NVIDIA GPU コンピューティング 入門

Senior Healthcare Solution Architect | Yong He (LEON) | 2025

Agenda

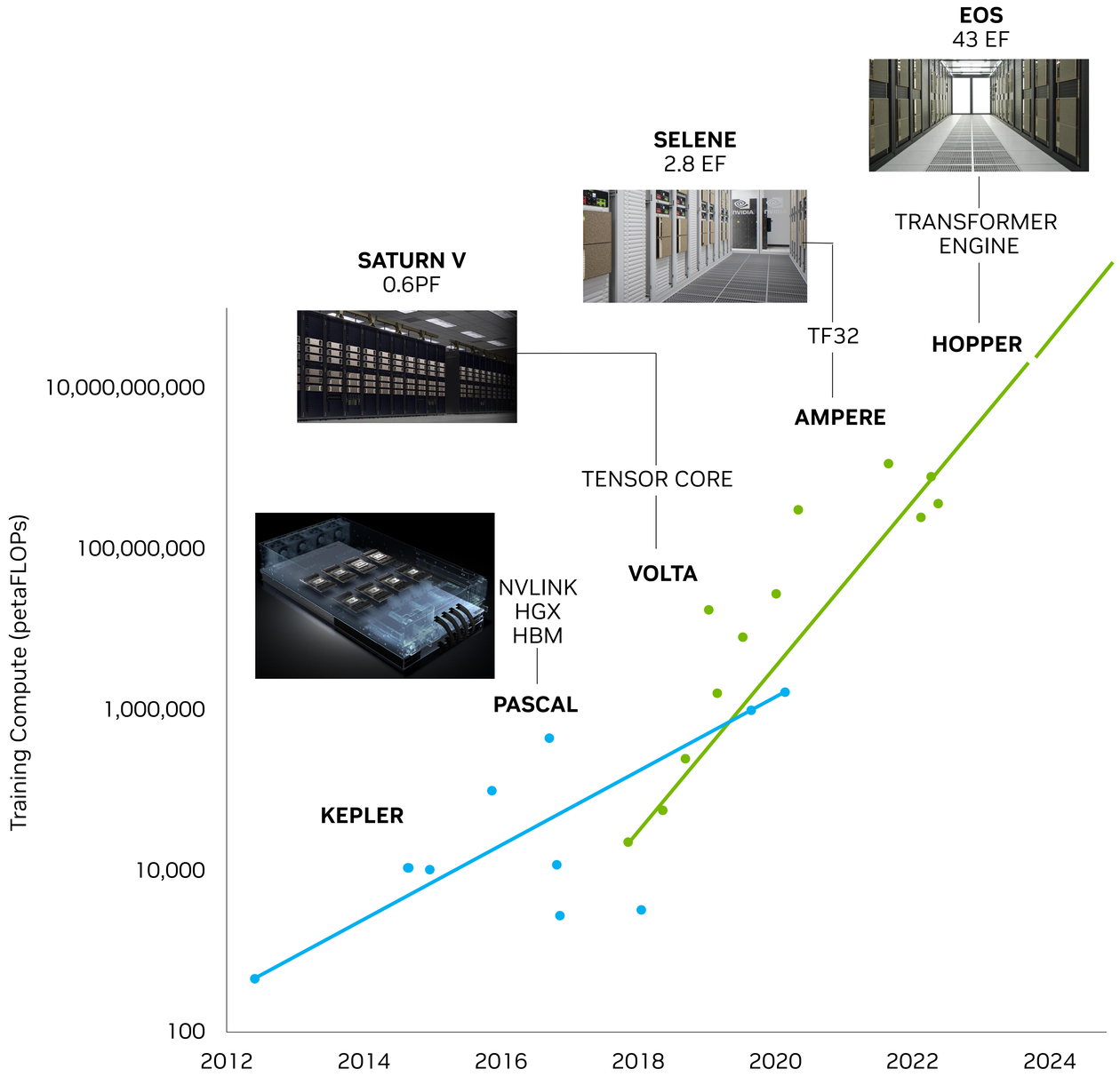
- NVIDIA GPU について
- NVIDIA GPUコンピューティング とは
- CUDA
- CUDA ライブラリ
- まとめ
-

NVIDIA GPU について

世界をリードするNVIDIAのAIプラットフォーム

データセンターからエッジまで同一のアーキテクチャ — GPU コンピューティング

爆発的なAI演算の計算量の伸びと、NVIDIA GPUアーキテクチャの進化



NVIDIA GPU の構造

NVIDIA H100 (GH100)

- PCI I/F
  - ホスト接続インタフェース
- Giga Thread Engine
  - SM に処理を割り振るスケジューラ
- DRAM I/F (HBM3, HBM2e)
  - 全 SM、PCI I/F からアクセス可能なメモリ（デバイスメモリ, フレームバッファ）
- L2 cache (50 MB)
  - 全 SM からアクセス可能な R/W キャッシュ
- GPC (Graphics Processing Cluster)
  - 高レベルなハードウェア ブロック、主要なプロセッシングユニットはすべてGPC内部に存在
- SM (Streaming Multiprocessor)
  - GPC 内部に存在、主要なプロセッシングユニットの集合体

SM (Streaming Multi-processor)

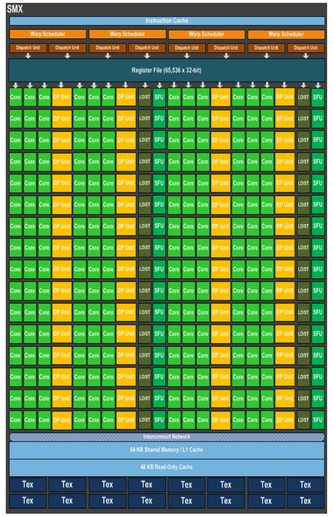
NVIDIA GPU の構造 (GH100)

- 汎用演算ユニット
  - INT32
  - FP32
  - FP64
  - 行列計算ユニット (TensorCore)
- Other units
  - LD/ST, SFU, etc
- レジスタ (32 bit): 64K 個
- 共有メモリ/L1 キャッシュ: 256 KB

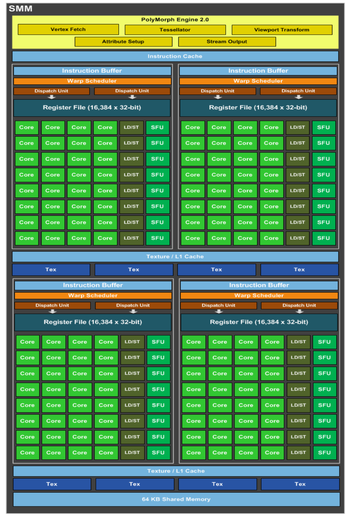
- Tensor Memory Accelerator

Compute Capability

https://developer.nvidia.com/cuda-gpus



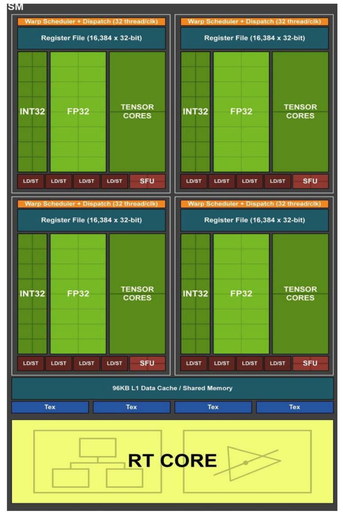
Kepler CC3.5



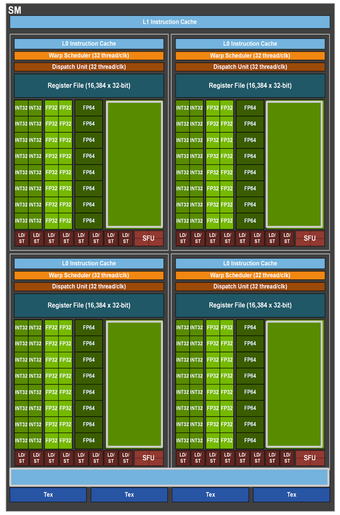
Maxwell CC 5.0

Pascal CC 6.0

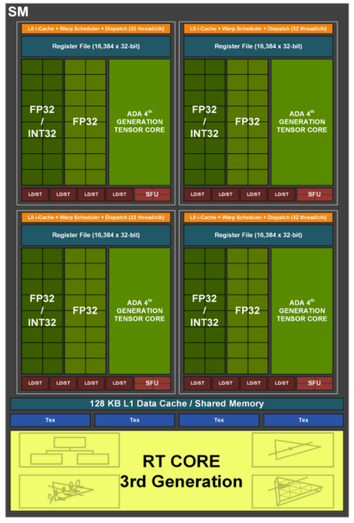
Volta CC 7.5



Turing CC 7.5



Ampere CC 8.0



Ada Lovelace CC 8.9

Hopper CC 9.0

Compute Capability による違い

A100 vs H100

- Compute Capability が異なるGPU で同一コードは動作する (see: cuda-compatibility documentation)
- HWアーキテクチャは性能向上の為に進化しており、HWの世代を新しくするだけでCUDAコードを変更する事なく性能向上させる事ができる

A100 (CC:8.0)

H100 (cc:9.0)

参考: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions

Compute Capability による違い

A100 vs H100

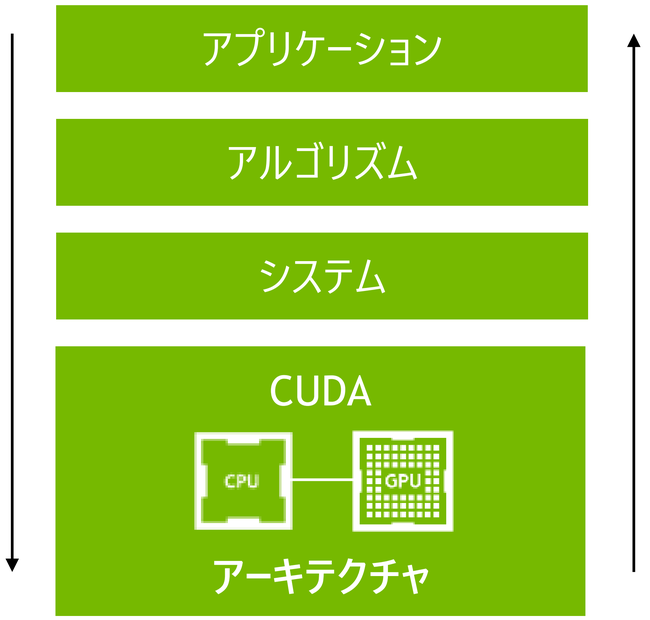
	A100 80GB SXM	H100 SXM
SMs	108	132
FP32 Core / SM	64	128
FP64 Core / SM (TensorCoreを除く)	32	64
Total Tensor Cores in GPU	432	528
FP64	9.7 TFLOPS	34 teraFLOPS
FP64 Tensor Core	19.5 TFLOPS	67 teraFLOPS
FP32	19.5 TFLOPS	67 teraFLOPS
TF32 Tensor Core	312 TFLOPS*	989 teraFLOPS
BF16 Tensor Core	624 TFLOPS*	1,979 teraFLOPS*
FP16 Tensor Core	624 TFLOPS*	1,979 teraFLOPS*
FP8 Tensor Core	1248 TOPS*	3,958 teraFLOPS2

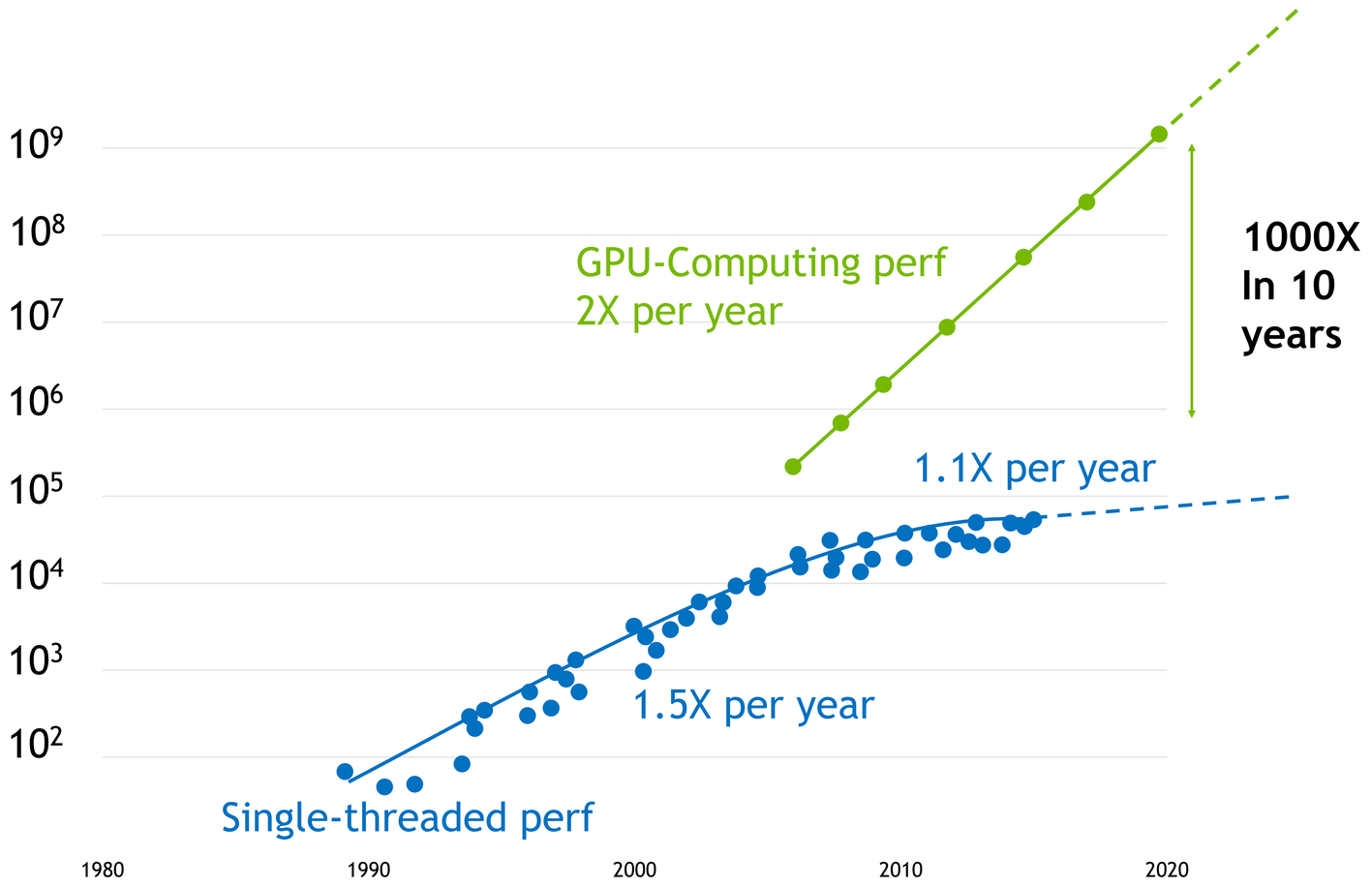
* With sparsity.

https://www.nvidia.com/en-us/data-center/h100/ および
https://www.nvidia.com/en-us/data-center/a100/ より数値を引用

GPU コンピューティングとは

NVIDIA GPU コンピューティングの台頭



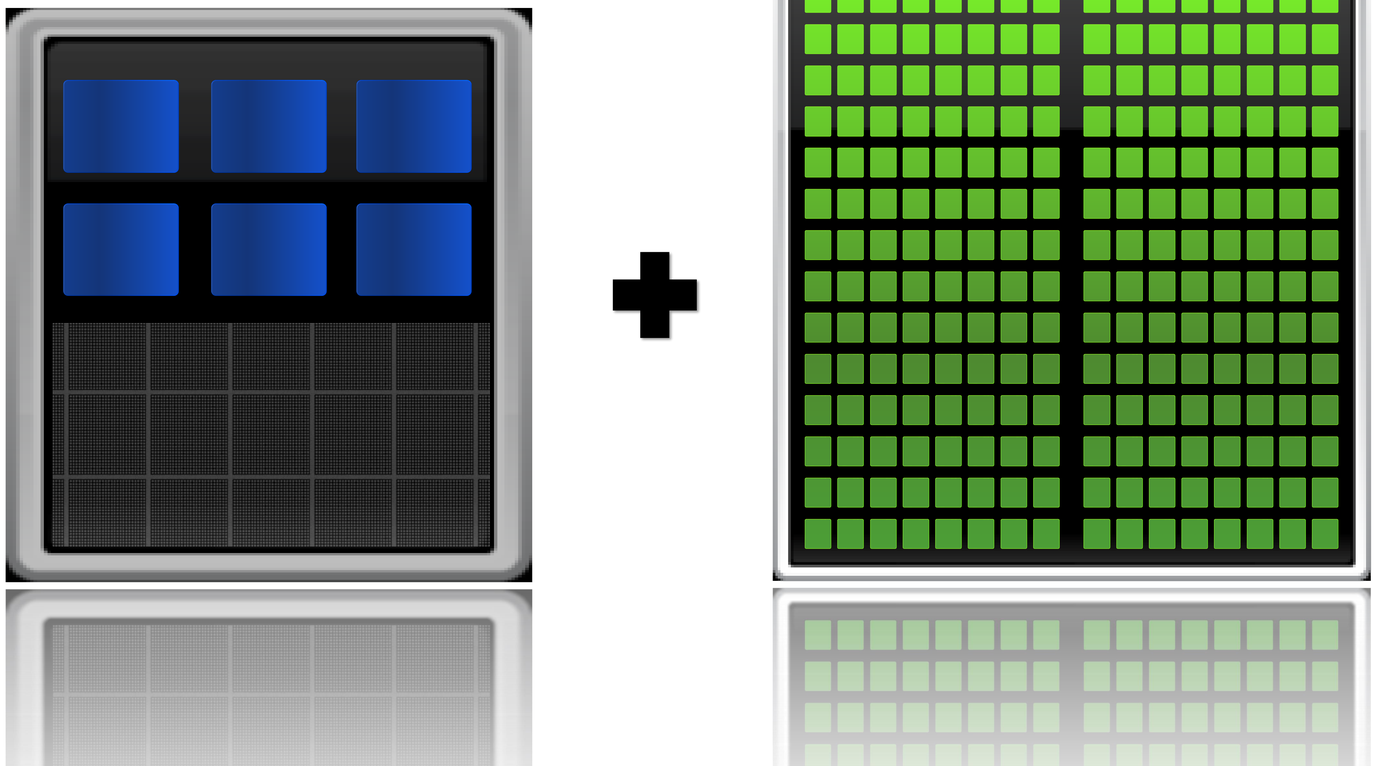


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

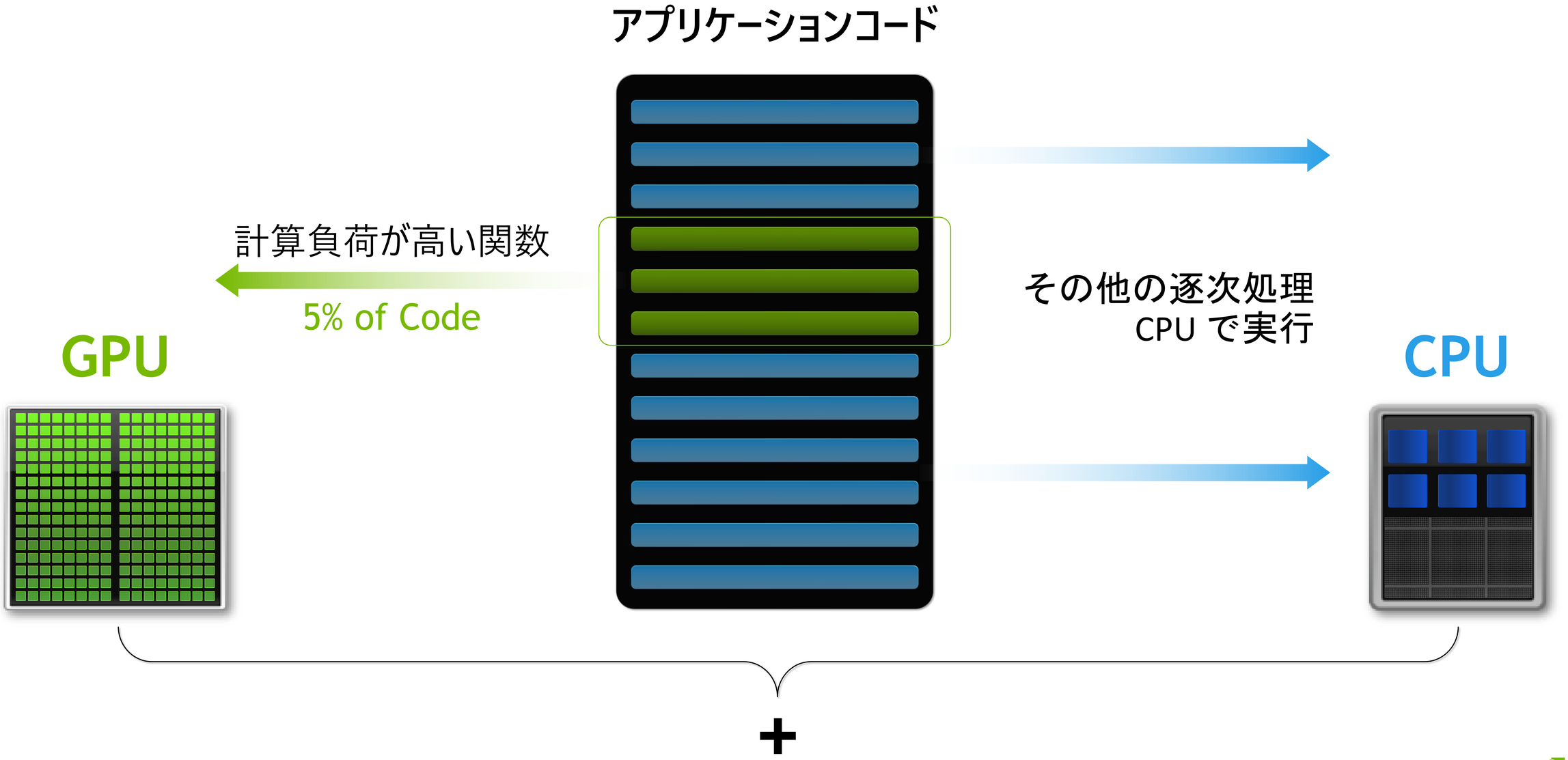
GPU コンピューティングとは？

CPU
逐次処理に最適化

GPU Accelerator
並列処理に最適化



GPU コンピューティングとは？

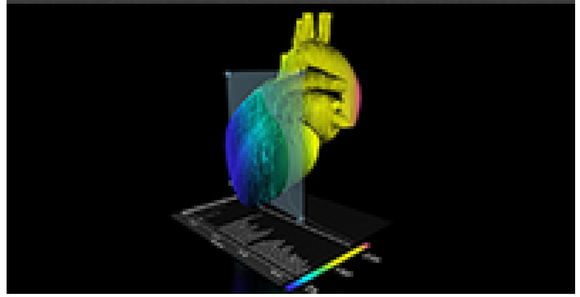


GPU アプリケーション

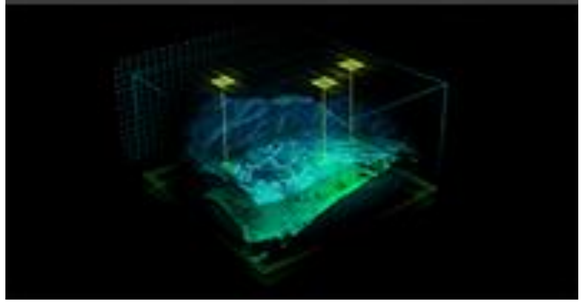
http://www.nvidia.com/object/gpu-applications.html

画像処理 機械学習

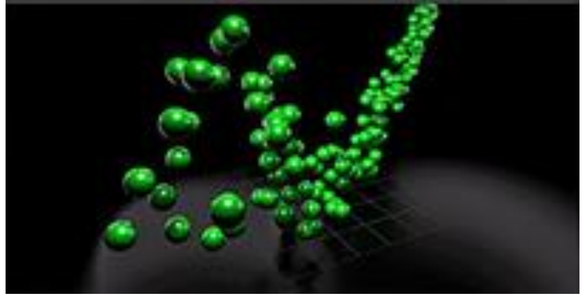
医療画像



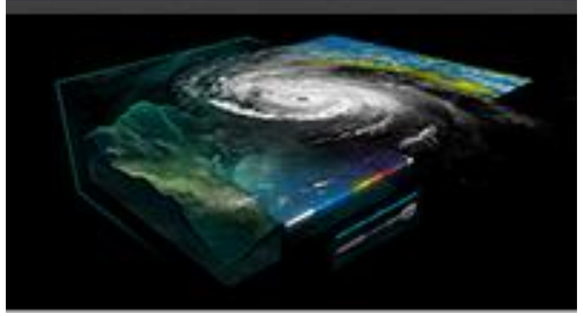
防衛



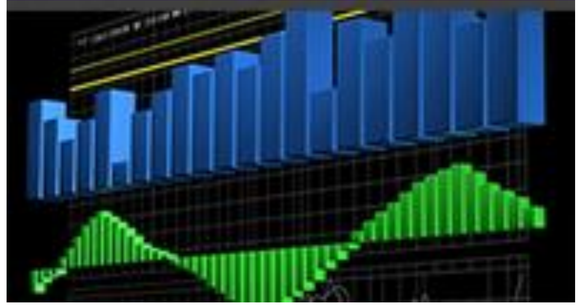
計算化学



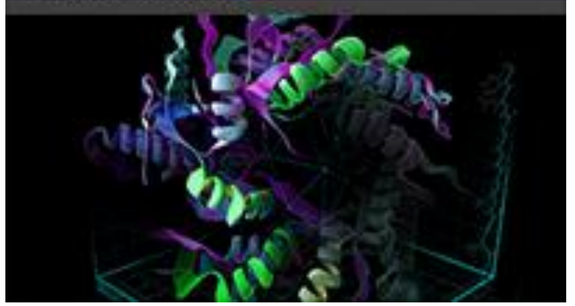
気象



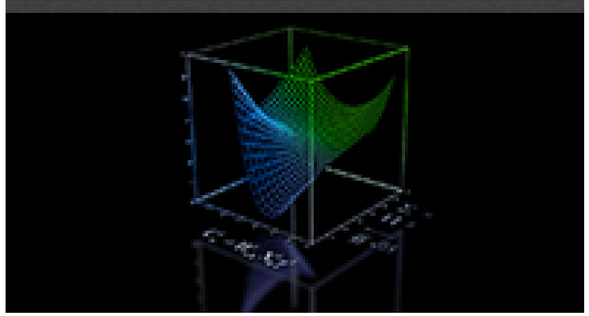
金融工学



バイオ

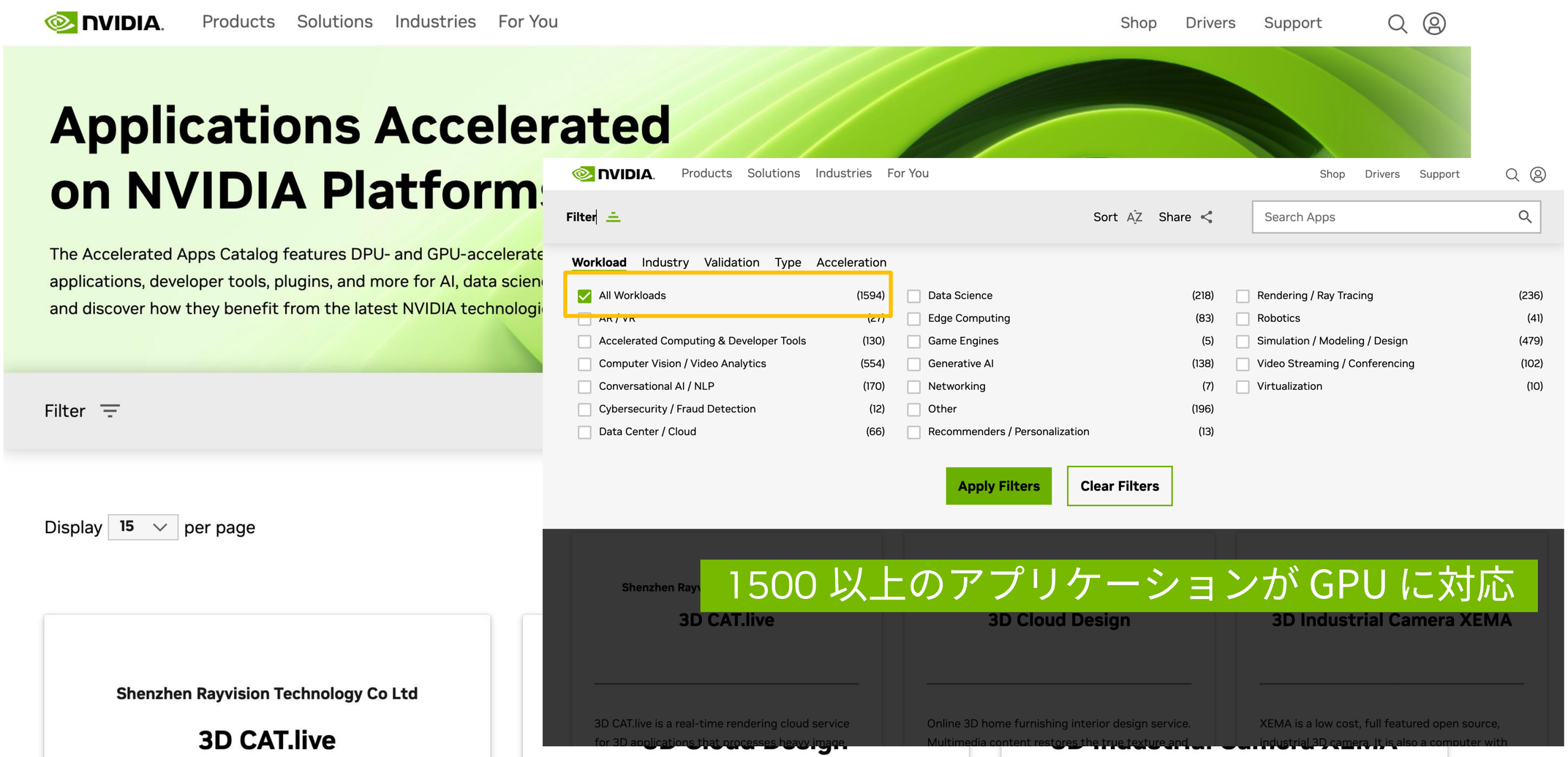


数値解析



GPU アプリケーション

https://www.nvidia.com/en-us/gpu-accelerated-applications/



NVIDIA エンタープライズ ソフトウェア プラットフォーム

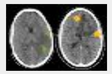
ユースケース


Speech


Translate


Recommender

コンシューマインターネット


Healthcare

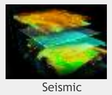

Manufacturing


Finance

インダストリアルアプリケーション


Molecular Simulations


Weather Forecasting


Seismic Mapping

スーパーコンピュータ

アプリケーション & フレームワーク

TensorFlow mxnet ONNX RAPIDS PYTORCH Chainer

Amber NAMD +600 Applications

CATIA Ps AUTODESK 3DS MAX

CUDA-X ライブラリ

機械学習: cuDF | cuML | cuGRAPH

深層学習 / HPC: cuDNN | CUTLASS | TENSORRT | CUDA Math Libraries

プログラム言語: python OpenACC | C++ | LLVM Compiler For CUDA

CUDA

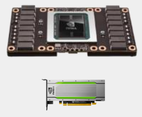
CUDA ツールキット: CUDA COMPILER | DEVELOPER TOOLS (DEBUGGERS | PROFILERS) | CUDA C++ CORE

CUDA ドライバ: MEMORY MANAGEMENT | WINDOWS & GRAPHICS | COMMS LIBRARIES

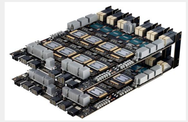
OS プラットフォーム

CentOS SUSE Windows Server

データセンタ GPUs & システム


データセンタ GPU


NVIDIA DGX FAMILY


NVIDIA HGX

DELL Hewlett Packard Enterprise IBM inspur
SYSTEM OEM

aws Google Cloud Platform Microsoft Azure
クラウド

DEPLOYMENT

コンテナ & オーケーストレーション

docker cri-o slurm workload manager

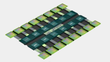
PACKAGING

rpm

GPU 管理とモニタリング

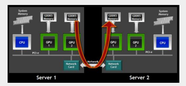

DCGM


NVML/ nvidia-smi


Fabric Manager

OOB

I/O AND STORAGE

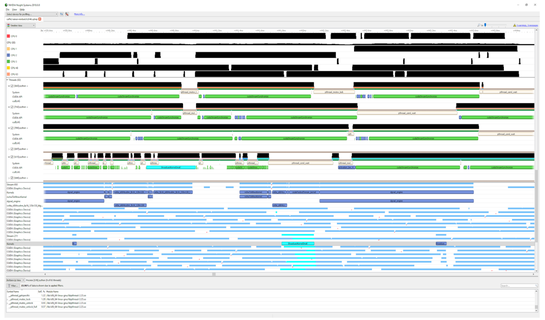

GPUDIRECT

VOLUMES

NETWORKING

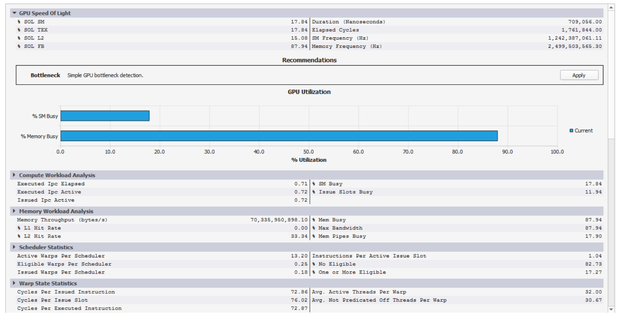
NVIDIA

CUDA デベロッパーツール



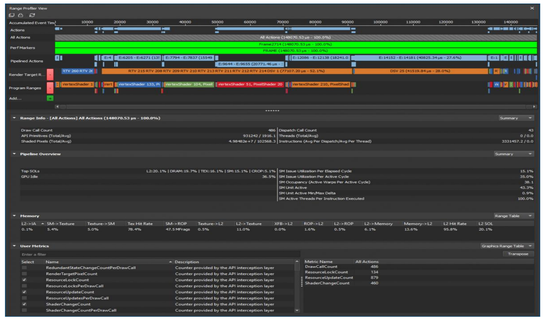
Nsight Systems

System-wide application algorithm tuning



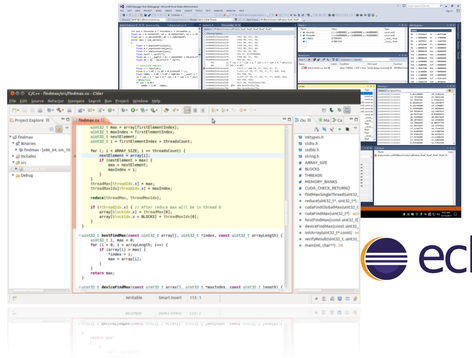
Nsight Compute

CUDA Kernel Profiling and Debugging



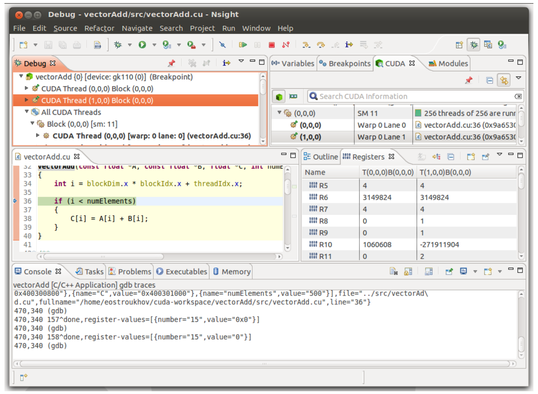
Nsight Graphics

Graphics Shader Profiling and Debugging



IDE Plugins

Nsight Eclipse Edition/Visual Studio (Editor, Debugger)



cuda-gdb

CUDA Kernel Debugging

```
//Out-of-bounds Array Access

__global__ void oobAccess(int* in, int* out)
{
    int bid = blockIdx.x;
    int tid = threadIdx.x;

    if (bid == 4)
    {
        out[tid] = in[dMem[tid]];
    }
}

int main()
{
    ...
    // Array of 8 elements, where element 4 causes the OOB
    std::array<int, Size> hMem = {0, 1, 2, 10, 4, 5, 6, 7};
    cudaMemcpy(d_mem, hMem.data(), size, cudaMemcpyHostToDevice);

    oobAccess<<<10, Size>>>(d_in, d_out);
    cudaDeviceSynchronize();
    ...

$ /usr/local/cuda-11.0/Sanitizer/compute-sanitizer --destroy-on-device-error kernel --show-backtrace no
basic
========= COMPUTE-SANITIZER
Device: Tesla T4
========= Invalid __global__ read of size 4 bytes
=========     at 0x480 in
/tmp/CUDA11.0/ComputeSanitizer/Tests/Memcheck/basic/basic.cu:40:oobAccess(int*,int*)
=========     by thread (3,0,0) in block (4,0,0)
=========     Address 0x7f551f200028 is out of bounds
```

Compute Sanitizer

Memory, Race Checking

NVIDIA プラットフォーム上でのプログラミング

GPU アクセラレーションの実現方法

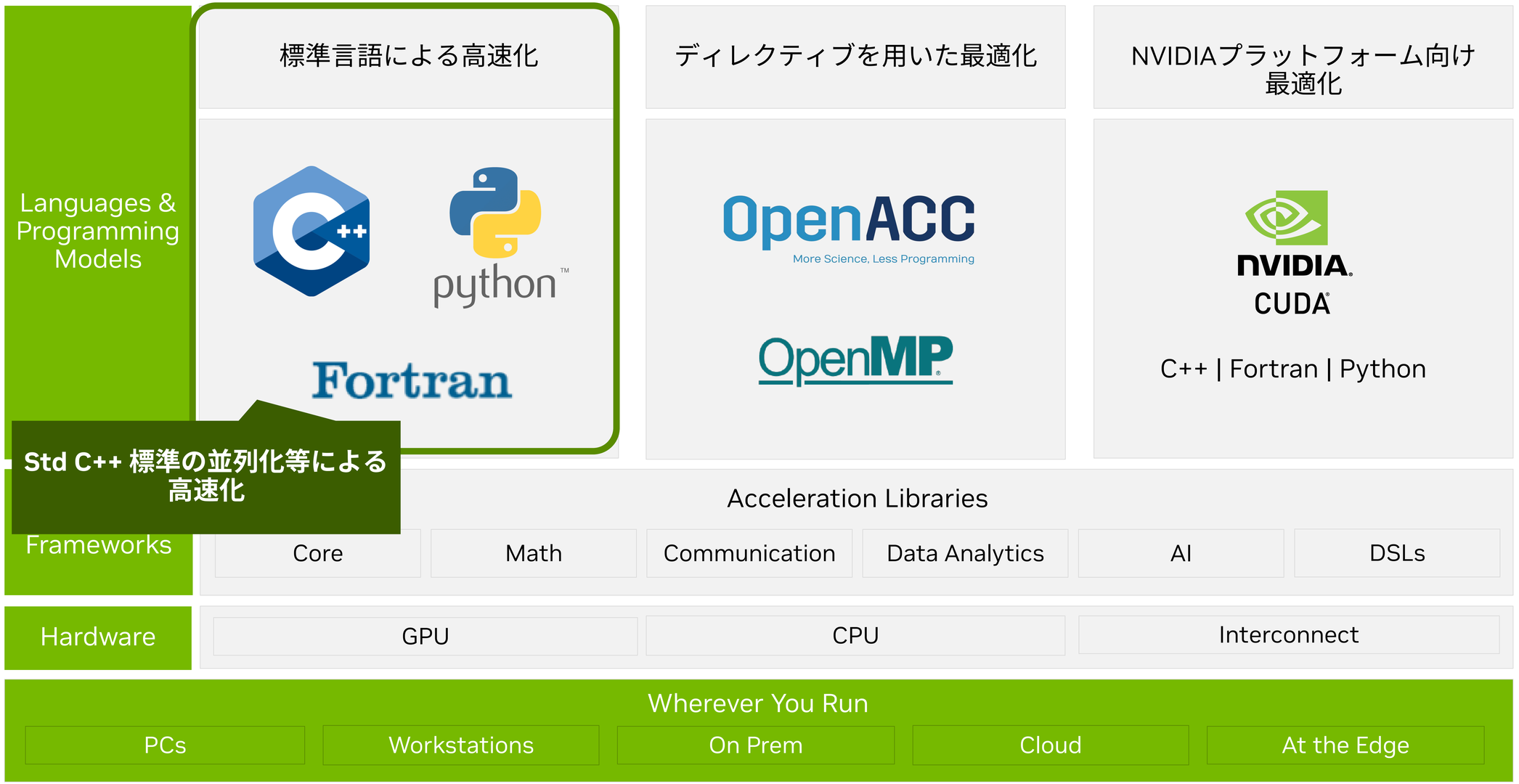
	標準言語による高速化	ディレクティブを用いた最適化	NVIDIAプラットフォーム向け最適化
Languages & Programming Models	  Fortran	OpenACC More Science, Less Programming OpenMP	NVIDIA CUDA C++ | Fortran | Python

Libraries & Frameworks	Acceleration Libraries					
	Core	Math	Communication	Data Analytics	AI	DSLs
Hardware	GPU		CPU		Interconnect	

Wherever You Run				
PCs	Workstations	On Prem	Cloud	At the Edge

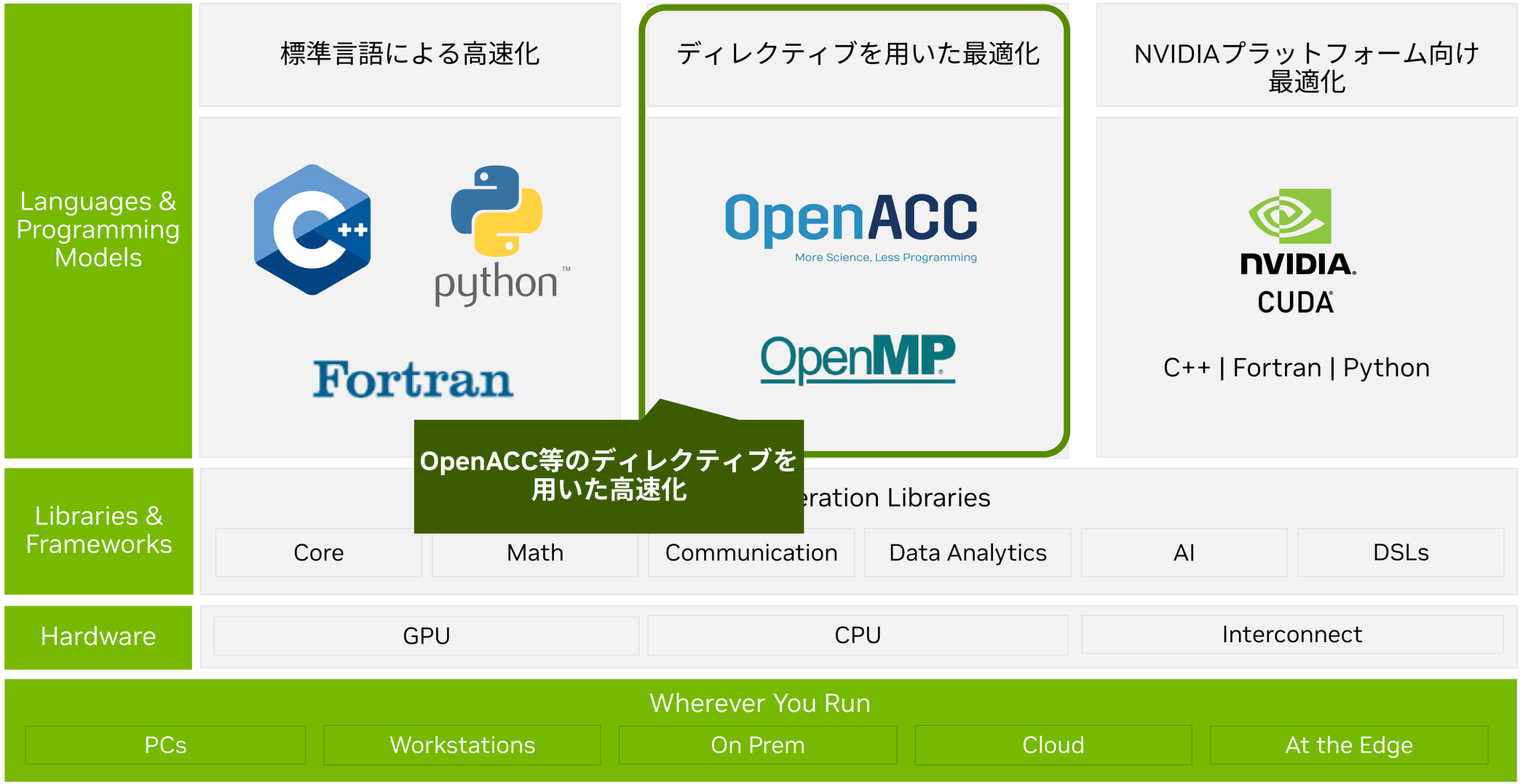
NVIDIA プラットフォーム上でのプログラミング

GPU アクセラレーションの実現方法



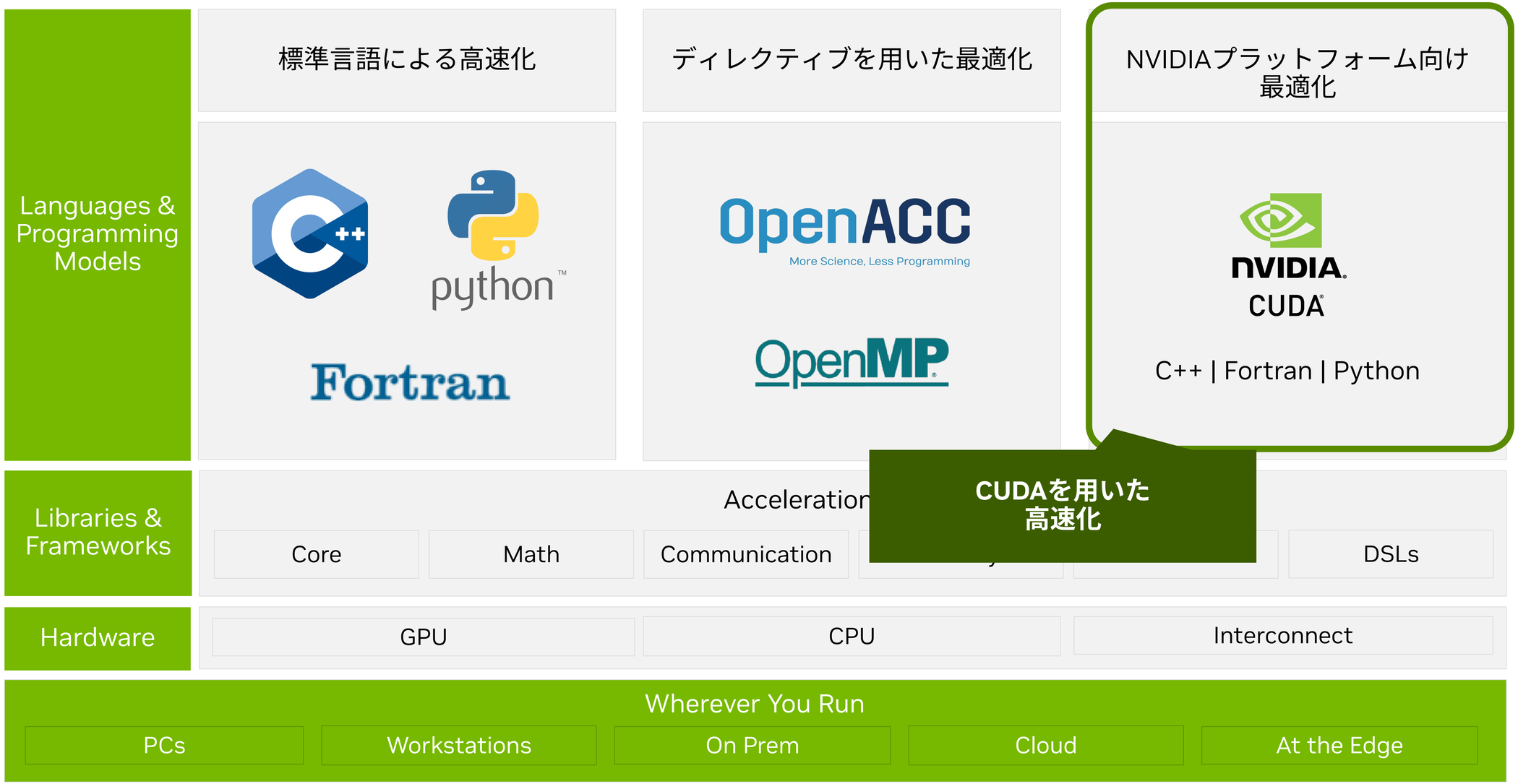
NVIDIA プラットフォーム上でのプログラミング

GPU アクセラレーションの実現方法



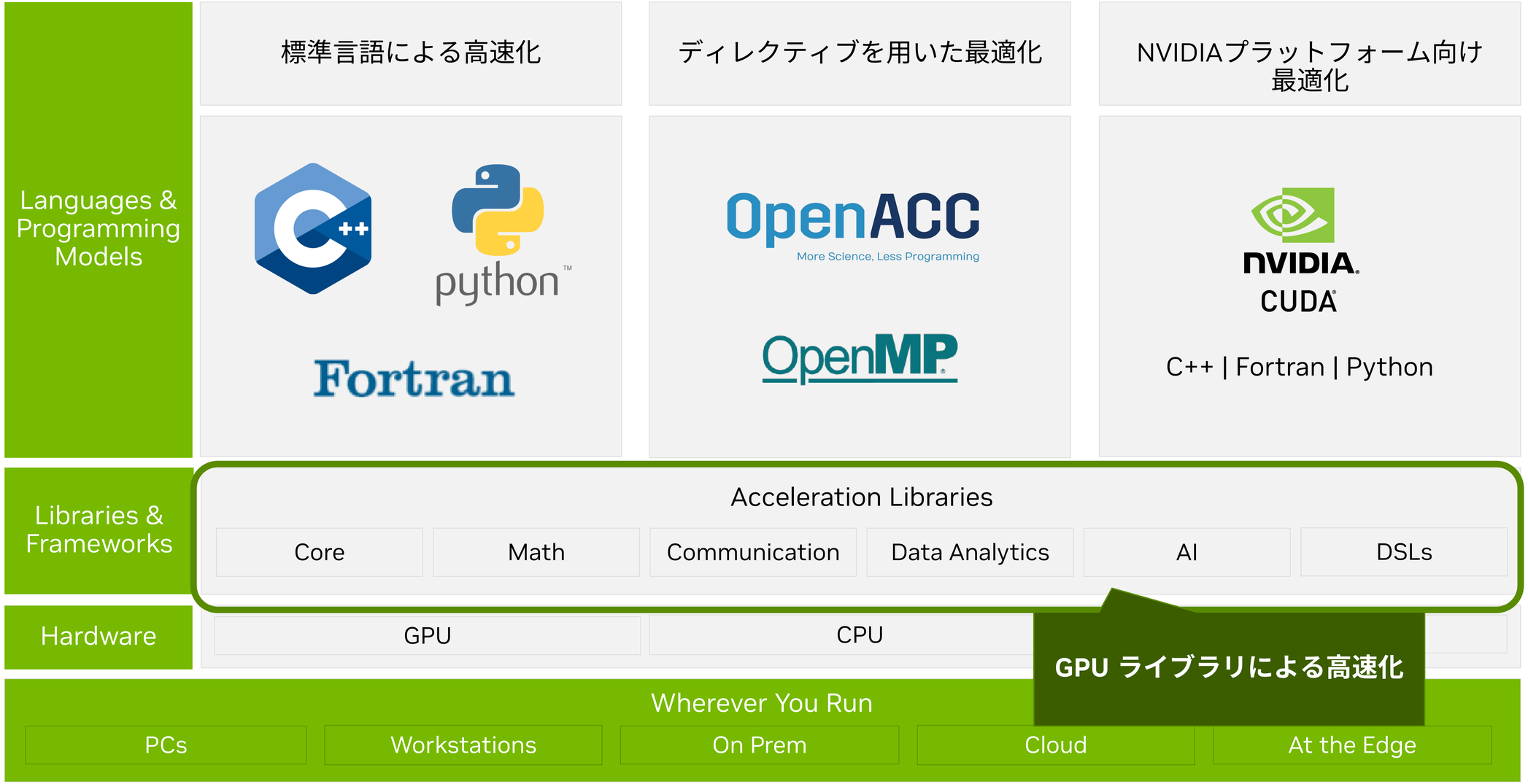
NVIDIA プラットフォーム上でのプログラミング

GPU アクセラレーションの実現方法



NVIDIA プラットフォーム上でのプログラミング

GPU アクセラレーションの実現方法



GPU アクセラレーションの実現方法

GPU ライブラリ

ライブラリを呼び出すだけで、高速化が可能

ライブラリとして提供されている機能のみ高速化が可能

OpenACC

既存のC言語やFortranのコードにディレクティブを挿入するだけで簡単に高速化

最適化はコンパイラが行う為、細かいチューニングを行う事は出来ない

CUDA

自由度が最も高く、細かいチューニングが可能

CUDAでのプログラミングを学ぶ必要がある

CUDA

CUDA の仕組み

GPU の並列処理の仕組み

超並列 (Massively Parallel)の重要性

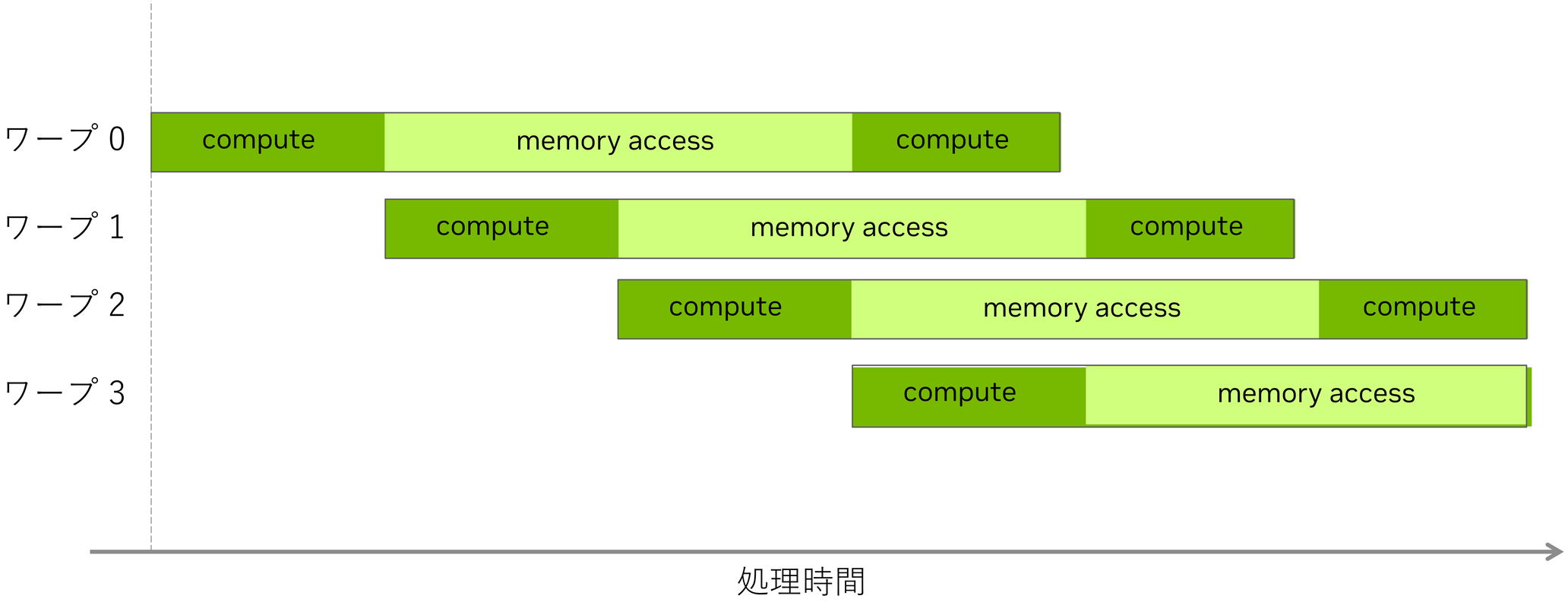
- CUDA スレッドは通常100万単位で作成する
  - 物理的な CUDA コアの数を意識せず十分に多くのスレッドを作成
    - CUDA による抽象化によりHWとしての CUDA コアの数を意識ぜずプログラミングをすることが可能
    - 例: 1920x1080の画像に対してCUDA によるフィルタ処理を行う場合、207万スレッド立ち上げて、各スレッド1pixelの処理を担当させる

- 十分に多くのスレッドを作成し、GPU 利用率を上げる
  - スレッドの処理単位は小さく
  - 多くのスレッドで超並列処理を行う

GPU の並列処理の仕組み

超並列 (Massively Parallel)の重要性

- Warp_mがメモリアクセス待ちをしている間に、その他のWarpが計算を実行
- メモリアクセスのレイテンシを隠蔽



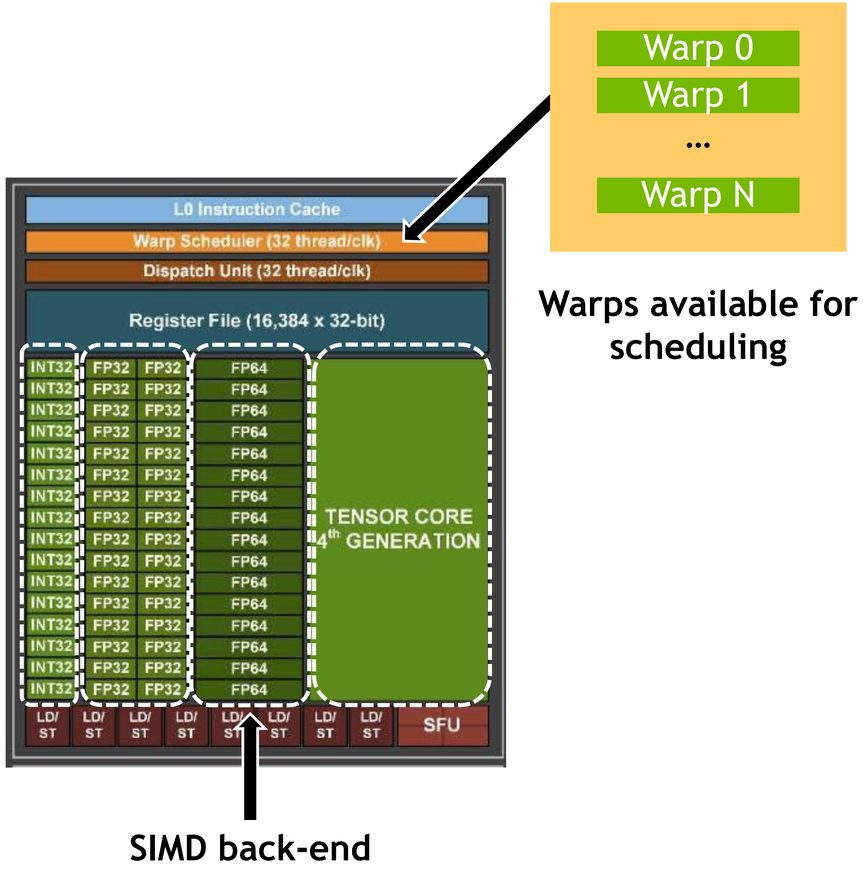
SIMT アーキテクチャ

シングル命令、マルチスレッド

- SIMTとはFlynnのコンピュータアーキテクチャ分類法を拡張したもの
- SIMD (シングル命令、マルチデータ)アーキテクチャに、よりきめ細かいマルチスレッドを組み合わせたアーキテクチャ。SIMTアーキテクチャでは単一スレッドではなく複数のスレッドが任意のデータに対して共通の命令を発行する

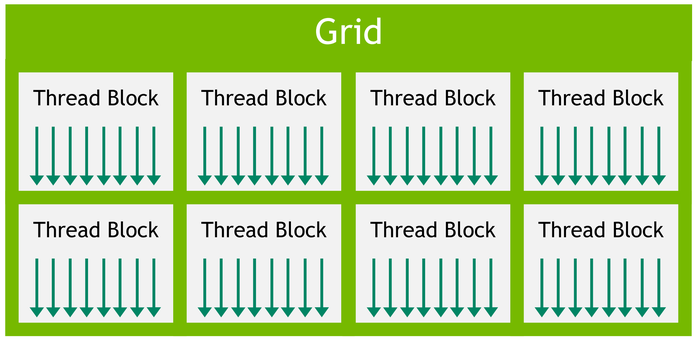
- SIMT アーキテクチャは、Warp(ワープ)と呼ばれるグループに分割されたハードウェアの大規模なセットを公開
  - レイテンシを隠すためにワープの実行をインターリーブする
  - 各ワープの実行コンテキストは、高速インターリーブのためにオンチップに保持される
- スケジュールされると、ワープの各スレッドはSIMDファンクションユニットの所定のレーンで実行される

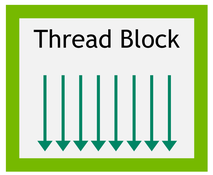
- 各SMサブパーティションは、32並列スレッドのワープを作成、管理、スケジュール、実行するSIMTエンジンと考えることができる



スレッド階層

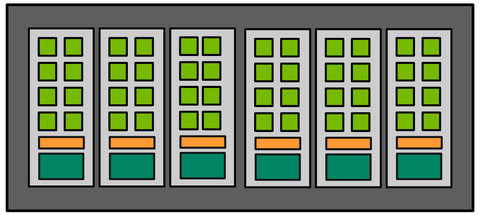
CUDA/ソフトウェア



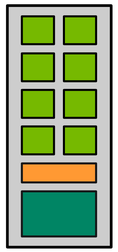


Thread

ハードウェア



Device



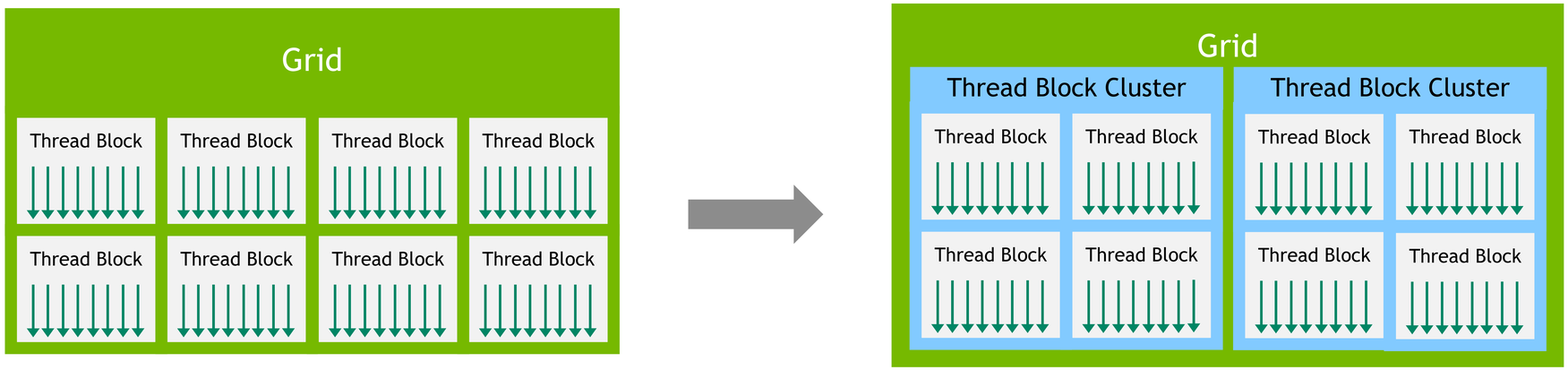
SM

Scalar CUDA core

- CUDAカーネルは、スレッドブロックのグリッド上で起動される。スレッドブロックは、完全に独立している
- スレッドブロックはSM上で実行される
  - 1つのSM上に複数のスレッドブロックを同時に配置することができる
  - スレッドブロックは割り当てられた場所からマイグレーションする事はない
  - 各ブロックは、利用可能な SM のいずれかに、任意の順序で、同時または直列にスケ ジュールされる
- 個々のスレッドはCUDAコア上で実行される

Thread Block Clusters

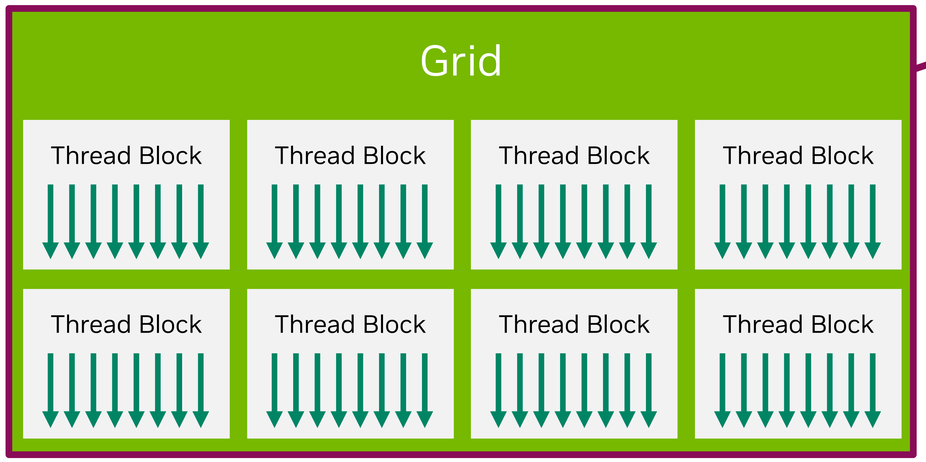
- Hopper世代から、CUDAは**Thread Block Clusters**と呼ばれるスレッド階層のオプションが新しく導入されました
- 複数のSM間の協調(スレッドの効率的な連携やデータ共有)を可能に
- このトピックに関する詳細は GTC セッション **[S62192]: “Advanced Performance Optimization in CUDA”**をご参照下さい
.



https://www.nvidia.com/en-us/on-demand/session/gtcfall22-a41095/

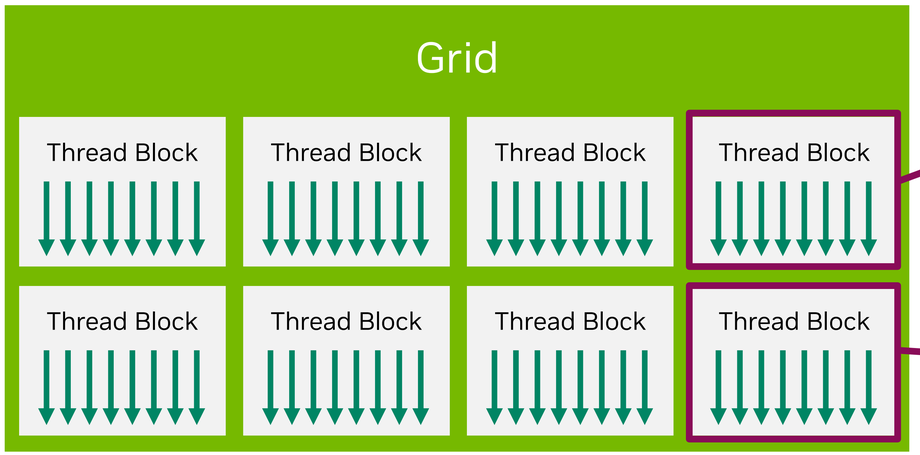
GPU カーネル実行の流れ

- CPU から GPU にグリッドを投入
- 具体的な投入先は Giga Thread Engine



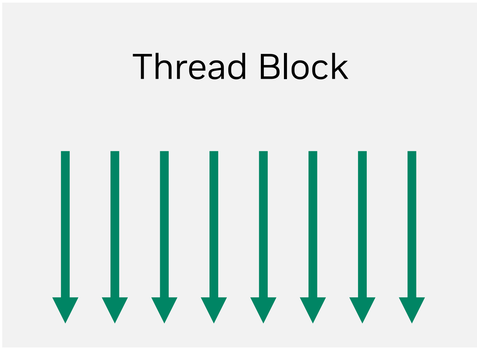
GPU カーネル実行の流れ

- ブロックを SM に割り当て
- 各ブロックは互いに独立に実行、実行順序の保証なし
- **1 つのブロックは複数 SM にまたがらない**
  - 1 つの SM に複数ブロックが割り当てられることはある



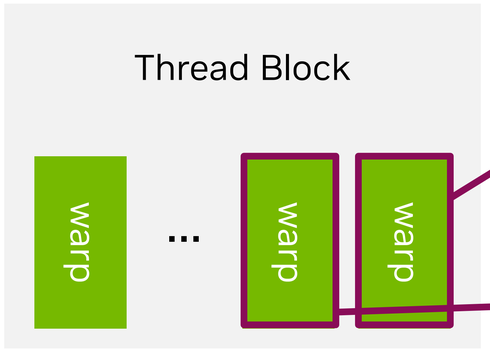
GPU カーネル実行の流れ

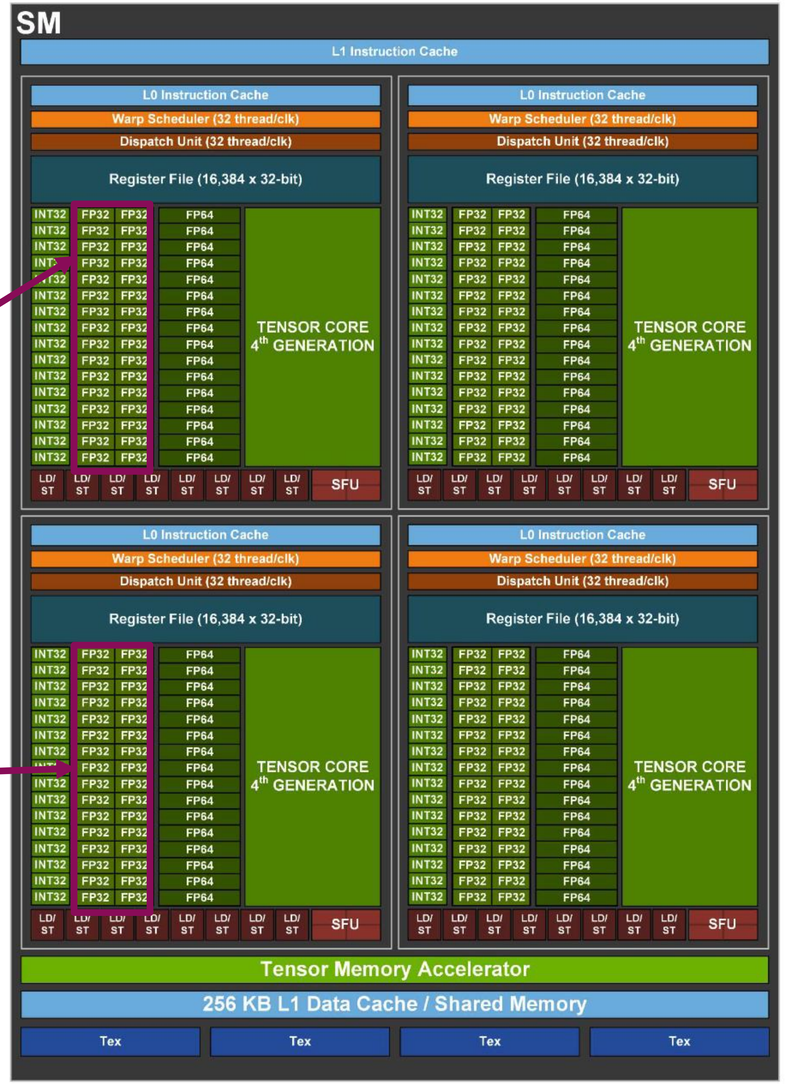
- SM 内のスケジューラが**ワープ**を CUDA コアに投入
  - ワープ: 32 スレッドのかたまり
  - ブロックをワープに分割、実行可能なワープを空 CUDA コアに割り当てる
- ワープ内に 32 スレッドは同期して同じ命令を実行
- 各ワープは互いに独立して実行
  - 同じブロック内のワープは、明示的に同期することも可能



GPU カーネル実行の流れ

- SM 内のスケジューラが**ワープ**を CUDA コアに投入
  - ワープ: 32 スレッドのかたまり
  - ブロックをワープに分割、実行可能なワープを空 CUDA コアに割り当てる
- ワープ内に 32 スレッドは同期して同じ命令を実行
- 各ワープは互いに独立して実行
  - 同じブロック内のワープは、明示的に同期することも可能

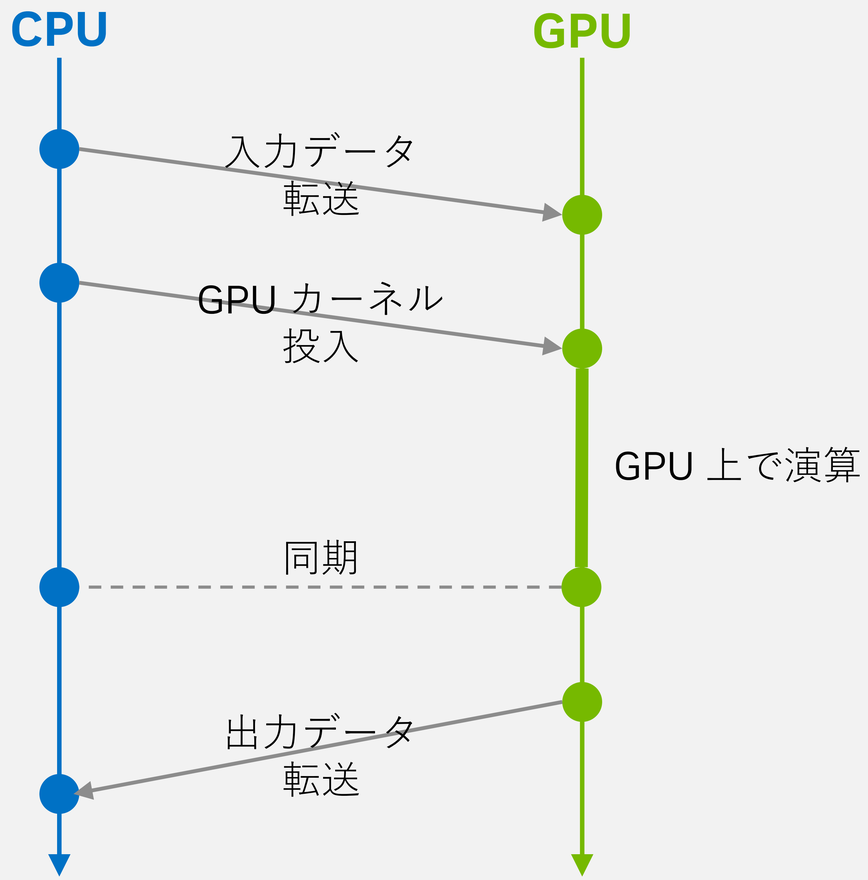




CUDA C++

GPU 実行の基本的な流れ

- GPU は CPU からの制御で動作
- 入力データ : CPU から GPU に転送 (H2D)
- GPU カーネル : CPU から投入
- 出力データ : GPU から CPU に転送 (D2H)



SAXPY (Y=A*X+Y)

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}



...
saxpy(N, 3.0, x, y);
...
```

CUDA

```
__global__ void saxpy(int n, float a,
           float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}


...
size_t size = sizeof(float) * N;
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

SAXPY (Y=A*X+Y)

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}


..
sa
..
```

CUDA

GPUカーネル

```
__global__ void saxpy(int n, float a,
           float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}

...
size_t size = sizeof(float) * N;
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

threadIdx: スレッドID
blockDim: ブロックサイズ
blockIdx: ブロックID

データ転送(H2D)

カーネル投入 <<<GS,BS>>>
GS: グリッドサイズ(ブロック数)
BS: ブロックサイズ(スレッド数)

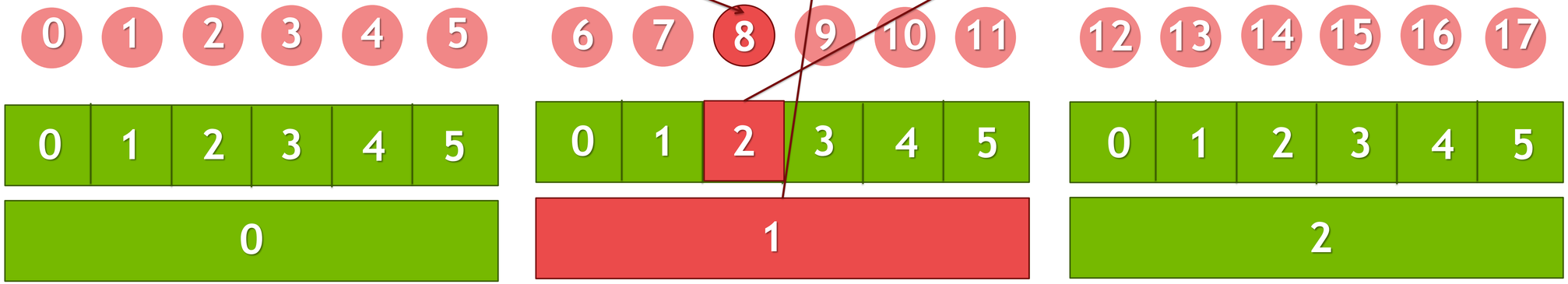
データ転送(D2H)

ブロックIDとスレッドID

- ブロックIDとスレッドIDから、インデックス (グローバルID) を生成する
- インデックスを用いて各スレッドから、グローバルメモリへアクセスする

```
index = blockDim.x * blockIdx.x + threadIdx.x;
```

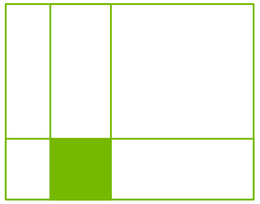
8　6　×　1　+　2



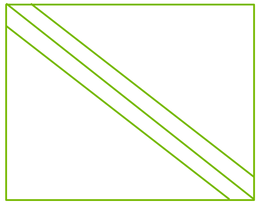
CUDA ライブラリ

NVIDIA 数学ライブラリ

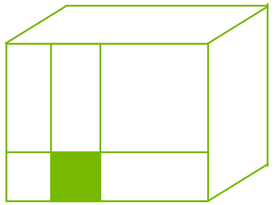
Linear Algebra, FFT, RNG, and Basic Math



cuBLAS

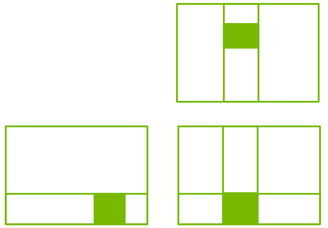


cuSPARSE

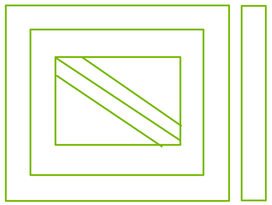


cuTENSOR

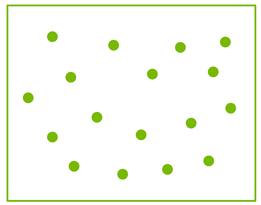
cuSOLVER



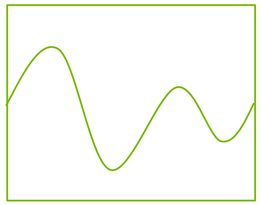
CUTLASS



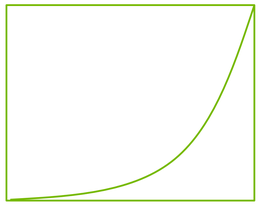
AMGX



cuRAND



cuFFT



Math API

CUDA ライブラリ – 数値演算・信号処理 -

柔軟性、使いやすさ、性能の架け橋

様々なタイプの開発者の為のGPUで高速化された数値演算ライブラリ

Python Bindings to CUDA

LLVM to CUDA

PYTHONISTA

CuPy

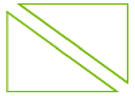
Numba

PyTorch

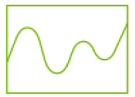
TensorFlow

C/C++ GURU

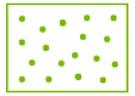
cuBLAS



cuSOLVER



cuFFT



cuRAND

CUTLASS

Optimized CUDA Libraries

MatX – 数値計算のためのC++17テンプレートライブラリ

https://github.com/NVIDIA/MatX

特徴

👍 Ease of Use:

使い慣れたインターフェース(MATLAB/Python-like)による素直なプログラミングモデル
Straightforward programming model with familiar interfaces

cuFFT, CUTLASS, cuRANDのような既存ライブラリをラップ

簡単にカスタマイズ可能

🔥 High Performance:

ストリーミングデータを効率的に処理する為の優先順位付け

アロケーションとプロセス処理の分離

主要コンセプト

MatXはCUDAマネージドメモリを活用し、開発者はデータローカリティの心配から解放される

計算操作はホストまたはデバイス上のデータ記述であるテンソルで実行される

テンソル はNumpy ndarrayのように、すべてのMatX関数で受け入れられます

Clone,Slice,Permute等の操作時にデータ移動がゼロになるようにコンパイル時にチェーンを生成

多くの変換をサポート: FFT, 畳み込み, GEMM, pointwise演算,etc..

最小限のコード変更でホストとデバイスの実行をサポート

MatX/Python比較

FFT Based Resampler – No Windowing

Python

```
import numpy as np
from numpy import fft as fft

N = min(num_samp, num_samp_resamp)
nyq = N // 2 + 1

# Create an empty vector with num_samps elements
sig = np.empty(num_samp)

# Real to complex FFT, time to freq domain
fft_sig = fft.rfft(sig)

# Slice
slice_sig = fft_sig[0:nyq]

# Complex to real IFFT
resamp_sig = fft.irfft(slice_sig,
num_samp_resamp)
```

5.36s (Xeon E5-2698v4 @ 2.20GHz)

MatX

```
uint32_t N = std::min(num_samp, num_samp_resamp);
uint32_t nyq = N / 2 + 1;

auto sigView        = make_tensor<float, 1>({num_samp});
auto sigViewComplex = make_tensor<complex, 1>({num_samp/2+1});
auto resampView     = make_tensor<float, 1>({num_samp_resamp});

// Real to Complex FFT, time to freq domain
(sigViewComplex = fft(sigView)).run(stream);

// Slice to half spectrum based on num_samp_resamp
auto sliceView = slice(sigViewComplex, {0}, {nyq});

// Complex to Real IFFT, back to time domain
(resampView = ifft(sliceView)).run(stream);
```

5.48ms (V100)

~1000x improvement!

柔軟性、使いやすさ、性能の架け橋

様々なタイプの開発者の為のGPUで高速化された数値演算ライブラリ

Python Bindings to CUDA

LLVM to CUDA

PYTHONISTA

CuPy

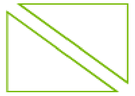
Numba

PyTorch

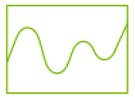
TensorFlow

C/C++ GURU

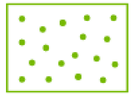
cuBLAS



cuSOLVER



cuFFT



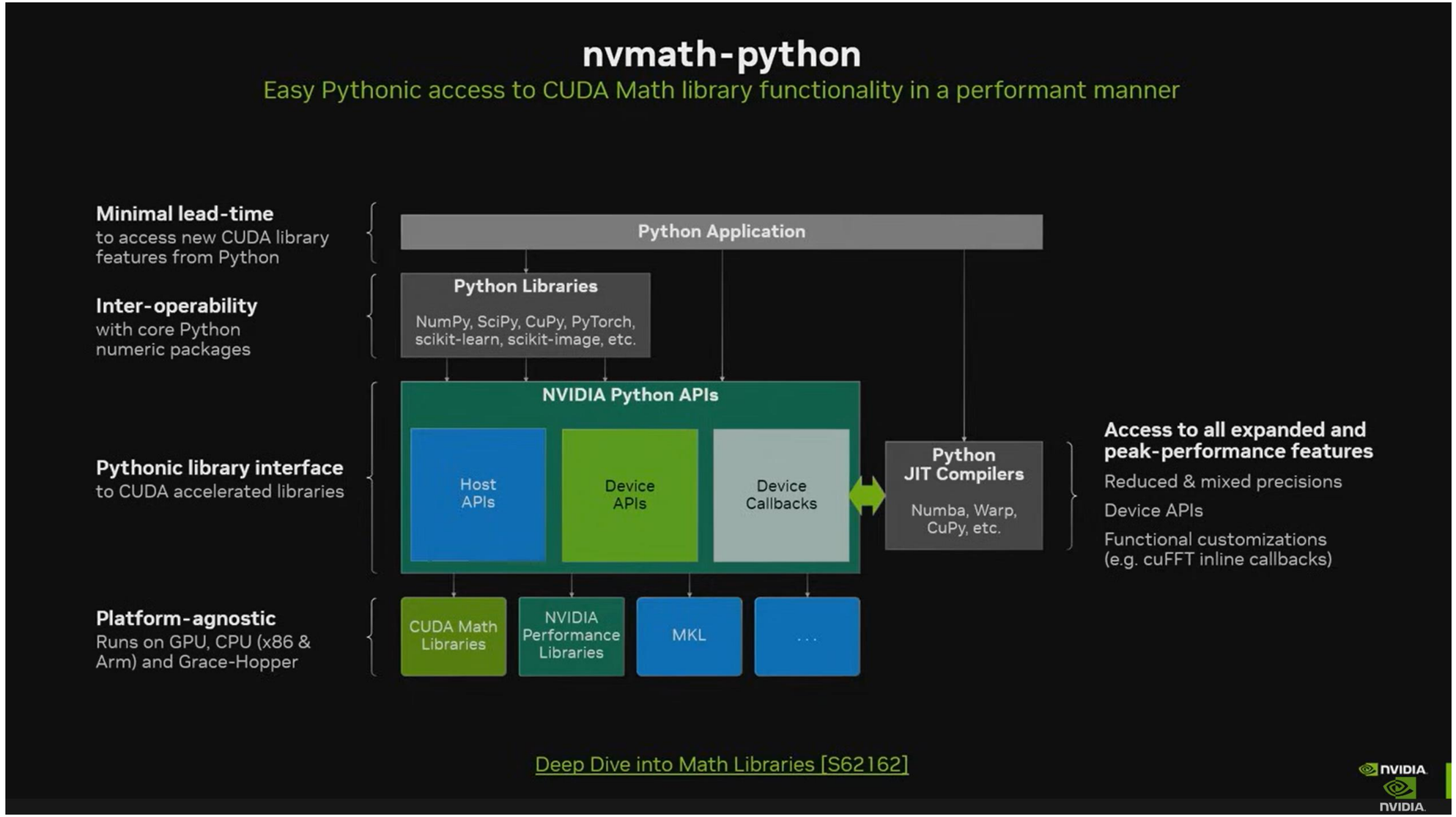
cuRAND

CUTLASS

Optimized CUDA Libraries

CUDA Pythonエコシステムの強化

[S62400] CUDA: New Features and Beyond at GTC 2024



CuPy

PythonによるGPU高速計算のためのNumPy/SciPy互換Arrayライブラリ

- CuPy はNumpy互換のAPIを持つオープンソースのライブラリで、NVIDIA GPUによるN次元配列計算で高い性能を発揮する.
- NumPyから容易に置き換え可能
- https://github.com/cupy/cupy

With NumPy

```
import numpy as np
x = np.random.rand(10)
W = np.random.rand(10, 5)
y = np.dot(x, W)
```

With CuPy

```
import cupy as cp
x = cp.random.rand(10)
W = cp.random.rand(10, 5)
y = cp.dot(x, W)
```

CuPy

CuPy Raw Kernel 機能

https://docs.cupy.dev/en/stable/user_guide/kernel.html#raw-kernels

- ユーザ定義のCUDA Kernelを動かす事ができる強力な機能。プロトタイプ開発時、動作確認を迅速に行いたい時に便利
- Raw Kernel サンプルコード
  - https://github.com/cupy/cupy/tree/main/examples/gemm

```
import cupy as cp
square_kernel = cp.RawKernel(r'‘’
extern "C" __global__ void my_square(long long* x) {
 int tid = threadIdx.x;
 x[tid] *= x[tid];
}
''', 'my_square’)
x = cp.arange(5)
square_kernel((1,), (5,), (x,))  # grid, block and arguments
print(x)                         # [ 0 1 4 9 16]
```

CuPy

cupy.scipyx.signal – Selected Algorithms

GPU-accelerated SciPy Signal (Python)

	Convolution	Convolve/Correlate FFT Convolve Convolve/Correlate 2D
	Filtering and Filter Design	Resampling – Polyphase, Upfirdn, Resample Hilbert/Hilbert 2D Wiener Firwin, FIR Filter
	Waveform Generation	Chirp Square Gaussian Pulse
Ambgfun MVDR	Phased Array	
	Window Functions	Kaiser Blackman Hamming Hanning
	Peak Finding	
	Spectral Analysis	Periodogram Welch Spectrogram

Full List of Supported Functions – CuPy Docs

SciPy Signal – Polyphase Resampler

```
import numpy as np
from scipy import signal

start = 0
stop = 10
num_samps = int(1e8)
resample_up = 2
resample_down = 3

cx = np.linspace(start, stop, num_samps, endpoint=False)
cy = np.cos(-cx**2/6.0)

%%timeit
cf = signal.resample_poly(cy, resample_up, resample_down, window=('kaiser', 0.5))
```

2x Xeon E5-2600: 2.36 seconds

CuPy – Polyphase Resampler

```
import cupy as cp
import cupyx.scipy.signal as cusignal

start = 0
stop = 10
num_samps = int(1e8)
resample_up = 2
resample_down = 3

cx = cp.linspace(start, stop, num_samps, endpoint=False)
cy = cp.cos(-cx**2/6.0)

%%timeit
cf = cusignal.resample_poly(cy, resample_up, resample_down, window=('kaiser', 0.5))
```

NVIDIA A100: 4.69 milliseconds, **503x SciPy Signal (CPU)**

本日のまとめ

- NVIDIA GPUはデータセンターからエッジまで同一のアーキテクチャ、Data Center GPUからJetsonまで同一のCUDA kernel 関数を実行する事ができる (※クロスコンパイルは必要です)
- NVIDIA GPUはアーキテクチャ更新の度に計算性能が向上する方向に進化
- GPU コンピューティングはディレクティブからCUDAまで様々な手段を提供、CUDAは複数のプログラミング言語に対応
- CUDAのスレッド階層は、NVIDIA GPUのハードウェアとしての特徴を考慮しながら上手に抽象化
- 最初からすべてをCUDA C/C++で実装するのではなく、CUDAライブラリの活用も考える。プロトタイプはPythonで開発するのも便利

Appendix.

Useful links

- CUDA toolkit documentation (cuda-c-programming-guide)
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Samples
  - https://github.com/NVIDIA/cuda-samples
- [S62400] CUDA: New Features and Beyond (GTC 2024 talk)
  - https://www.nvidia.com/en-us/on-demand/session/gtc24-s62400/
- [A21118] GPU Acceleration in Python (GTC fall 2020 talk)
  - https://www.nvidia.com/en-us/on-demand/session/gtcfall20-a21118/
- [A31149] GPU Acceleration in Python using CuPy and Numba (GTC fall 2021 talk)
  - https://www.nvidia.com/en-us/on-demand/session/gtcfall21-a31149/
- [A21103] GPU-Accelerated End-to-End Signal Processing with Python (GTC fall 2020 talk)
  - https://www.nvidia.com/en-us/on-demand/session/gtcfall20-a21103/
- ICASSP '21 Tutorial: GPU-Acceleration of Signal Processing Workflows from Python
  - https://github.com/awthomp/cusignal-icassp-tutorial
- CuPy 公式
  - https://github.com/cupy/cupy
  - https://cupy.dev/
  - https://docs.cupy.dev/en/stable/user_guide/kernel.html#raw-kernels (CuPy Raw Kernel)