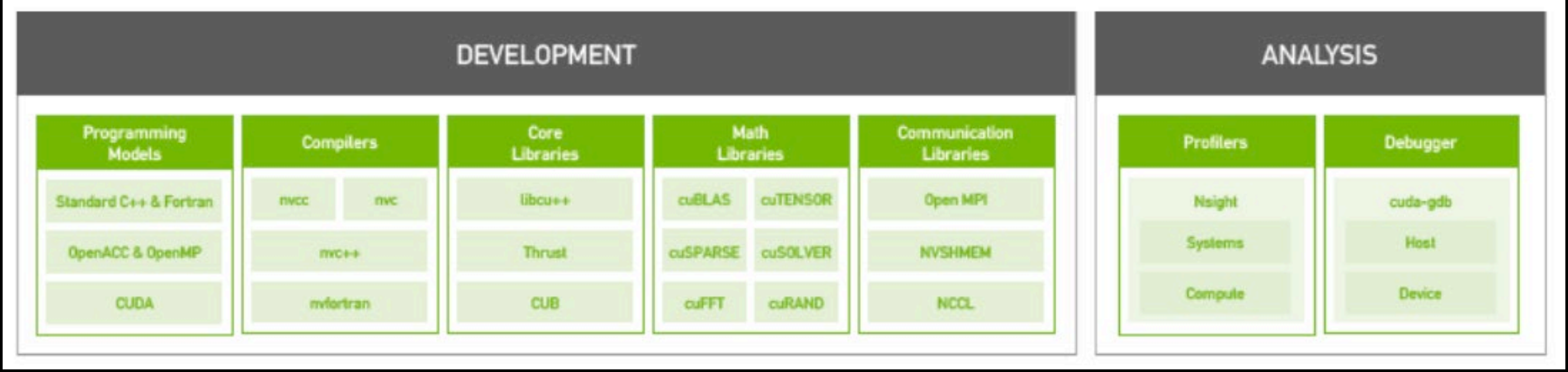
Table 15. Technical Specifications per Compute Capability

Technical Specifications	Compute Capability												
	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16	128		
Maximum number of resident blocks per SM	16				32						16	32	16
Maximum number of resident warps per SM	64										32	64	48
Maximum number of resident threads per SM	2048										1024	2048	1536
Maximum number of 32-bit registers per thread block	64 K				32 K	64 K		32 K	64 K				
Maximum number of 32-bit registers per thread	255												
Maximum amount of shared memory per SM	48 KB	112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB		64 KB	164 KB	100 KB
Maximum amount of shared memory per thread block [31]	48 KB								96 KB	48 KB	64 KB	163 KB	99 KB

See Table 14 and 15 https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

NVIDIA HPC SDK

A Comprehensive Suite of Compilers, Libraries and Tools for HPC



Lexicon, 10-node GPU cluster

- CPU: Intel(R) Xeon(R) Gold 5118(2.3GHz 12Cores) * 2
- GPU: Nvidia Tesla V100 32GB * 4
- Memory: 192GB
- INTERCONNECT: 100G Infiniband, 10G Ethernet
- OS : Ubuntu 18.04.2
- Scheduler: slurm

Login to one of Lexicon GPU server

- Login to one of GPU servers, Lex0[1~5]
  (본인 주민등록번호 끝자리 / 5) +1
- Follow the login directions sent by the KIAS

cuda-install-samples

- `cd ~`
- `cuda-install-samples-10.0.sh .`

→Generate ~/NVIDIA_CUDA-10.0_Samples

deviceQuery

- `cd ~/NVIDIA_CUDA-10.0_Samples/1_Utilities/deviceQuery`
- `make`
- `deviceQuery`

Bandwidth Test

- `cd ~/NVIDIA_CUDA-10.0_Samples/1_Utilities/bandwidthTest`
- `make`
- `./bandwidthTest`

Copy programs to your home directory

- `cd ~`
- `cp -r ~cac004/cuda_tutorial/ .`

Exercise: Hello world

```
#include <stdio.h>

void hello_world(void) {
     printf(“Hello World\n”);
}

int main (void) {
    hello_world();
    return 0;
}
```

Exercise: Hello world

```
#include <stdio.h>

__global__ void hello_world(void) {
     printf(“Hello World\n”);
}

int main (void) {
    hello_world<<<1,5>>>();
    cudaDeviceReset();
    return 0;
}
```

C Language Extensions

- Function Type Qualifiers
  - __global__
    - executed on the device (GPU)
    - callable from the host (CPU) only
    - functions should have void return type
    - any call to a __global__ function must specify the execution configuration for that call
  - __device__
    - executed on the device
    - callable from the device only
    - …

Compile and Run

- `nvcc hello.cu`
- `./a.out`

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

- Run hello.cu with different combinations of numbers in <<<1,5>>>

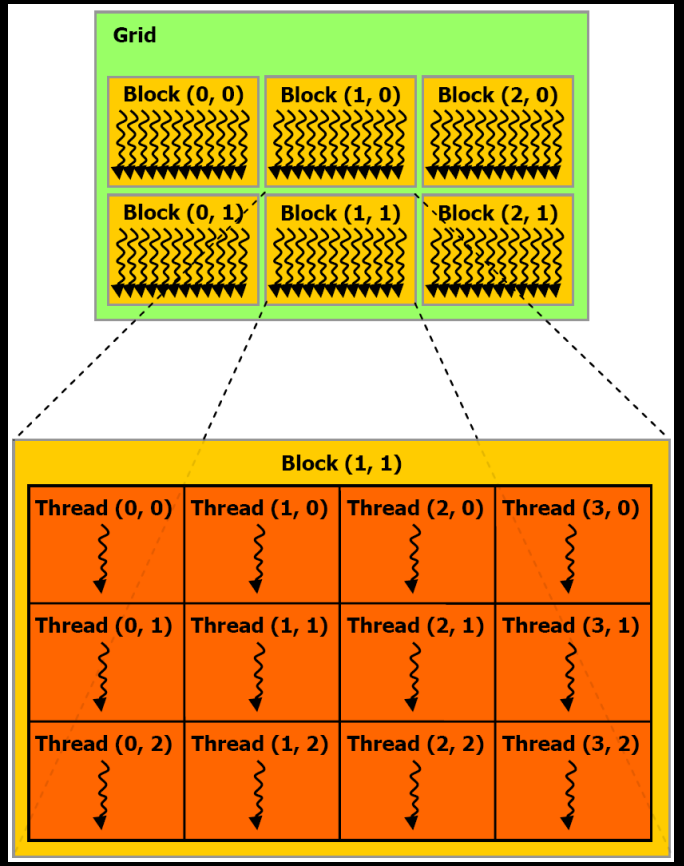
Grid, Block, Thread

- Maximum dimension size of each block in a grid

  (2147483647, 65535, 65535)
- maximum # of threads per block

  1024
- Maximum dimension size of each thread in a block

  (1024)x(1024)x(64)



Built-in variables and vector type

- Built-in Variables

  threadIdx = (threadIdx.x, threadIdx.y, threadIdx.z)

  thread index in each direction

  blockIdx = (blockIdx.x, blockIdx.y, blockIdx.z)

  block index in each direction

  blockDim = (blockDim.x, blockDim.y, blockDim.z)

  number of threads in each direction

  gridDim = (gridDim.x, gridDim.y, girdDim.z)

  number of blocks in each direction

- Built-in Vector type

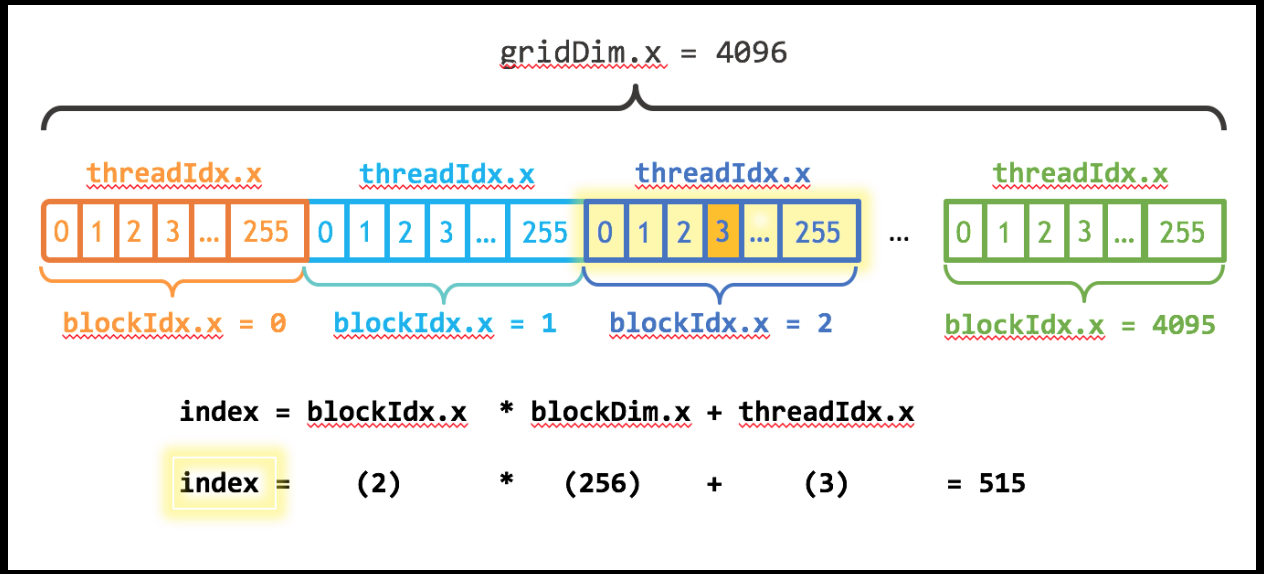
  dim3:

  Integer vector type based on unit3 used to specify dimensions

Execution configuration(1)

<<<blocksPerGrid,threadsPerBlock>>>

<<<4096,256>



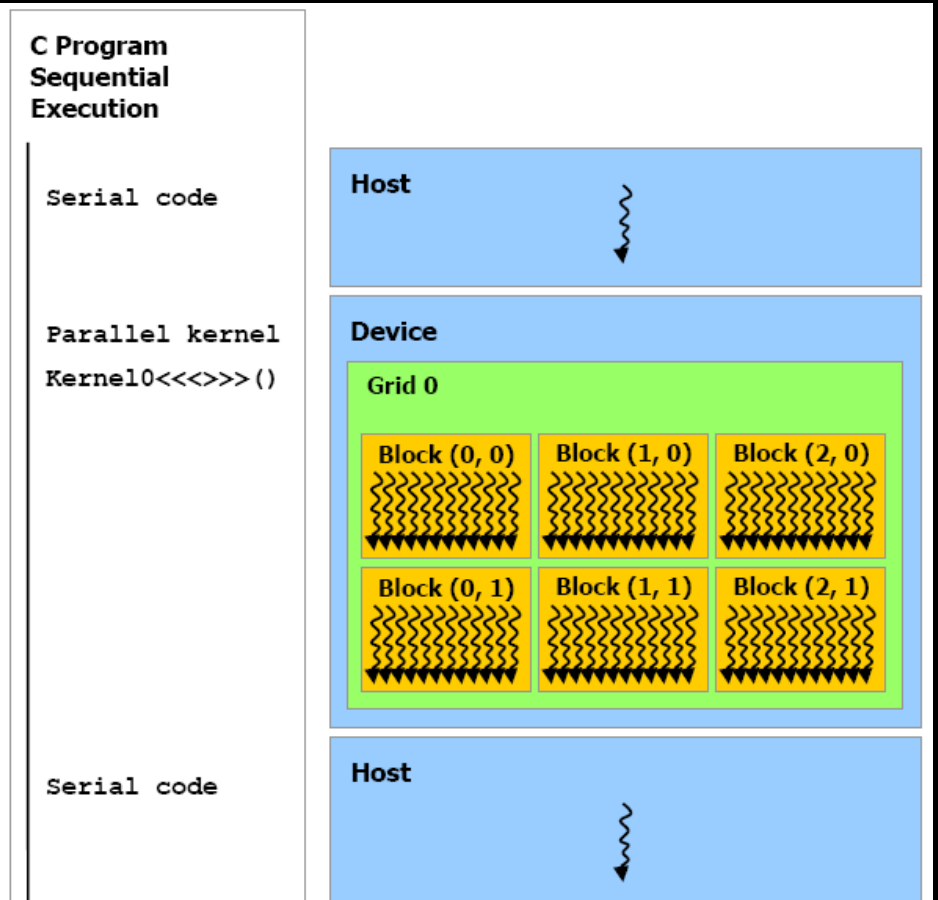
Execution configuration (2)

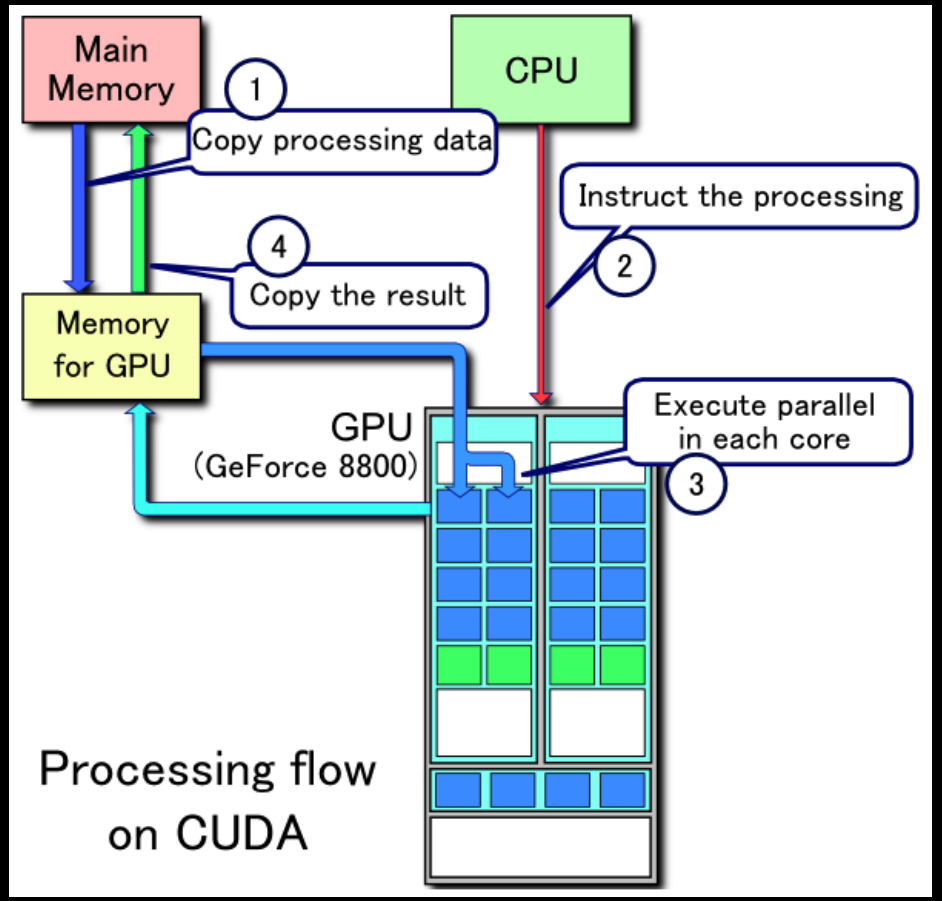
```
dim3 blocksPerGrid(16,16,1)
dim3 threadsPerBlock(1024,1,1)
<<<blocksPerGrid,threadsPerBlock>>>

dim3 blocks(4096,16,2)
dim3 threads (256,2,2)
<<<blocks,threads>>>
```

Exercise: Execution Configuration

- nvcc exec_conf_1d.cu
- ./a.out
- nvcc exec_conf_2d.cu
- ./a.out
- Run the two programs with different thread and block numbers in the execution configuration.

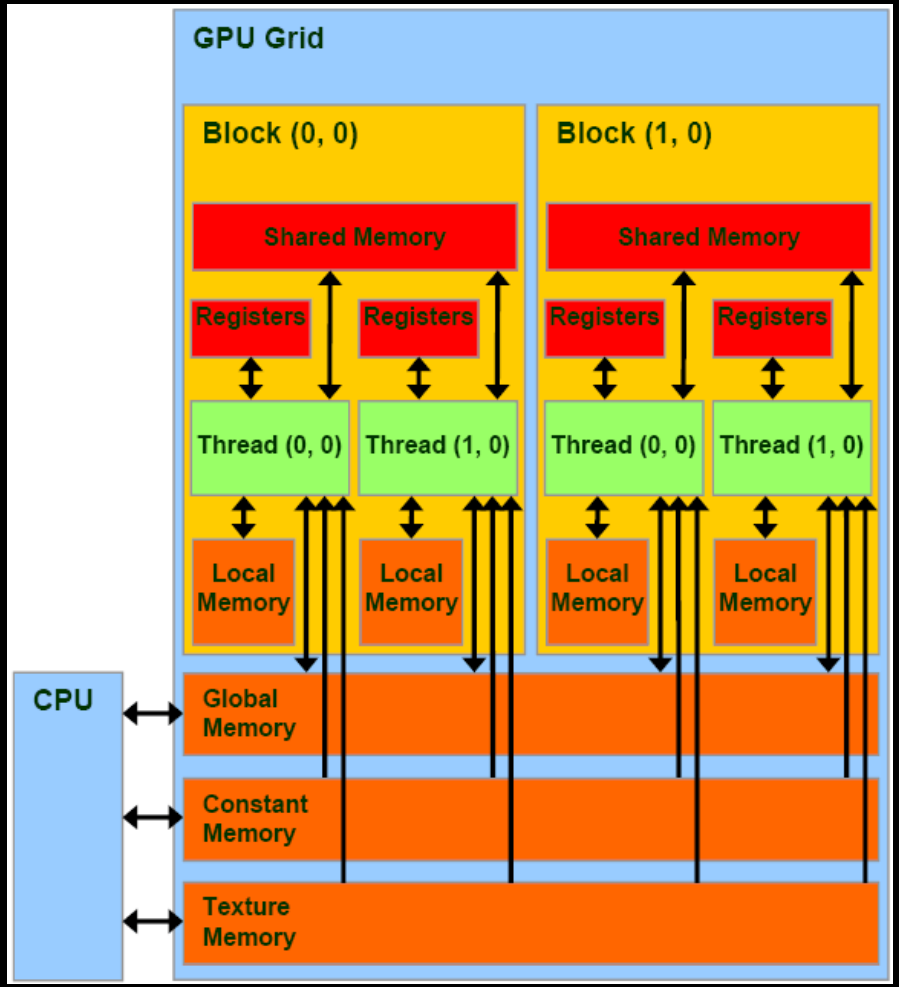
Processing flow





allocate/deallocate memory on the device

- cudaMalloc (void** devPtr, size_t size)
- cudaMallocManaged (void** devPtr, size_t size, unsigned int flags = cudaMemAttachGlobal)
- cudaFree (void* devPtr)



Register
- Fastest
- Thread scope, thread lifetime

Local
- Does not exist
- Abstraction of thread scope

Global memory
- Implemented in DRAM
- High-access latency: 400-800 cycles
- Finite bandwidth: 900GB/sec for V100
- Potential of traffic congestion
- Grid scope, application lifetime

Data copy between host and device

- cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpy Kind kind);
- cudaMemcpyKind
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyDefault

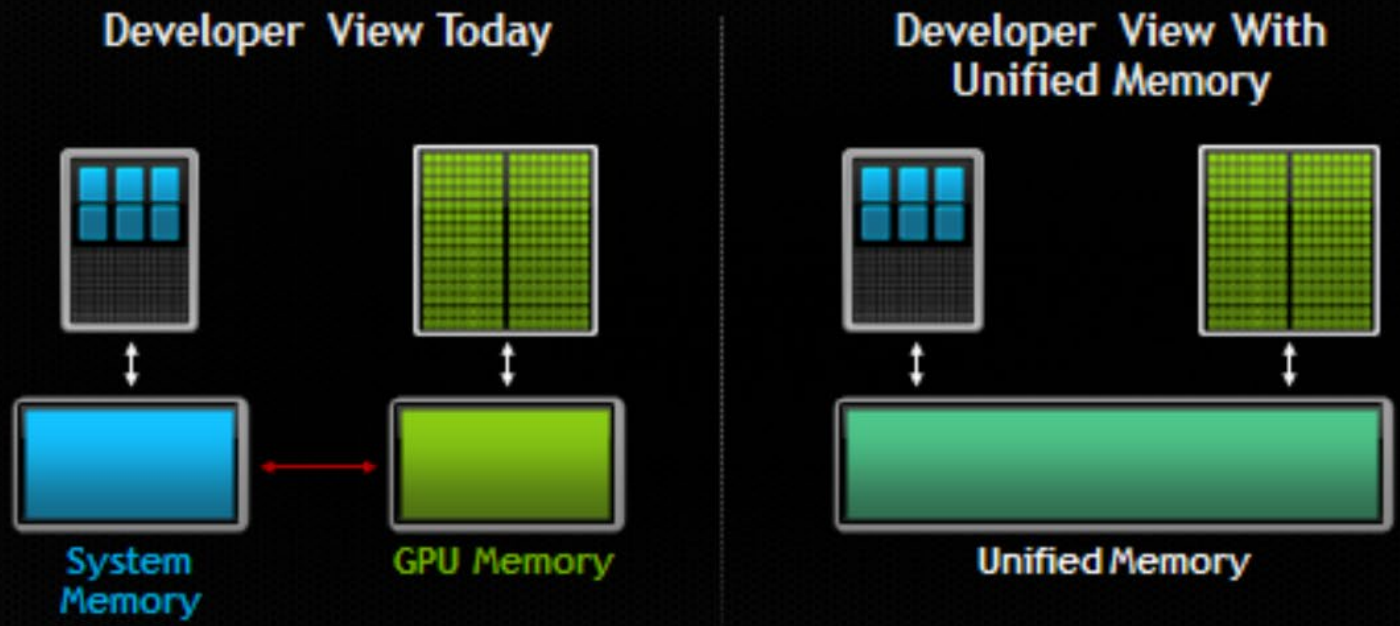
Exercise 3: Vector sum

$$c_i = x_i + y_i$$

nvcc vector_sum.c
nvcc vector_sum_block.cu
nvcc vector_sum_thread.cu
nvcc vector_sum_tb.cu
nvcc vector_sum_block.unified.cu

Unified Memory

Dramatically Lower Developer Effort

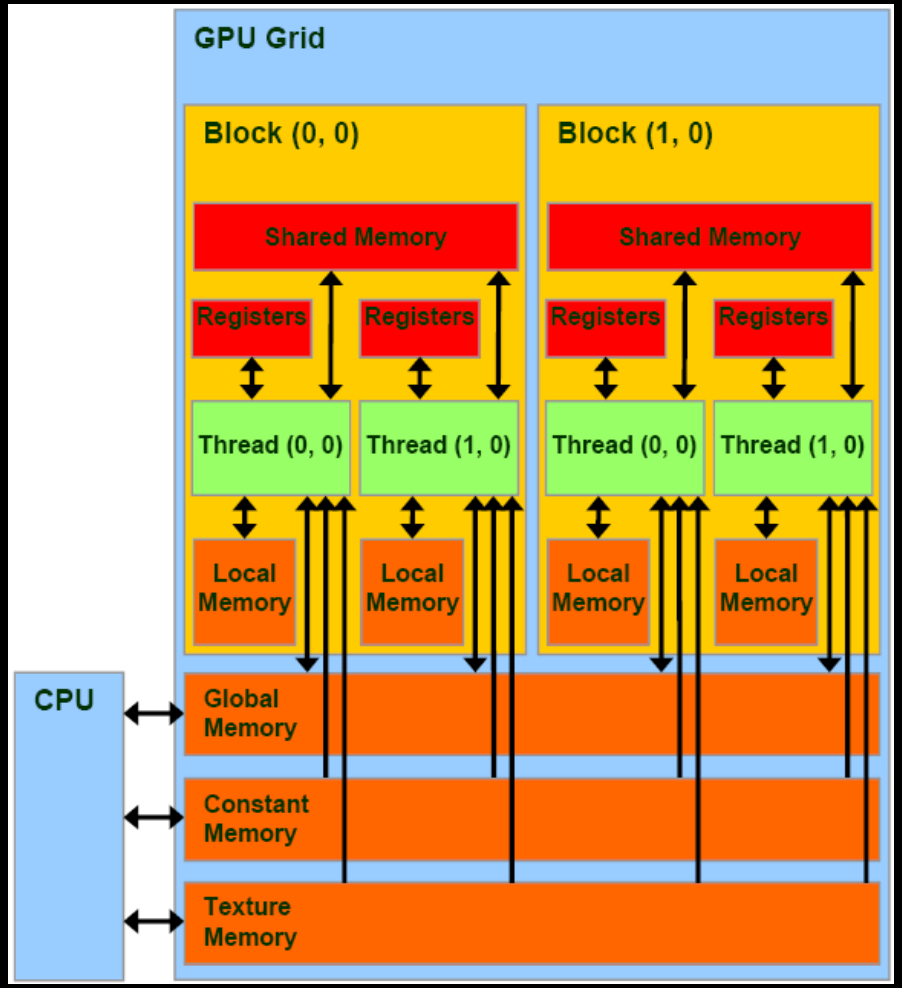


Exercise: Dot-Product

$$c = \sum_{i=0}^{N-1} a_i b_i$$

One example of reduction problems such as SUM, MAX, MIN, …

nvcc dot_product.c
nvcc dot_product.cu
nvcc dot_product_double.cu
nvcc -l cublas dot_product_cublas.cu



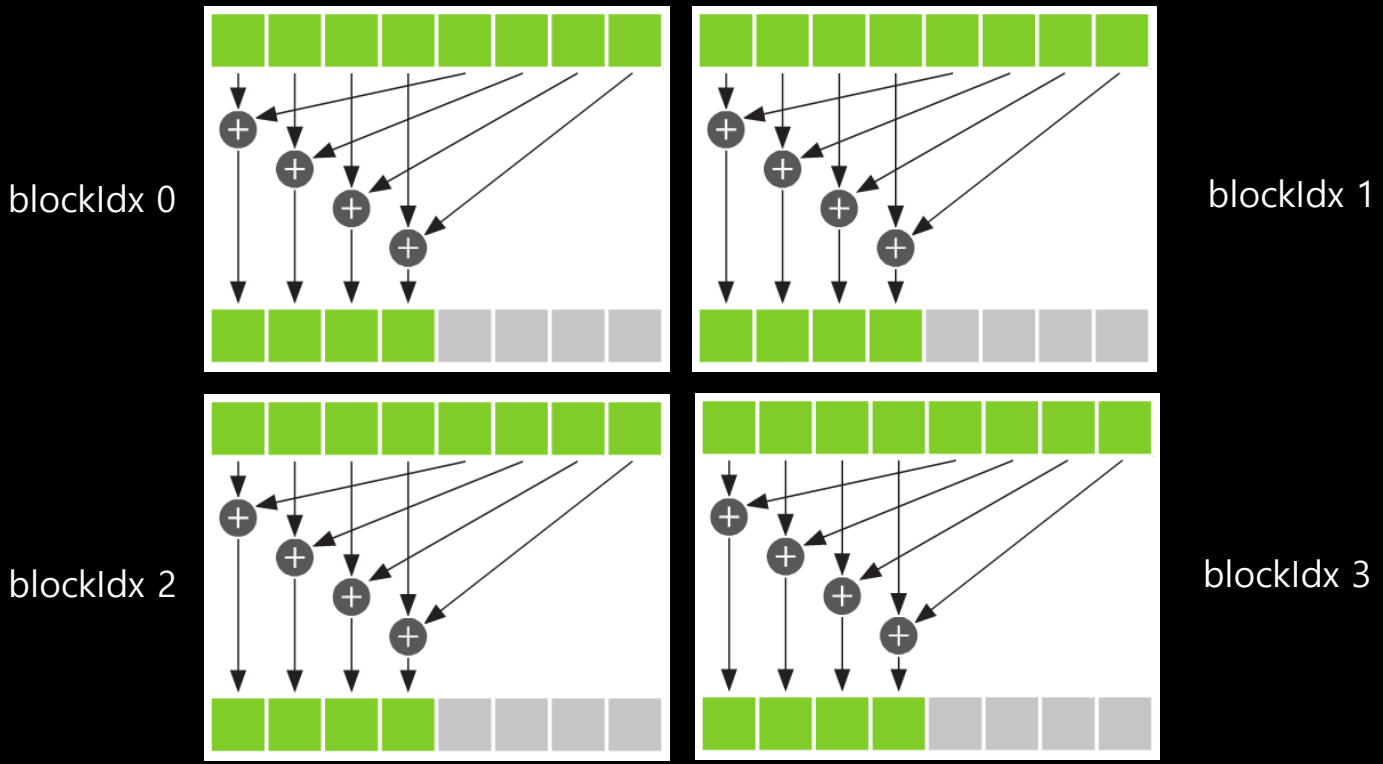
Shared memory

- Extreme fast, highly parallel
- Block scope, kernel lifetime
- 49152 Bytes for V100

L2 cache

- 6291456 Bytes for V100

Reduction



cublasSdot

```
cublasStatus_t cublasSdot (cublasHandle_t handle, int n,
                           const float           *x, int incx,
                           const float           *y, int incy,
                           float           *result)
```

Param.	Memory	In/out	Meaning
handle		input	handle to the cuBLAS library context.
n		input	number of elements in the vectors `x` and `y`.
x	device	input	<type> vector with `n` elements.
incx		input	stride between consecutive elements of `x`.
y	device	input	<type> vector with `n` elements.
incy		input	stride between consecutive elements of `y`.
result	host or device	output	the resulting dot product, which is `0.0` if `n<=0`.

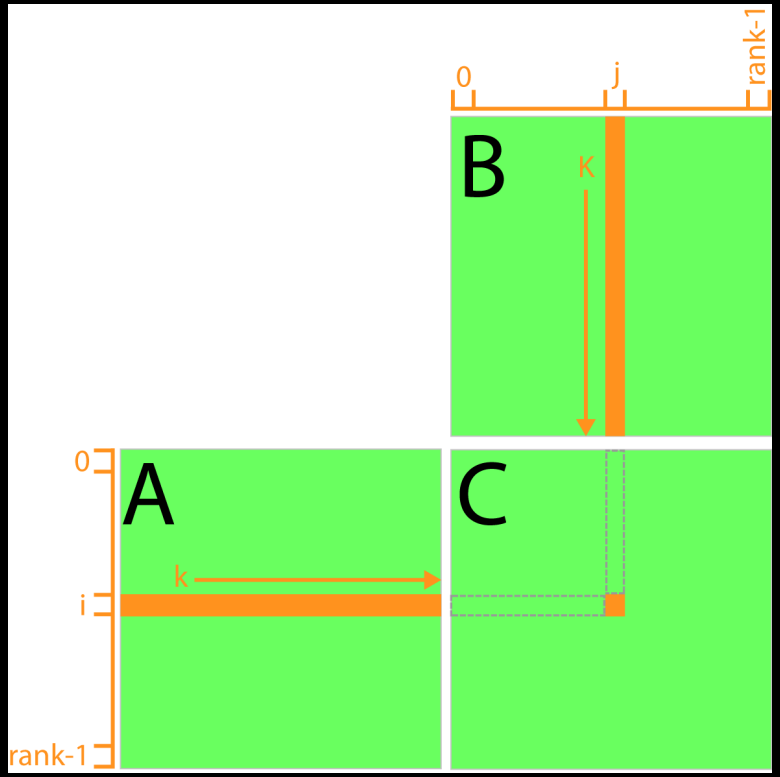
AI (Arithmetic Intensity)

- Definition: number of operations per byte
- AI for a[N]+b[N]

  N ops / 8N bytes = 1/8
- AI for A[N][N] * B[N][N]

  $2N^3$ ops/$8N^2$ bytes = N/4
- If AI ~ 1, performance = AI x memory BW
- If AI >> 1 better performance

M-M Multiplication: Global



Timing using CUDA Events

```
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );

cudaEventRecord( start,0 ); 0 is a default stream id
cudaMemcpy
Kernel
cudaMemcpy
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime( &etime, start, stop ); etime measured in units of millisecond
```

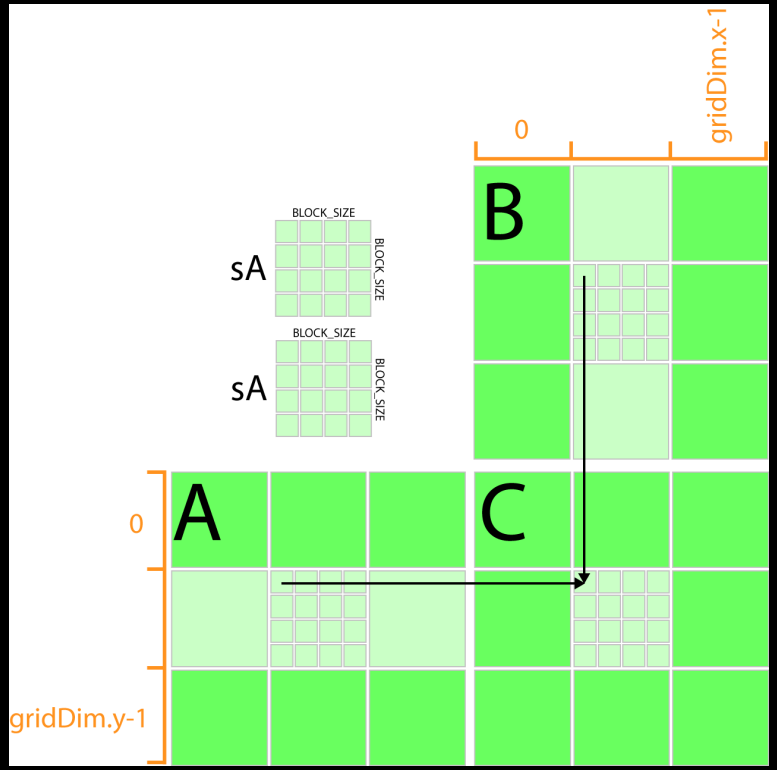
Exercise: Matrix Multiplication

- gcc –std=c99 matmul.c
- nvcc –Xptxas –v matmul_global.cu
- nvcc –Xptxas –v matmul_shared.cu
- nccc –l cublas matmul_cublas.cu

Reducing Global Memory Traffic

- Reducing global memory access enhances performance
- tiling: partition of data into subsets called tiles, such that each tile fits into fast (shared) memory

Matrix-Multiplication: shared



cuBLAS

- cuBLAS: CUDA Basic Linear Algebra Subroutine Library
  - cuBLAS API: the application must allocate the required matrices and vectors in the GPU memory space, fill them with data, call the sequence of desired cuBLAS functions, and then upload the results from the GPU memory space back to the host.
  - cuBLASXt API: the application may have the data on the Host or any of the devices involved in the computation, and the Library will take care of dispatching the operation to, and transferring the data to, one or multiple GPUs present in the system, depending on the user request.
  - cuBLASLt API: a lightweight library dedicated to GEneral Matrix-to matrix Multiply (GEMM) operations with a new flexible API.

https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm

cublasSgeMM

$$C = \alpha \text{op}(A)\text{op}(B) + \beta C$$

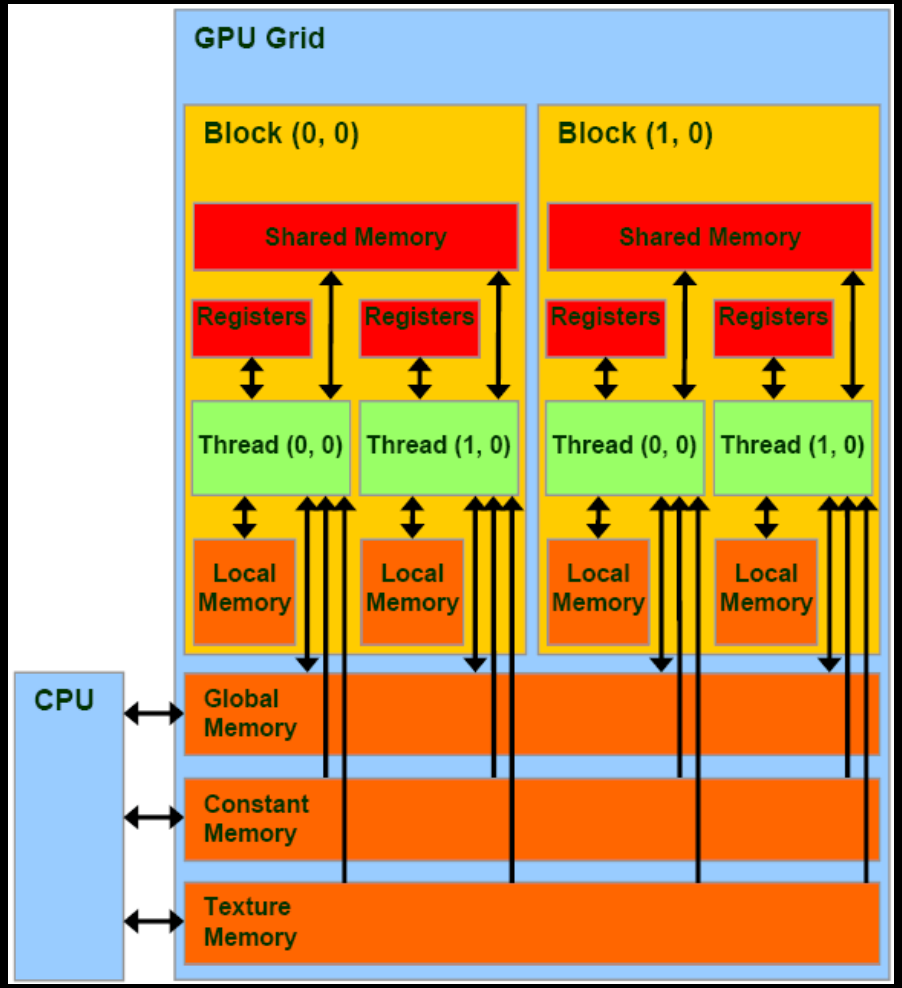
where α and β are scalars, and A , B and C are matrices stored in column-major format with dimensions $\text{op}(A)$ $m \times k$, $\text{op}(B)$ $k \times n$ and C $m \times n$, respectively. Also, for matrix A

$$\text{op}(A) = \begin{cases} A & \text{if transa == CUBLAS\_OP\_N} \\ A^T & \text{if transa == CUBLAS\_OP\_T} \\ A^H & \text{if transa == CUBLAS\_OP\_C} \end{cases}$$

and $\text{op}(B)$ is defined similarly for matrix B .

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

Param.	Memory	In/out	Meaning
handle		input	handle to the cuBLAS library context.
transa		input	operation op(`A`) that is non- or (conj.) transpose.
transb		input	operation op(`B`) that is non- or (conj.) transpose.
m		input	number of rows of matrix op(`A`) and `C` .
n		input	number of columns of matrix op(`B`) and `C` .
k		input	number of columns of op(`A`) and rows of op(`B`).
alpha	host or device	input	<type> scalar used for multiplication

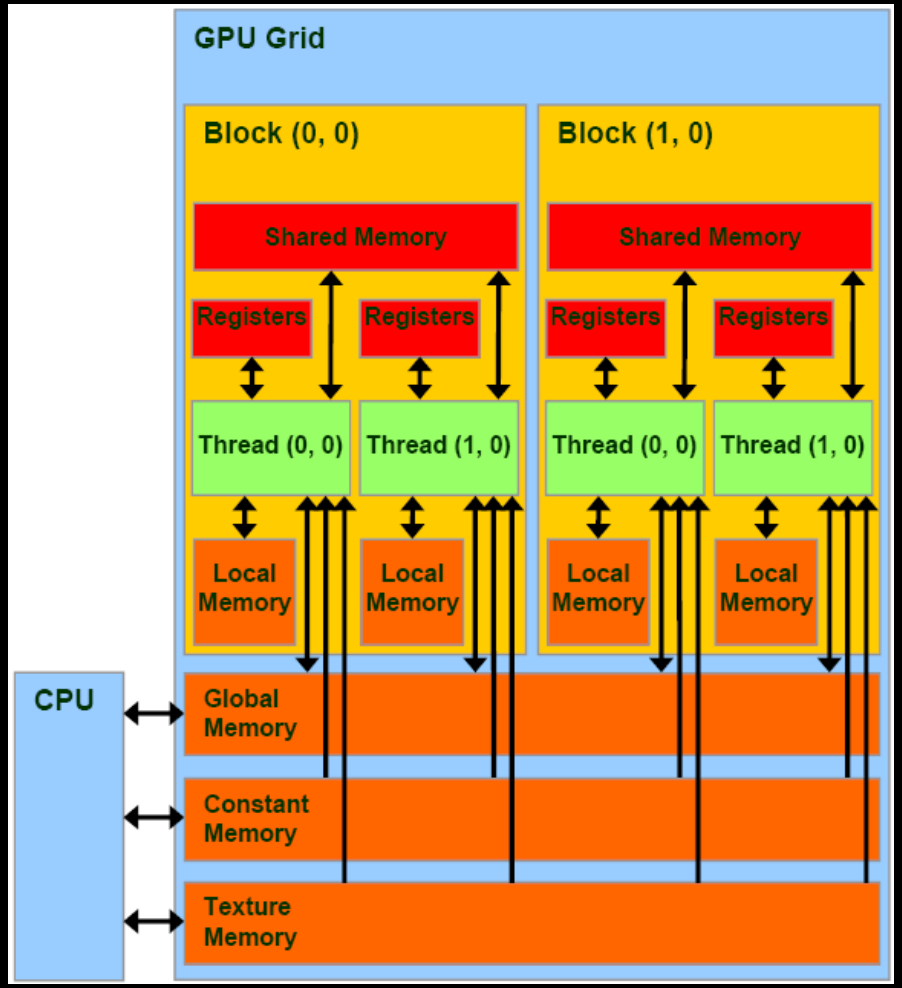


Constant memory

- Space for "constant" data during a kernel execution
- 64kB per a GPU board
- Cached on chip
- Fast if threads of a half warp reads the same address

Exercise 6: constant memory

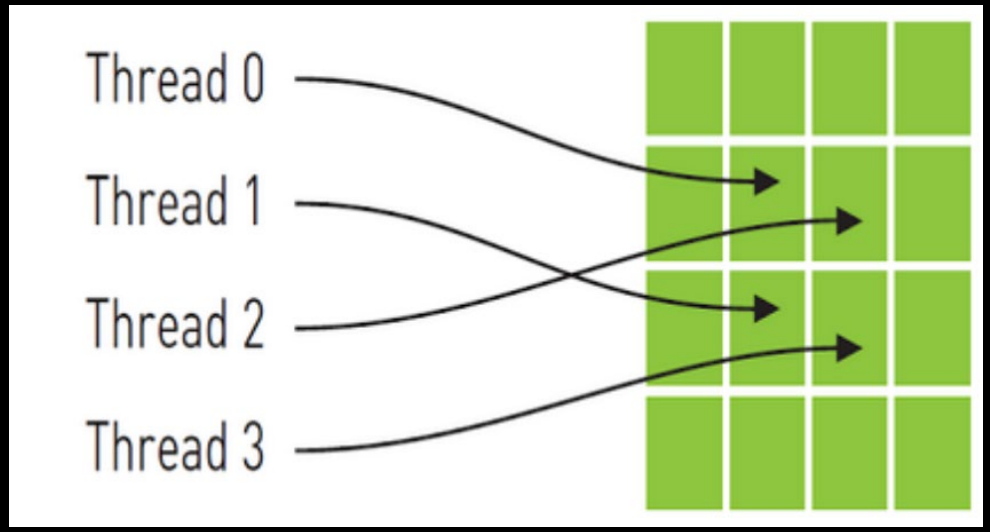
- nvcc const_mem.cu



Texture memory

- cached on chip
- read only
- designed for graphics applications
- Optimized 2D "spatial locality"

Spatial Locality



Heat equation

$$\frac{\partial T}{\partial t} = \kappa\left(\frac{\partial}{\partial x^2} + \frac{\partial}{\partial y^2}\right)T$$

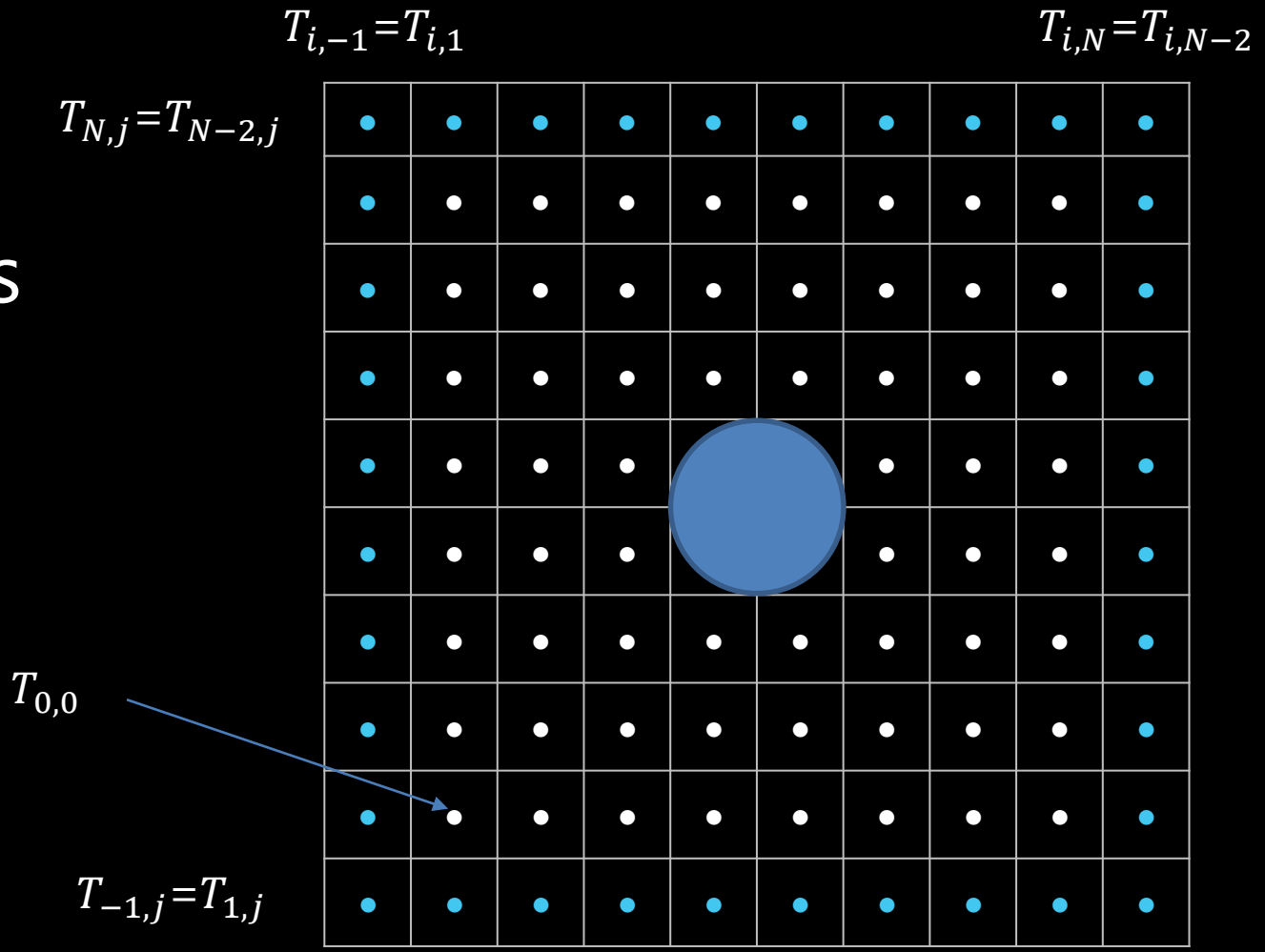
$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = \kappa\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right)$$

$$T_{i,j}^{n+1} = \left(1 - 4\frac{\kappa\Delta t}{h^2}\right)T_{i,j}^{n} + \frac{\kappa\Delta t}{h^2}(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n})$$

where $h = \Delta x = \Delta y$

$$T_{i,j}^{n+1} = \frac{1}{4}(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n}) \text{ when } \Delta t = \frac{h^2}{4\kappa}$$

Initial and Boundary Conditions



Exercise 7: heat equation

- nvcc heat.c
- nvcc heat_global.cu
- nvcc heat_texture_1d.cu
- nvcc heat_texture_2d.cu

https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

CUDA Streams

- A stream is a sequence of operations that execute on the device in the order in which they are issued by the host code.
- Operations in different streams can be interleaved and, they can even run concurrently.
- Default stream: When no stream is specified, the default stream is used.

Default stream I

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- From the device point of view
  - All three operations will be executed in the order that they were issued.
- From the host point of view
  - First cudaMemcpy is synchronous transfer.
  - Increment kernel is asynchronous.
  - Second cudaMemcpy cannot begin due to the device-side order of execution

Default stream II

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
myCpuFunction(b)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- As soon as the increment() kernel is launched on the device, the CPU executes myCpuFuction with the kernel execution on the GPU.

```
increment<<<1,N,0,stream1>>>(d_a)
```

Non-default streams

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1)
result = cudaStreamDestroy(stream1)
```

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
```

```
increment<<<1,N,0,stream1>>>(d_a)
```

example

- streams.cu
  - overlapping computation and data copy
- nvprof ./a.out
- nvvp ./a.out

Continuous vs discrete Fourier Transform

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-j2\pi ft} dt$$

$$H = \sum_{k=0}^{N-1} h(k) e^{-j2\pi nk/N} \qquad n = 0,1, \ldots N-1$$

Fourier Transform of a rectangular pulse

$$h(t) = A \quad |t| < T_0$$
$$= 0 \quad |t| > T_0$$

$$H(f) = \int_{-T_0}^{T_0} Ae^{-j2\pi ft} dt$$

$$= A\int_{-T_0}^{T_0} \cos(2\pi ft)\, dt - jA\int_{-T_0}^{T_0} \sin(2\pi ft)\, dt = \frac{A}{2\pi f}\sin(2\pi ft)\,|_{-T_0}^{T_0}$$

$$= 2AT_0\,\frac{\sin(2\pi T_0 f)}{2\pi T_0 f}$$

cuFFT, CUDA Fast Fourier Transform

- Highly optimized for input sizes in the form $2^a \times 3^b \times 5^c \times 7^d$. Powers of two are fastest.
- An O(nlogn) algorithm for every input data size
- Half-precision (16-bit floating point), single-precision (32-bit floating point) and double-precision (64-bit floating point). Transforms of lower precision have higher performance.
- Types supported are:
  - C2C - Complex input to complex output
  - R2C - Real input to complex output
  - C2R - Symmetric complex input to real output
- 1D, 2D and 3D transforms
- Execution of multiple 1D, 2D and 3D transforms simultaneously.
- In-place and out-of-place transforms
- Arbitrary intra- and inter-dimension element strides (strided layout)
- FFTW compatible data layout
- Execution of transforms across multiple GPUs
- Streamed execution, enabling asynchronous computation and data movement

https://docs.nvidia.com/cuda/cufft/index.html#cufft-api-reference

cufftPlan1d

```
cufftResult
    cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int
batch);
```

Input

`plan`	Pointer to a `cufftHandle` object
`nx`	The transform size (e.g. 256 for a 256-point FFT)
`type`	The transform data type (e.g., `CUFFT_C2C` for single precision complex to complex)
`batch`	Number of transforms of size `nx`. Please consider using `cufftPlanMany` for multiple transforms.

Output

`plan`	Contains a cuFFT 1D plan handle value

https://docs.nvidia.com/cuda/cufft/index.html#plan-execution

CUFFTExecC2C

```
cufftExecC2C(cufftHandle plan, cufftComplex *idata,
    cufftComplex *odata, int direction);
```

Input

`plan`	`cufftHandle` returned by `cufftCreate`
`idata`	Pointer to the complex input data (in GPU memory) to transform
`odata`	Pointer to the complex output data (in GPU memory)
`direction`	The transform direction: `CUFFT_FORWARD` or `CUFFT_INVERSE`

Output

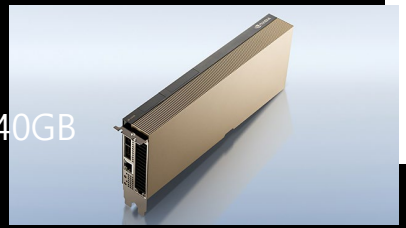
`odata`	Contains the complex Fourier coefficients

fftw, cufft

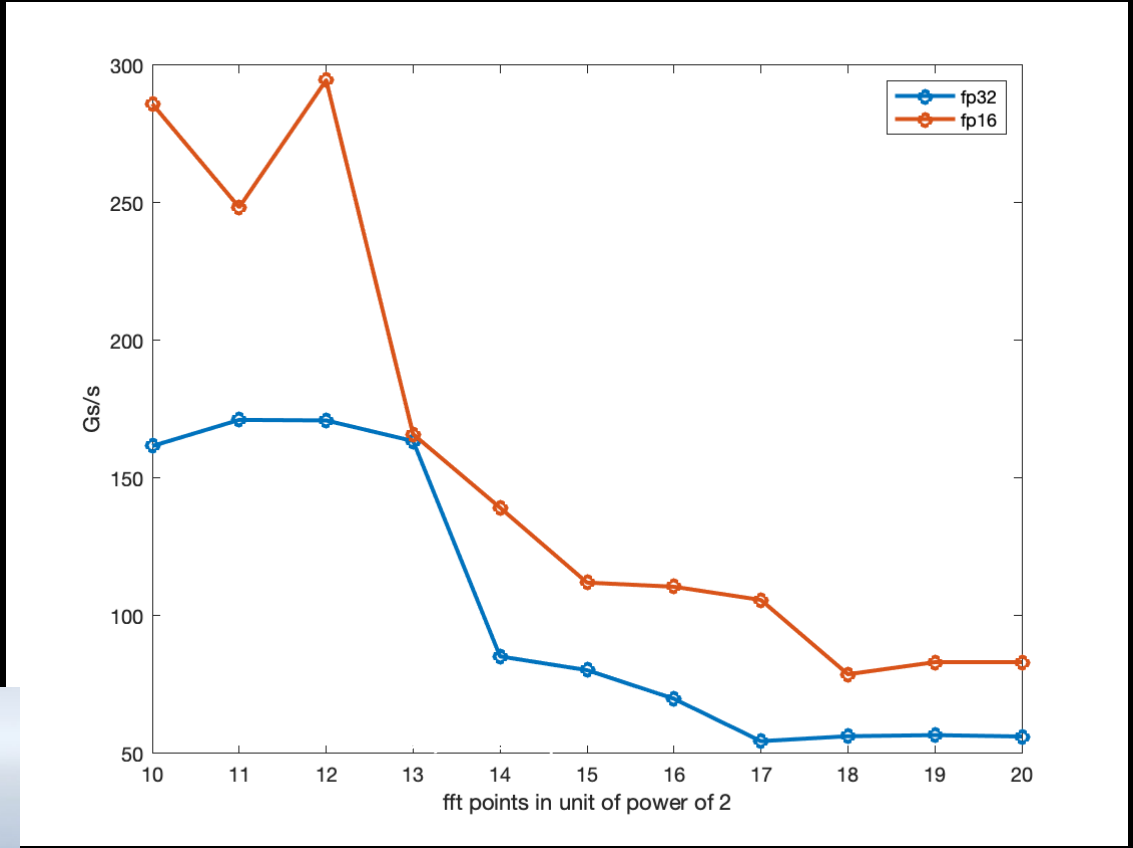
- gcc -std=c99 -I/usr/include -L/usr/lib64 -lfftw3f fftw_c2c_rect.c
- nvcc -l cufft cufft_c2c_rect.cu
- gnuplot
- gnuplot> load “cufft_c2c_rect.plt”

FFT performance of NVIDIA A100-PCIE-40GB

- The single-precision (half-precision) FFT performance of NVIDIA A100 is 80 (112) Gsamples/s with 32768 FFT points.
- Assuming the 80 Gs/s FFT performance of one GPU, we need about 100 GPUs to process FFTs of 8.2Ts/s.



NVIDIA A100-PCIE-40GB



1024 32768 1048576